R18 Data Structures 18PC1IT01

Course objectives

1. To impart basic concepts of DS and algorithms
2. To introduce various searching and sorting techniques
3. To demonstrate operations of linear and non-linear DS
4. To develop an application using suitable DS.

Course outcomes:

1. Understand basic concepts of DS, computation and complexity.
2. Understand linear DS, various sorting and searching techniques
3. Understand various operations of linear and non-linear DS
4. Identify appropriate and efficient DS to implement a given proble

Syllabus:-

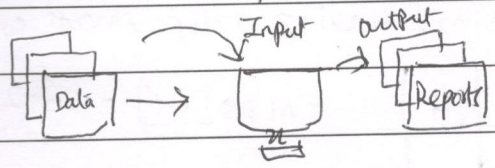
UNIT-I

1) Introduction to DS; Abstract Data type (ADT), Asymptotic notations, Time-Spac tradeoff, Searching: linear and Binary Search techniques and their time complexit
2) Linear DS: STACKS, ADT Stacks and its operations, Applications of stack Recursion, Expression conversion and evaluation.

U-1: Introduction:-

Data → It is simply values or set of values
A Data Item → A single unit of values — are divided into sub items — name (Firstna, Lastna)
2) Address 3) Mobile no (1) (2)
Social Security NO — } Single item
Student Hallticket no }
Collection of data → Fields → Records → Files
Entity has certain attributes/properties which may be assigned values
Entity has a range of values, the set of all possible values
Ex:- Age: {...} Ex: {Male, Female..} ---
Information: → data with given attributes → Processed data.

Each record in a file may contain field names, but the value in a certain field may uniquely determine the record in a file Ex:- P.K
Ex:- 1. Serial NO, Type, Year, Price, Accessories } inventory of automobile
2. ~~Name, Address, Telephone number, Sales record~~
X can't be PK
Add another field like "Membership ID"



For efficient process of data, storage and toget results we need to organize the datas into more complex types of structu

The study of data structures include:
1) Logical / Mathematical ~~structure~~ description of the structure
2) Implementation of the structure on a computer
3) Quantitative analysis of the structure ← Amount of memory needed, Time required to proces
Stored in primary memory — Main concern
Secondary "

Data Structure! - "The physical and logical organization of data to perform a particular task"

This Data structure selection depends on 1) Preserving the relationships 2) Effective Process

---

classification of Data Structure:

- Primitive Data Structures
  - Basic Data types
    1. Integer
    2. Real
    3. Character
    4. Boolean
- Non-Primitive Data Structures
  - Linear Data Structures
    1. Arrays
    2. Linked lists
    3. Stacks
    4. Queues
  - Non-linear Data Structures
    1. Trees
    2. Graphs

fig: classification of Data Structures

↳ These data types consists of characters that cannot be divided and hence they are also called simple data types / Primitive Data Types

To process complex numbers / arrangement of data in different format we use non-primitive data structures.

Linear D.S: elements form a sequence / linear list. They are arranged in linear fashion but it need not to be stored sequential

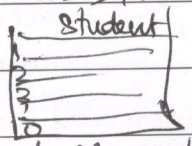
Non-linear DS: Data is not arranged in sequence. The insertion and deletion of data is therefore not possible in a linear fashion.

Arrays:- Simplest data structure. A linear array (1D Array)

Each element in array referenced by its subscript variable.

1) A [| | | |] A[1], A[2], ··· A[4]
   1 2 3 4

2) 10 student names



A chain of 28 stores, each store having 4 departments which stores its weekly sales

SALES	Dept / Store	1	2	3	4
	1	2872			
	2				
	⋮				
	28				

SALES[1,1] = 2872

Linked lists:

7	Keller	Ray
8	Hill	Smith
9	Infield	Ray

	Customer	Salesperson
1	Adams	Smith
2	Brown	Ray
3	Clark	Jones
4	Drew	Ray
5	Evans	Smith
6	Farmer	Jones

	customes	link
1.	Adams	5
2	Brown	4
3	Clark	6
4	Drew	7
5	Evans	8
6	Farmes	0
7	Getes	9
8	Hill	0
9	Infeld	0

	salesperson	Pointer
1	Jones	3
2	Ray	2
3	Smith	1

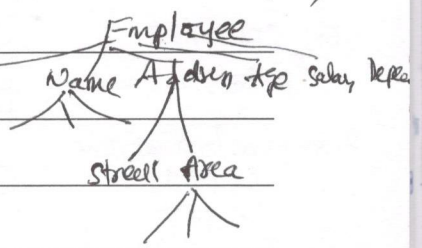
2 customess
4
3
9

Trees: To represent employe data contains social security No, Name, Address, Age, Salary, Department, Dependents

Name
- Lname
- Fname
- Mname

Street
Area
- City
- State
- PIN

social security NO



Data structure operations:-

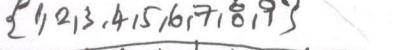
1. Traversing: Accessing data / visiting the record
2. searching: finding the location of a record based on key
3. Inserting
4. Deleting.

