> Prequisites:-

The term data structure is used to describe the way the data is stored
The algorithm is used to describe the way it is processed

so Algorithm and data structures are Interrelated

Choosing a different datastructure affects the algorithm we might use and algorithm choosen affects the datastructure we choose

Algorithm

The method of solving a problem is called Algorithm

It is a sequence of instructions that act on input data to produce some output in a finite number of steps is called an algorithm

An Algorithm must have following Properties

- Input
  - An algorithm requires an input supplied externally
- Output
  - It must produce atleast one output
- finiteness
  - No matter what it must terminate at some point
- definiteness
  - It must be clear and unambiguous and should be understood by a kindergarten student
- effectiveness
  - One must be able to perform the steps without any intelligence

Data structure

It is a logical relationship existing between individual elements of data

** Algorithm + datastructure = program

ADT (Abstract datatype)

It is a mathematical model or concept or that defines a data type logically

data structure is a programming construct used to implement an ADT

It is the physical or actual representation of ADT

- List ADT
- Stack ADT
- quee ADT

Three Adt's are differentiated by types of operations performed on them

ADT

Data structure

ADT tells us what is to be done and data structures tells us what and how it is to be done

A program that generally uses a data structure is called as a client. The specification of ADT is called the interface and it is the only thing that is visible to the client programs that uses the data structure

Advantages of data structure

efficiency

Proper choice of data structures makes our program efficient

reusability

data structures are reusable; once they are implemented they can be used at any other place

Abstraction

The client program uses the data structure through only interface only without getting into the implementation details

Linear data structure & non linear data structure

If all the elements are arranged in sequential order then it is called linear data structure

In linear data structure each element has a only one predecessor and successor

except the last and first elements as first element has no predecessor and last element has no successor

array, linked list, string, stack and quee are examples

If there is no linear order arrangement of elements then it is called non linear data structure

trees and graphs are examples

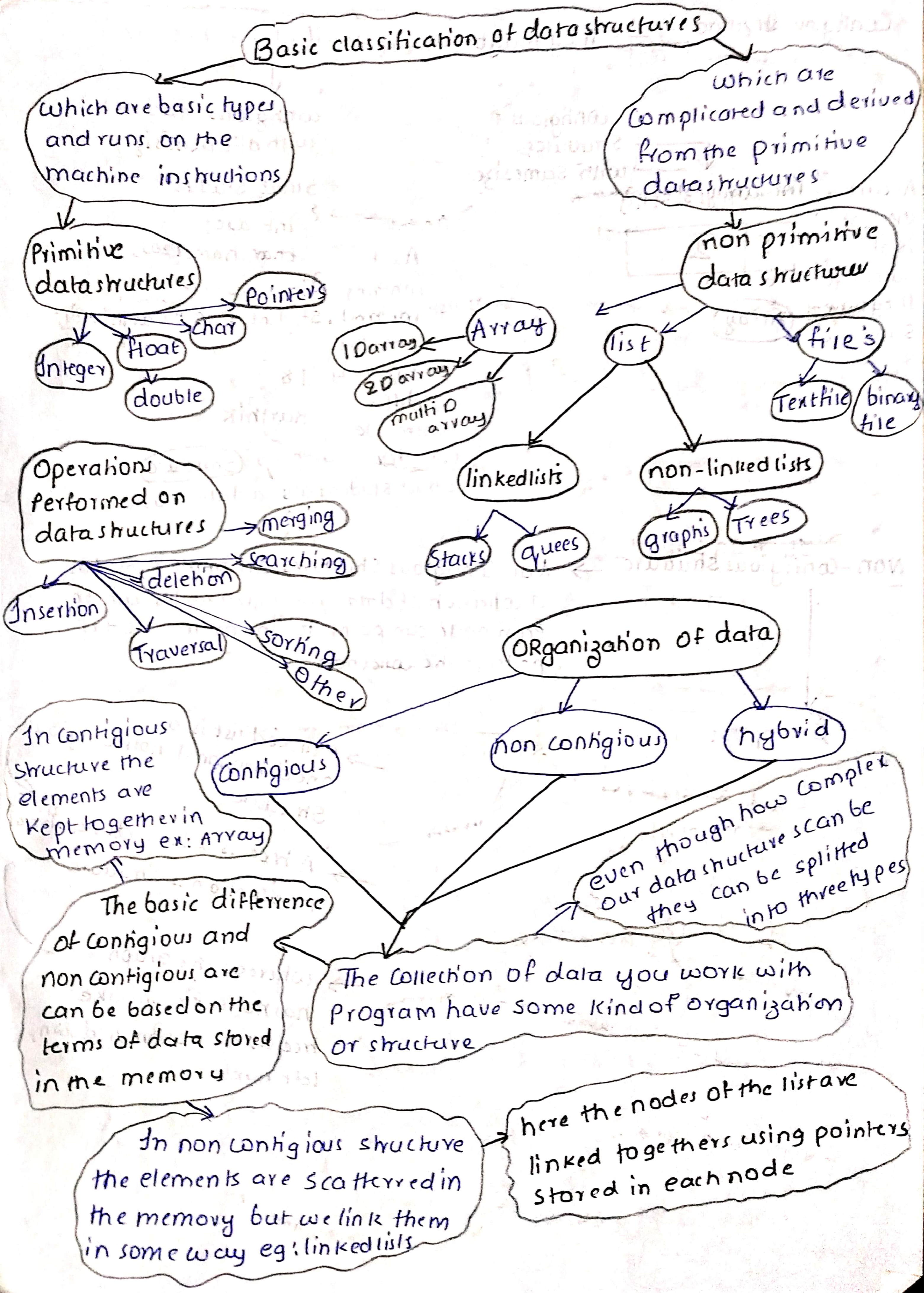
Based on memory there are two types of data structures