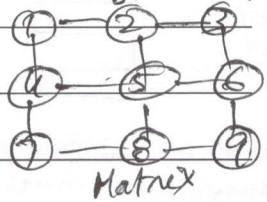
Abstract Data Types (ADT): It is refers to a set of data values and associated operations that are specified accurately, independent of any particular implementation.

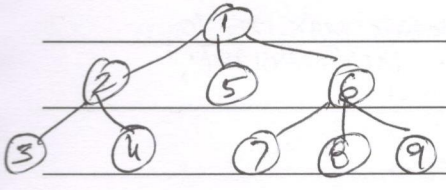
Ex:- Data: {1,2,3,4,5,6,7,8,9}

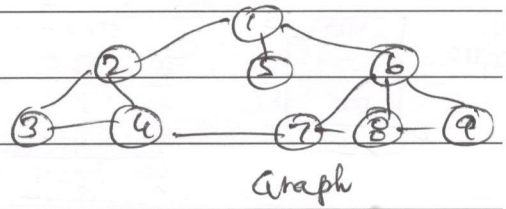
Array: A

1	2	3	4	5	6	7	8	9









An ADT can thus be further defined as a data declaration package together with the operations that are meaningful for the data type. In otherwords we encapsulate the data and the operations on the data and then we hide them from the user

The user need not know the data structure to use the ADT.

The ADT Model:

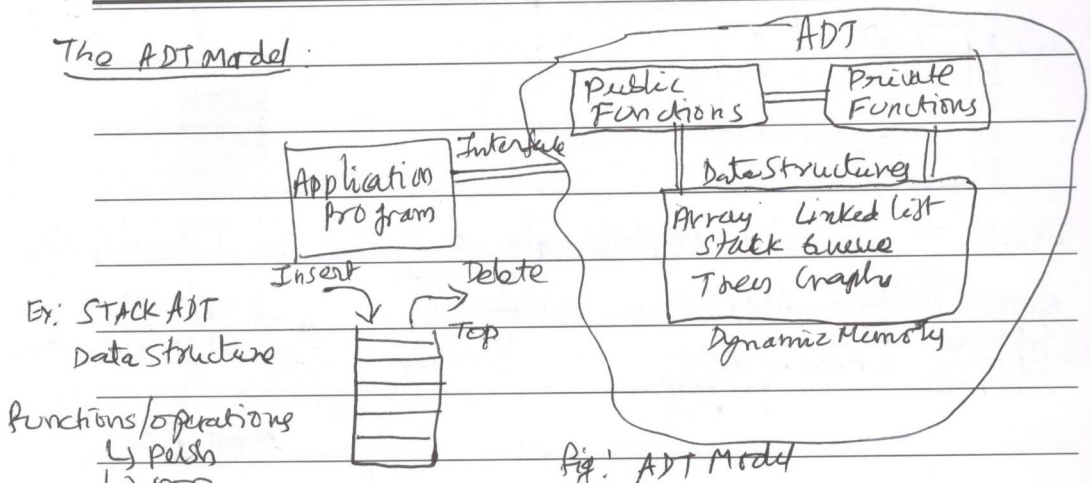


fig: ADT Model

STACK : A stack is a list of elements in which an element may be inserted or deleted only at one end, called the 'top' of the stack.

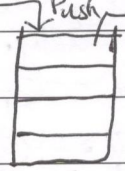
Stack works with "First-in-last-out" method.

Terminology used for two basic operations in stack:

1) 'Push' – to insert an element into a stack
2) 'Pop' – to delete an " from "
3) 'Display' – to display

Stack uses an underlying principle that insertions and deletions can occur at the 'top' of the stack. stack looks like one side open pipe

Ex:-



stack of size 4 (MAX) Here

Initially we set the top of the stack to '-1'

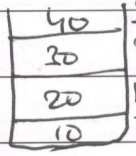
For every insert operation top will be incremented by +1 and decremented by -1 for every pop

Initially set the size of stack through global variable set the size of stack

```
int stack[10];
```

```
#define MAXSIZE 10
int stack[MAXSIZE];
```

1) Push: 10, 20, 30, 40



top = -1

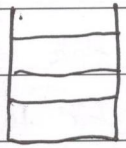
10 top = 0

20 top = 1

30 top = 2

40 top = 3

2) POP



1. pop — deleted element is 40

top = top - 1 ⇒ top = 2

2 pop

top = top - 1 ⇒ top = 1

Conditions in Stack

1. Overflow Condition: During every push operation we must check for overflow condition of the stack. i.e checking the stack whether it is full or not. If it is full we can't insert/push the element. Otherwise we can push/insert the element into stack.

if (top == MAXSIZE-1) then Overflow condition is occured
else
apply push operation.