static data structure

memory is allocated at compilation and maximum size is fixed

dynamic data structure

memory is allocated at run time and maximum size can be changed at run time



Basic classification of datastructures

- which are basic types and runs on the machine instructions
  - Primitive datastructures
    - Integer
    - float
    - double
    - char
    - Pointers
- which are complicored and derived from the primitive datastructures
  - non primitive datastructures
    - Array
      - 1D array
      - 2D array
      - multi D array
    - list
      - linkedlists
        - Stacks
        - quees
      - non-linked lists
        - graphs
        - Trees
    - file's
      - Text file
      - binary file

Operations performed on datastructures
- merging
- searching
- deletion
- Insertion
- Traversal
- sorting
- Other

ORganization of data
- Contigious
- non contigious
- hybrid

even though how complex our data structure scan be they can be splitted into three types

The Collection of data you work with Program have some Kind of organization or structure

In contigious structure the elements are kept together in memory ex: Array

The basic differrence of contigious and non contigious are can be based on the terms of data stored in the memory

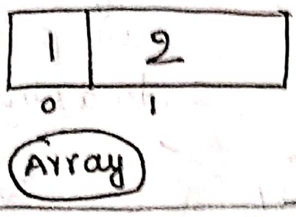
In non contigious structure the elements are scatterred in the memory but we link them in some way eg: linked lists

here the nodes of the list are linked togethers using pointers stored in each node

Contigious structure → They maybe simply classified into two types

Contigious structures with same size

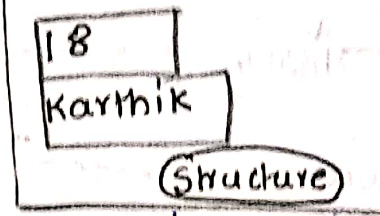
```
int array[3]={1,2}
```



As each element is of same type they have same size

Contigious structures with different size

```
struct student
{
  int age;
  char name[20];
};
Student s={18,"Karthik"};
```

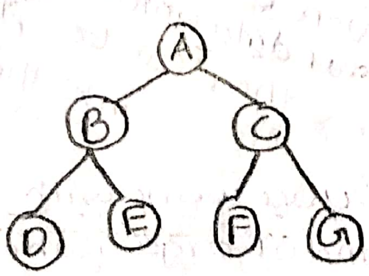


As it contains int and char the size may differ when we consider struct student as a data structure

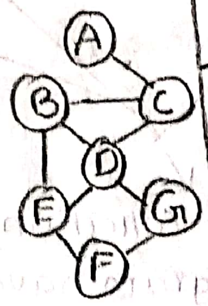
Non-contigious structure → Non Contigious structures are implemented as a collection of data items called nodes, where each node can point to one or more other nodes in the collection

Examples:

(a) Linked list



(b) Tree



(c) graph

A linked list is a one dimensional non contigious structure

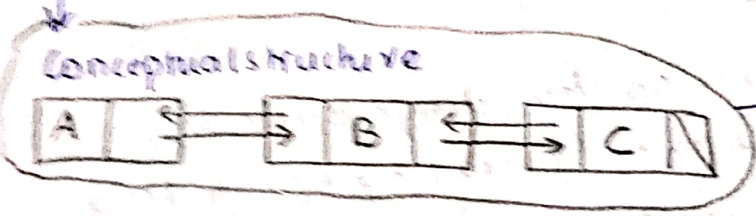
A tree is a two dimensional non contigious structure

whereas the graph has no restrictions like tree as to notion up, down, left right

Hybrid structure

If two basic types of structures are mixed then it is called a hybrid structure. Then one part is contigious and another part is non contigious

Conceptual structure



The implementation of a double linked list using three parellel arrays and stored a past from each other in memory

Problems faced when designing a practical Algorithm

These are also the basic design goals we must strive in a program

- Trying to save time [TIME complexity]
  - faster running of program is required
- Trying to saves space [space complexity]
  - Program that saves space is preferable
- Trying to have face
  - we want to prevent the program from generating loads of unwonted data

Approaches to design an algorithm

Top-down approach

- Generally structured oriented Programs use Top-down approach
- It is also called as step-wise approach
- In this approach the programmer needs to write code for the main-function, In main function they will call other functions
- It requires the good understanding of program

bottom-up approach

- Generally Object oriented programming languages use this approach
- In this approach the Programmer has to write code for the modules. Then they have to look for the integration of modules
- It is suitable for the projects that can be developed from existing projects

Note:

- Top down approach first focuses on abstract of overall system or project. At last it focuses on detail design on development
  - It goes from high level design to low level design (or) system (Top-down approach)
- Bottom-up approach first focuses on detail design on development At last it concentrates on the abstract of overall system or design
  - Bottom up approach starts from low level design to high level design or system

*nowadays moder software designs combine both of these approaches for the better results

Control structures used in Algorithm statements

- sequence statements
  - PRINT, READ, CALULLATE etc
- Selection structures (or) Branching statements
  - *If
  - *If-else
  - *If-else-if
  - *Nested if
  - *Switch
  - *Jump·st
- Iteration structures or looping statements
  - * while
  - * do while
  - * for

Performance of a program: → is amount of computer memory & time to run a program

we use two approaches to determine the Performance of a program

- Analytical (or) logical approach
  - In analysis we use this method
  - In which we approximately calculate the memory, space using the logical approach
  - The results of the logical approach may vary with actual figures
  - due to the platform dependencies and machines
- Experimental approach
  - But while measurement we use this approach
  - In which we calculate exact amount of time, memory accurately
  - These are generally actual figures

Time complexity

Time needed by algorithm expressed as a function of size of problem is called time complexity of an algorithm

It is the time required for the completion of program

The limiting behaviour of the complexity as size increases is called asymponic time complexity

Asymponic time complexity ultimately determines the size of problems that can be solved by algorithm

Space complexity

is the amount of memory required by a program for its completion

Components of program occupying space

Instruction space

It is the space needed to store the compiled version of program instructions

Data space

It is the space needed to store all constant and variable values

It is of two types

Space needed by constants and variables

Space needed by dynamically allocated objects such as arrays and class instances

which may be dynamically allocated at the run time at the convinience of the user

Environment stack space

is used to save information needed to resume execution of partially completed functions

Instruction space

The space required depends on some factors

The compiler used to complete the program into machine code

The compiler options in effect at the time of compilation

The target computer

classification of the algorithm

If n is the number of data items to be processed or degree of polynomial or the size of file to be sorted or searched or nodes in a graph

1

Next instructions of most programs are executed at once or mostly a few times, then we say the running time is constant

logn

The program gets slightly lower as n grows. this running time constantly occurs in the programs that solve a big problem by transforming into smaller problem. whenever n doubles logn increases by a constant, but logn never doubles until n becomes n^2

n

This occurs when running time is linear, In this case small amount of processing is done on each input element

n·logn

This running time occurs when we solve problem by breaking up into smaller problems and solve them independently. If n doubles, running time more than doubles

n^2

When the running time of algorithm is quadratic, it can be used practically on relative small problems. In case of double nested loop, if n increases running time increases four fold. But in case of n^3 it increases 8 fold times. And these process tripples of data (eg: triple nested loop).

2^n

These algorithms with exponential running time are appropriate for practical use. such algorithms are brute-force solution to problems. If n doubles, running time squares complexity of algorithm

Complexity shall refer to the running time of algorithm

Complexity function f(n) which gives running time / storage of algorithm in terms of size 'n' of input data.

The storage space required by algorithm is simply a multiple of data of size 'n'

The complexity of an algorithm also depends on the particular data

Best case

The minimum possible value of f(n) is called best case

Average case

The expected or predicted value of f(n)

Worst case

The maximum value of f(n) for any f(n) for any key possible input

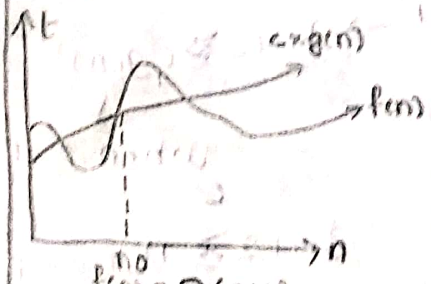
The feild of computer sceince which studies efficiency of algorithms is called analysis of algorithms

Rate of growth

These notations are used in performance analysis

Also use to chategorize complexity of algorithm

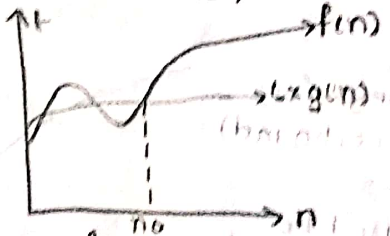
Big-OH (O)
(upper bound)



f(n) = O(g(n))

f(n) = O(g(n))
{Pronounced as order of (on) big oh} says that growth rate of f(n) is less than or equal (≤) that of g(n); n ≥ n0; c > 0; n0 ≥ 1