Algorithm: Push operation
Step1: start
2: [check the overflow condition]
if (top == (MAXSIZE-1))
write "stack is full"
else goto step3
3: [insert an element into stack]
top = top+1
Stack[top] = item;
4: stop

2. Underflow condition:- During every pop operation we must check for underflow condition of the stack. i.e checking the stack whether it is empty or not. If it is 'empty', we can't delete the element. Other wise we can delete/pop the element.

if (top == -1) then underflow condition is occured.
else
apply pop operation.

Algorithm: pop operation
Step1: start
2: [check the underflow condition]
if (top == -1)
write "stack is empty"
else goto step3
3: [Pop the element]
item = stack[top]
write "The deleted item is "item"
top = top-1;
4: stop

3. Algorithm: display operation
Step 1: start
2: [check the emptyness of the stack]
if (top == -1)
write "stack is empty"
else goto step3
3: [Displays the elements of stack]
For i = top to 0 (decrement by 1)
write "stack[i]"
4: stop

```
#include<stdio.h>
#include<conio.h>
int stack[10], choice, n, top, x, i;
void push(void);
void pop(void);
void display(void);
int main()
{
  clrscr();
  top=-1;
  printf("\n Enter the size of stack
          [max=10]: ");
  scanf("%d", &n);
  printf("\n Stack operations using
                          Array");
  printf("\n 1.PUSH \n 2.POP \n 3.display
    \n 4.EXIT");
  do
  {
    printf("\n Enter your choice:");
    scanf("%d", &choice);
    switch(choice)
    {
      case 1: { push();
                break;}
      case 2: { pop();
                break;}
      case 3: { display();
                break;}
      case 4: { printf("\n exit option");
                break;}
      default: { printf("\n Enter valid
        input choice (1/2/3/4)");
               }
    }
  } while (choice!=4);
    return 0;
}
```

```
void push()
{ if(top==n-1)
  { printf("\n Stack is full \n overflow");
  }
  else
  { printf("\n Enter a value to be
                          pushed");
    scanf("%d", &x);
    top++;
    stack[top]=x;
  }
}
void pop()
{ if(top==-1)
  { printf("\n Stack is empty \n underflow");
  }
  else { printf("\n The popped is %d", stack[top]
         top--;
  }
}
void display()
{ if(top==-1)
    printf("\n The stack is empty");
  else
  {
    for(i=top; i>=0; i--)
      printf("\n %d", stack[i]);
    printf("\n Press next choice");
  }
}
```

Applications of Stack:

1. Recursion
2. Keeping the track of interrupts.
3. Conversion of expressions (infix, Postfix, Prefix)
4. Evaluation of Expression
5. Postponed decisions project A → B → C → D
   Project dependency.
6. Quicksort

Arithmatic Expression:

Operator levels of precedence : Highest : ^ (exponentiation)
Next highest : *, /
Lowest : +, -

Ex. - $2 \wedge 3 + 5 * 2 \wedge 2 - 12/6 \Rightarrow 8 + 5 * 4 - 12/6 \Rightarrow 8 + 20 - 2 \Rightarrow 26$

Q: 5 * (6+2) - 12/4

Converting into postfix ⇒ 5, 6, 2, +, *, 12, 4, /, -,

Symbol scanned		STACK
(1)	5	5
(2)	6	5, 6
(3)	2	5, 6, 2
(4)	+	5, 8
(5)	*	40
(6)	12	40, 12
(7)	4	40, 12, 4
(8)	/	40, 3
(9)	-	37
(10)	)	

Recursion: Stack is used to store and restore the recursive function and its arguments. The general idea behind recursion is to divide the problem into smaller pieces until reaching to solvable pieces.