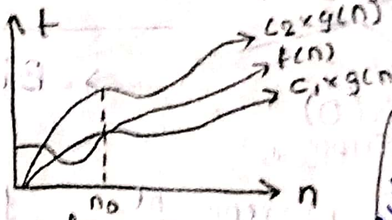
Big omega (Ω)
(lower bound)



f(n) = Ω(g(n))

f(n) = Ω(g(n)) say that growth rate of f(n) is greater than or (≥) equal to that of g(n)

Big-Theta (Θ)
((same) order)



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

f(n) = Θ(g(n))

f(n) = Θ(g(n)) says that growth rate of f(n) equals to the growth rate of g(n).
[If f(n) = O(g(n)) and T(n) = Ω(g(n))]

Little-OH (o)
T(n) = o(P(n))

Say's growth rate of T(n) is less than P(n) if T(n) = O(P(n)) and T(n) ≠ Θ(P(n))

Proof for Big Oh [O]

let us consider $f(n) = 3n+2$ and $g(n) = n$

To prove $f(n) = O(g(n))$ we need to find 2 constants c, n_0

$\therefore f(n) \leq c(g(n))$, $c>0$; $n_0 \geq 1$

$3n+2 \leq cn$
If $\boxed{c=4}$ the condition satisfies

$3n+2 \leq 4n$
$\therefore n \geq 2$

$\therefore f(n) = O(g(n))$

Proof of Big omega (Ω)

let us consider $f(n) = 3n+2$ and $g(n) = n$

To prove $f(n) = \Omega(g(n))$ we need to find 2 constants c and n_0

$f(n) \geq c\, g(n)$
$c>0$, $n_0 \geq 1$

$3n+2 \leq cn$
If $c=1$ the condition satisfies

$3n+2 \geq n$

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

case 1

we cannot take any value lower bounded by n^2 but we can take a value upper bounded by n^2 and values less than n like $\log n$

Proof of Theta (Θ)

let us consider $f(n) = 3n+2$ and $g(n) = n$

upperbound → $3n+2 \leq \underbrace{\textcircled{4}}_{c_2}(n)$, $n_0 \geq 1$

lowerbound → $3n+2 \geq c_1 g(n)$

when $c=1$

$3n+2 \geq n$, $n_0 \geq 1$

Uses of the notations

Big oh (O) (upper bound)

It is used to represent the worst case of an algorithm. It is the maximum time taken and it does not exceed this

In practice we actually tend to find the worst case rather than the best case

gives max time

Big omega (Ω) (lower bound)

It is used to represent the best case of an algorithm. In any case we cannot acheive better than this

We use this to show the least possible time taken by algorithm for it's execution

Big-Theta (Θ)

It is used to represent the average case of an algorithm

We use average case and if both best case and worst case are same time i.e, takes same time to any input

Rules for O notation → Transitivity → If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $O(h(n))$

If $f_1(n)$ is $O(h(n))$ and $f_2(n)$ is $O(h(n))$ then $f_1(n)+f_2(n)$ is $O(h(n))$

If $f_1(n)$ is $O(h(n))$ and $f_2(n)$ is $O(g(n))$ then $f_1(n)+f_2(n)$ is $\max(O(h(n)), O(g(n)))$

If $f_1(n) = O(h(n))$ and $f_2(n) = O(g(n))$ then $f_1(n) f_2(n) = O(h(n)\, g(n))$

If $f(n) = c \Rightarrow f(n)$ is $O(1)$

If $f(n) = c * h(n)$

Any polynomial $p(n)$ of degree 'm' is $O(n^m)$

n^a is $O(n^b)$ only if $a <= b$

All logarithms grow at same rate while computing O notation, base of the logarithm is not important

let $xa = \log_a n$ and $yb = \log_b n$

$a^{xa} = n$ and $b^{yb} = n$

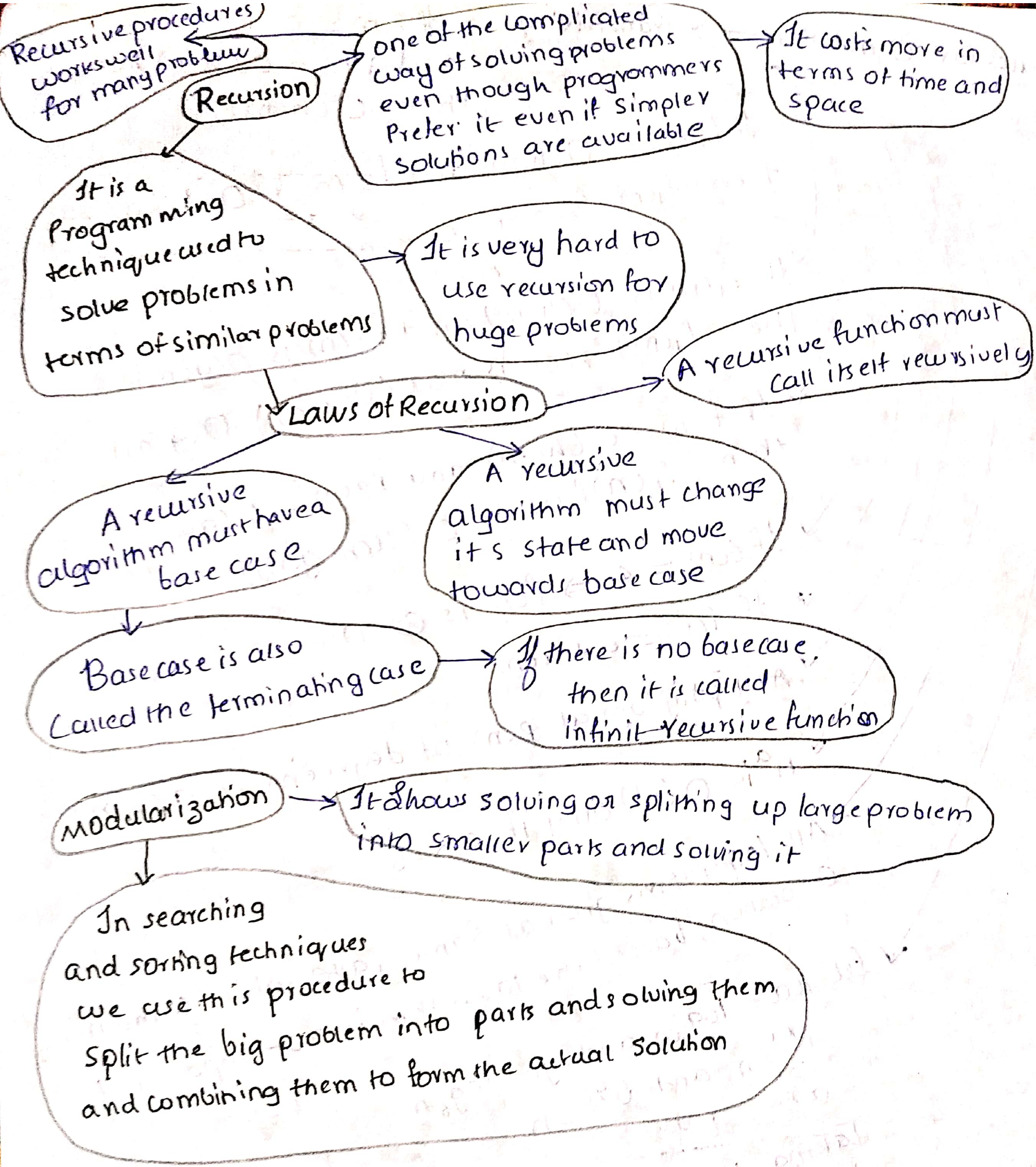
$a^{xa} = b^{yb}$

Taking $\log_a$ of both sides

$\log_a a^{xa} = \log_b b^{yb}$

$xa * \log_a a = yb * \log_a b$

$xa = yb * \log_a b$



Recursive procedures works well for many problems

Recursion

one of the complicated way of solving problems even though programmers Prefer it even if simpler solutions are available

It costs more in terms of time and space

It is a Programming technique used to solve problems in terms of similar problems

It is very hard to use recursion for huge problems

Laws of Recursion

A recursive function must call itself recursively

A recursive algorithm must have a base case

A recursive algorithm must change it's state and move towards base case

Base case is also called the terminating case

If there is no base case, then it is called infinit recursive function

Modularization

It shows solving or splitting up large problem into smaller parts and solving it

In searching and sorting techniques we use this procedure to split the big problem into parts and solving them and combining them to form the actual solution

⑨ **

```
//Program to find the factorial of a number using recursion
#include <stdio.h>
int factorial (int n);
void main()
{
 int num,fact;
 printf("\n enter a positive integer value");
 scanf("%d",&num),
 fact =factorial(num);
 printf("\n factorial = %.5d", fact);
}
int factorial(int n)
{ int result;
  if(n==0)
  return 1;
  else    result =n*factorial(n-1);
  return result;
 }
```

non-recursive function

```
factorial(int n),
{
 int i, result =1,
 if(n==0)
 return 1;
 else
 {
  for(i=1; i<=n; i++)
  result = result*i;
 }
 return result;
}
```

logic:-

if n=0, 0! =1 → Base case

if n>0, n! = n*(n-1)! → recursive case

let n=5

∴ 5! = 5*4! factr(5) = 5*factr(4)

4! = 4*3! factr(4) = 4*factr(3)

3! = 3*2! factr(3) = 3*factr(2)

2! = 2*1! factr(2) = 2*factr(1)

1! = 1*0! factr(1) = 1*factr(0)

0! = 1 factr(0) = 1

∴ 5! = 5*4!

= 5*4*3!

= 5*4*3*2!

= 5*4*3*2*1!

= 5*4*3*2*1*0!