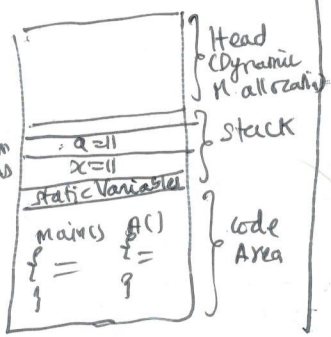
2) Then, to solve the problem by solving these small pieces of problems and merging the solutions to each other

Generally a recursive alg has two parts:

1) checks if the problem is solvable
2) Divides the problem into smaller pieces in each round until it falls under part 1

Program

```
Void A(int a)
{ print ("%d", a);
}
Void Main()
{ int x=10;
  x++;
  A(x);
}
```

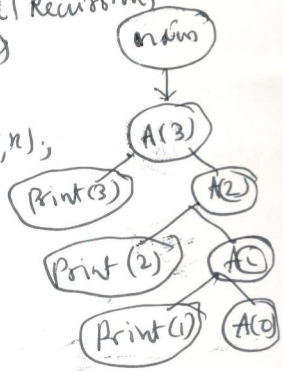


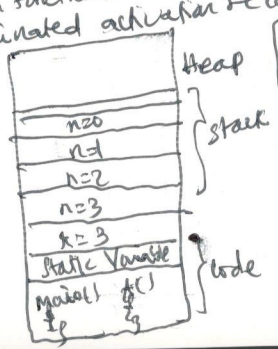
When function is called its activation record is created and when the function is terminated activation record is terminated

Recursion: (Tail Recursion)

```
void A(int n)
{ if(n>0)
  { printf("%d", n);
    A(n-1);
  }
void main()
{ int x=3;
  A(x);
}
```

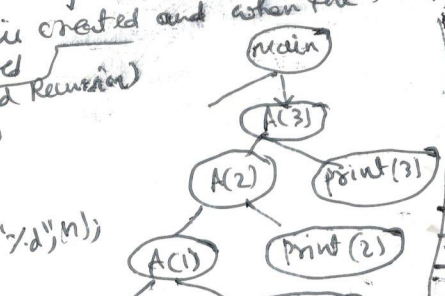
Output: 3 2 1



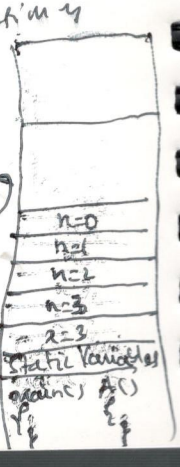


Example-2 (Head Recursion)

```
Void A(int n)
{ if(n>0)
  { A(n-1);
    printf("%d", n);
  }
}
void main()
{ int x=3;
  A(x);
}
```



Output: 1 2 3



Infix, Postfix and Prefix Expressions

Infix expression: - The expression in which operator comes in between the operands. Ex: A+B-C This is an ordinary expression

Postfix Expression: - The expression in which operator comes after the operands Ex:- AB+

Prefix Expression: - The expression in which the operator comes before the operands. Ex:- +AB

Example:- A+B*C-D

A+x-D	x = B*C ⇒	postfix form is x = BC*
y-D	y = A+x ⇒	" " y = Ax+
z	z = y-D ⇒	z = yD-

z is the expression now

Apply z values in postfix form

= yD-

= Ax+D-

= ABC*+D- This is the postfix expression to the give infix expression

Transform infix expression into postfix expression

Algorithm InfixtoPostfix(Q, P)

Suppose Q is an arithmatic expression written in infix notation. This alg finds the equivalent postfix expression P.

Step 1: Push "(" onto stack, and add ")" to the end of Q.

2. Scan Q from left to right and repeat step 3 to 6 for each element of Q until the stack is empty.
3. If an operand is encountered, add it to P
4. If a left parenthesis is encountered, Push it onto Stack
5. If an operator ⊗ is encountered then:
   (a) Repeatedly pop from stack and add to P each operator (on the top of stack) which has the same precedence as or higher precedence than ⊗
   (b) Add ⊗ to stack.
   [End of If structure]
6. If a right parenthesis is encountered then;
   (a) Repeatedly pop from stack and add to P each operator (on the top of stack) until a left parenthesis is encountered
   (b) Remove the left parenthesis. [Do not add the left parenthesis to P
   [end of If structure]

[End of step 2 loop]

7 2x4

Example: a : A+(B*C-(D/E↑F)*G)*H

	Symbol Scanned	Stack	Expression P
1	A	(	A
2	+	(+	A
3	(	(+(	A
4	B	(+(	AB
5	*	(+(*	AB
6	C	(+(*	ABC
7	-	(+(-	ABC*
8	(	(+(-(	ABC*
9	D	(+(-(	ABC*D
10	/	(+(-(/	ABC*D
11	E	(+(-(/	ABC*DE
12	↑	(+(-(/↑	ABC*DE
13	F	(+(-(/↑	ABC*DEF
14	)	(+(-	ABC*DEF↑/
15	*	(+(-*	ABC*DEF↑/
16	G	(+(-*	ABC*DEF↑/G
17	)	(+	ABC*DEF↑/G*-
18	*	(+*	ABC*DEF↑/G*-
19	H	(+*	ABC*DEF↑/G*-H
20	)		ABC*DEF↑/G*-H*+