= 120

Similarities involved in both recursion and Iteration

- Both of the techniques involve repetition
- Both of the solving methods have a terminating case or a condition
- Both can occur infinitely if there is no terminating condition

Differences between recursion and Iteration

recursion

It acheives repeleation through repeated function calls

It keeps Producing simple version of the problems until base case is reached

It terminates when a base case is recognized

It creates another copy of the function hence Considerable memory Space occupied

It increases processors Operating time

Iteration

It explicitly uses a repetition structure

It terminates when the Condition is not satisfied

It keeps modifying the counter until the loop Continuation statement fails

It occurs within a loop so no extra memory required

It reduces processors operating time

Towers of Hanoi

In towers of hanoi we use recursion technique

Types of recursion

Binary recursion

If the function repeats itself twice then it is called binary recursion

```
int rfibo(int n)
{
   if(n==1)
   return 0;
   else if(n==2)
   return 1;
   else
   return rfibo(n-1) + rfibo(n-2);
}
```

Tail recursion

It is a form of linear recursion

If recursive function is last case and there are no pending operations on return recursive call, then that function is said to be tail recursion

f(4,5)
→ f(3, 4*5)

```
#include <stdio.h>
int tailit(int n, int r);
void main()
{ int r=1;
  r = tailit(n,r);
}
int tailit(int n, int r)
{ if(n==1)
   return r;
   else
   return tailit(n-1,n*r);
}
```

Designing and developing a recursive function

STEP: 1

determine the Base case or nonrecursive case / simple case:

- It is the terminating condition of a recursive function
- It never contains a function call

STEP 2

determine the recursive case / general case
In which the program is expressed in terms of similar problems

step 3

Combine Base case and recursive case to complete recursive function

NOTE

The recursive function never contains any loops like while, do while

EXAMPLE logics

To print numbers from 1 to n using recursion

```
f(i,n) i=1
if(i=n) Print n
If i<n, Print i
        f(i+1,n)
```

To find x^y value using recursion

```
P(x,y)
If (y==0)
return 1
if (y>0)
x* p(x,y-1)
```

Applications of data structure

- numerical analysis
- operating system
- compiler design
- gaming software
- computer networks
- database management engine
- Artificial intelligence
- Simulation and modeling software

Towers of Hanoi

- It is an classic recursion game.

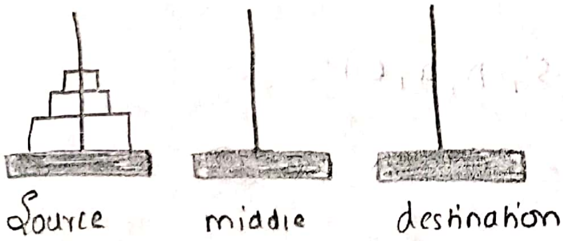
Rules

1. All disks must be of different size
2. only one should disk should be moved at a time
3. only top most disk can be moved
4. A larger disk cannot be placed on a smaller disk

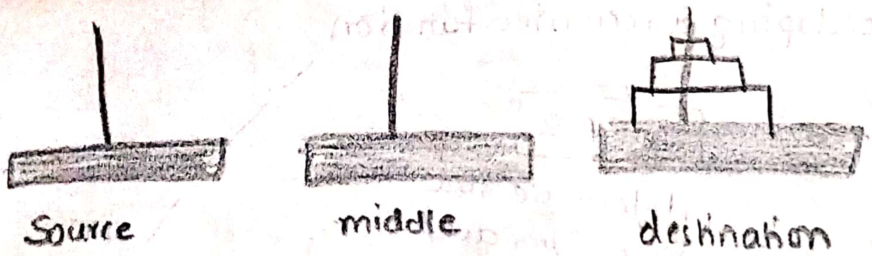
It contains 3 towers namely source, middle, destination

The target of the game is to move the disks from source tower to destination tower using middle tower

The towers of hanoi Problem can be easily solved using recursion.



Initial position of disks in hanoi towers game



Finalsetupof towers of hanoi.

```
// Program to solve towers of hanoi problem / game
#include<stdio.h>
void towers_of_hanoi(int n, char *a, char *b, char * c);
void main()
{ int n;
  Printf("\nenter how many disks you want : ");
  scanf("%d", &n);
  towers_of_hanoi(n,"source", "auxillary", "destination");
}
void towers_of_hanoi(int n, char *a, char *b, char *c)
{
  If (n==1)
  { Printf("\n %5d : move disk 1 from %s to %s",a,c);
  }
  else
  {
    towers_of_hanoi(n-1, a, c, b);
    Printf("\n move disk %d from %s to %s", n, a, c);
    towers_of_hanoi(n-1, b, a, c);
  }
}
```

★ If there are 'n' disks in the game then there will be $\boxed{2^n - 1}$ of steps

Fibonaccis equence problem → A fibonacci sequence starts with 0 and 1.

→ Successive elements are obtained by summing the preceeding two elements in sequence

∴ 0 1 1 2 3 5 8 13 21

→ A recursive function can be used to find n^{th} number in fobonacci sequence.

Fib(n) = n if n=0 or n=1
Fib(n) = fib(n-1) + fib(n-2) for n >= 2

eg: fibo[5] = fib[4] + fib[3]

fibo[3] + fibo[2] + fibo[3]

fibo[2] + fibo[1] + fibo[2] + fibo[3]

fibo[1] + fibo[0] + fibo[1] + fibo[2] + fibo[3]

* These are reduced to base case *

1 + 0 + 1 + fibo[1] + fibo[0] + fibo[3]

1 + 0 + 1 + 1 + 0 + fibo[3]

1 + 0 + 1 + 1 + 0 + fibo[2] + fibo[1]

1 + 0 + 1 + 1 + 0 + fibo[1] + fibo[0] + fibo[1]

1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 = 5

* fibo(2) ↓ 3 times

* fibo(3) ↓ 2 times

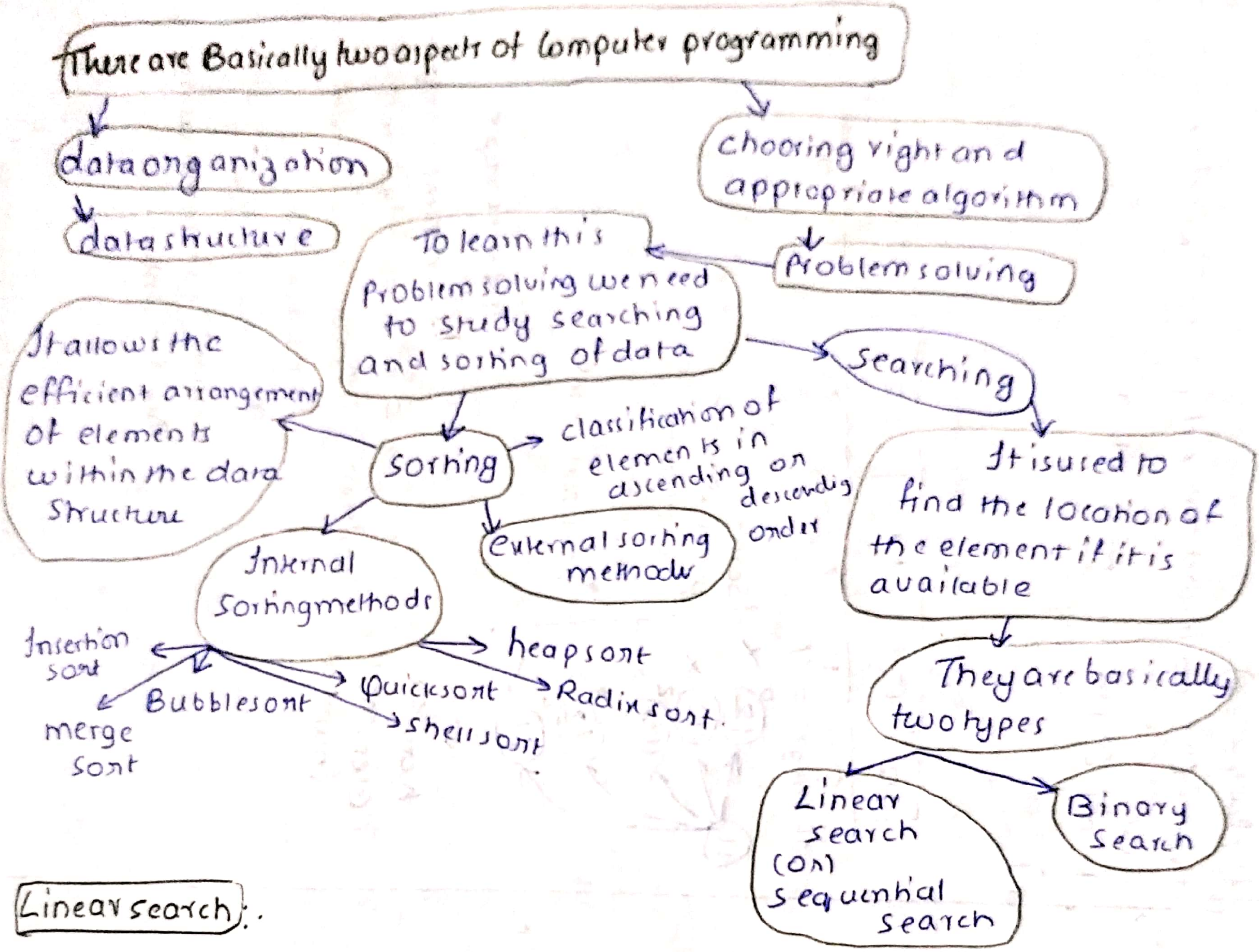
Note: we save the values of fibo[2] and fibo[3] and use them if needed later.

```
// recursive function to compute the fibonacci number using nth position given
#include <stdio.h>
int fibo(int x);
void main()
{
 int n, result;
 Printf("\n enter the position of in fibonacci series you want : ");
 Scanf("%d", &n);
```

```
  result = fibo(n);
  Printf("\n the number in %d position is %d", n, result);
}
int fibo(int x)
{
  if((x==0)||(x==1))
  return x;
  else
  return fibo(x-1)+fibo(x-2);
}
```

GCD of two numbers using recursion

```
// Program to find gcd of two numbers using recursion
#include <stdio.h>
int gcd(int x, int y);
void main()
{
   int x,y,result;
   Printf("\n enter any two numbers ");
   scanf("%d%d", &x, &y);
   result = gcd(x,y);
   Printf("\n gcd of %d and %d is %d", x,y,result);
 }
 int gcd (int x, int y);
 {
   while (x!=y)
   {
     if (x>y)
     return gcd(x-y,y);
     else
      return gcd(x,y-x);
   }
   return x;
 }
```



Linear search:

> It is the simplest of all searching techniques.

> In this technique ordered or unordered list be searched with the target value one by one from beginning till the desired element is found.

> If the desired element is found search is successful otherwise it is unsuccessful. → Suppose there are n elements organized on the list sequentially

The number of comparisions required to retreive an element purely depends on the where element is stored → If the search is unsuccessful you need 'n' comparisions.

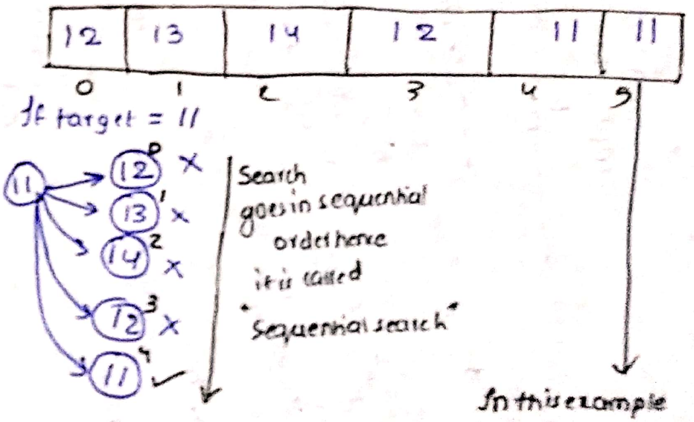
→ 1st element = target (or) 2nd (or) 3rd - - - - : $\left[\frac{(n+1)}{2}\right]$ average comparisions u need

Advantages → It is simple
→ It works for Ordered or unordered list

Disadvantages → It is only efficient when the number of elements are less

```
//Program to perform Linearsearch using non-recursion.
#include <stdio.h>
Void linearsearch (int A[], int  x); → correction **
                                  (int A[], int x, int n);
Void main ()
{
 int A[20],n, target,i;
 Printf("\n enter how many Value to store in array");
 Scanf("%d", &n);
 Printf("\n enter values into array");
 for(i=0;i<n;i++)
 {
   Scanf(" %d", & A[i]);
 }
 Printf("\n enter target Value tobesearched");
 scanf("%d", & target);
 linearsearch(A, target); →(A,target,n)
                 → Baseadress of Array 'A'
}
Void linearsearch(int A[], int x) → (int A[],int x,int n).
{
  int index =0,flag =0;
  While (index < n)
  { If(x = =a[index])
    { flag =1;
      break;
    }
    index++;
}
```

```
If (flag = = 1)
Printf("Data found at %d position", index);
else
Printf (" Data not found ");
}
```



Search goes in sequential order hence it is called "Sequential search"

here the element is found at 4th position

In this example the last element = target But we already found at 4th position we cannot find search Also the repeated elements cannot be searched using this mechanism.

```
// Program to perform Linear search using recursion.
#include <stdio.h>
void linearsearch(int a[],int data, int position, int n);
void main ()
{ int a[20],i,n,data;
  Printf("\n enter number of elements");
  Scanf("%d", &n);
  Printf("\n enter n elements");
  for(i=0; i<n; i++)
  { Scanf("%d",&a[i]);
  }
  Printf("\n enter element to be searched");
  Scanf("%d", &data);
  linearsearch(a,data,0,n);
}
void linearsearch(int a[],int data,int position,int n)
{
  If(position<n)
  {
   If(a[position]==data)
   Printf("\n data found at %d", Position);
   else
   linearsearch(a,data,position+1,n);
  }
  else
  Printf("\n data not found");
}
```

Efficiency of the linear search

Best case

If the target value is found at first element then it is Bestcase

$\therefore O(1)$

As $\left[\frac{n+1}{2}\right]$ Average comparisions required.

Average case

If the target value is found at middle

$\therefore 0+1+2 \text{--} n/2$

$\therefore \frac{n/2(n/2+1)}{2}$

$= n^2$

$\therefore O(n^2)$

(or)

$O(n)$

Based on Input*

WORSTCASE

If the targetvalue is last element (or) element is not Present

$\therefore O(n^2)$

(or)

$O(n)$

Based on Input*

Observation

for a linear search Average case and worst case are the same

Applications of Linear search

Although in practical we never use linear search But studying its method and comparing is needed