(A+B*C)+E/F-(G*H)

ANS: ABC*+EF/+GH*-

Example:2 Q: A+B*C/E-F Ans: A+B*C-D

1	A	(	A
2	+	(+	A
3	B	(+	AB
4	*	(+*	AB
5	C	(+*	ABC
6	/	(+/	ABC*
7	E	(+/	ABC*E
8	-	(-	ABC*E/+
9	F	(-	ABC*E/+F
10	)		ABC*E/+F-

1	A	(	A
2	+	(+	A
3	B	(+	AB
4	*	(+*	AB
5	C	(+*	ABC
6	-	(-	ABC*+
7	D	(-	ABC*+D
8	)	Popall	ABC*+D-

Example-3 Q= A*(B+C)*D

1	A	(	A
2	*	(*	A
3	(	(*(	A
4	B	(*(	AB
5	+	(*(+	AB
6	C	(*(+	ABC
7	)	(*	ABC+
8	*	(*	ABC+*
9	D	(*	ABC+*D
10	)		ABC+*D*

C function to convert infix expression into Postfix expression using stack

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
int index=0, pos=0, top=-1, len;
char symbol, temp;
char p[50], q[50], stack[50];
void main()
{
  clrscr();
  Printf("\nEnter Infix Expression");
  scanf("%s", p);
  infix_to_postfix(p,q);
  printf("\nThe infix is = %s", p);
  printf("\nThe Postfix is = %s", q);
  getch();
}
/* function to convert infix to postfix
infix_to_postfix(char p[], char q[])
{
  len = strlen(in);
  while(index<len)
  {
   symbol = p[index];
   switch(symbol)
   {
    case '(': push(symbol); break;
    case ')': temp = pop();
              while(temp != '(')
              {
               q
               post[pos] = temp;
               pos++;
               temp = pop();
              }
              break;
    case '+':
    case '-':
    case '*':
    case '/':
    case '^': while(Precedence
    (stack[top]) >= Precedence(symbol))
    {
     temp = pop();
     post q[pos] = temp;
     pos++;
    }
    push(symbol);
    break;
    default: q[pos++] = symbol;
   }
   index++;
  }
  while(top >= 0)
  {
   temp = pop();
   post q[pos++] = temp;
  }
  return;
}
/* Function for Push operation
void push(char symbol)
{
  top++;
  stack[top] = symbol;
}
/* Function for Pop operation
Char pop()
{ char ch;
  ch = stack[top];
  top--;
  return(ch);
}
/* Function for Precedence operation
int precedence(char symbol)
{
  int r;
  Switch(symbol)
  {
   case '^': r=3; break;
   case '*':
   case '/': r=2; break;
   case '+':
   case '-': r=1; break;
   case ')':
   case '(': r=0; break;
  }
  return(r);
}
```

main() { read(P) call infix_to_postfix(P,Q) print Q }

infix_to_postfix(P,Q) { Based on length of P { check(Symbol) '(': Push(Symbol) ')': POP() } }

Arrays: input: P, stack, output: Q

repeat until = '('

+, -, *, /, ^

Check Precedence of (stack[top], symbol) >=

{ pop() } else push()

default: send to Q

Evaluation of a Postfix Expression pgno: 212

Suppose P is an arithmetic expression written in postfix notation. The following algorithm which uses a stack to hold operands, evaluates P.

Algorithm: This algorithm finds the value of an arithmetic expression P written in postfix notation

Step 1: Add a right parenthesis ")" at the end of P [This acts as a sentinel] ; guarding point

2. Scan P from left to right and repeat step 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, push it to stack.
4. If an operator ⊗ (⊗) is encountered, then:
   (a) Remove the two top elements of stack, where A is the top element and B is the next-to-top element.
   (b) Evaluate B⊗A
   (c) Place the result of (b) back on stack

   [End of If structure
[End of step-2 loop]
5. Set value equal to the top element on stack
6. Exit

Example:-

P = 5 2 3 + *) (infix expression)

SNO	Symbol Scanned	A	B	result B⊗A	stack
1	5	—	—	—	5
2	2	—	—	—	5 2
3	3	—	—	—	5 2 3
4	+	3	2	5	5 5
5	*	5	5	25	25
6	)	—	—	—	—

Example: 2 P: 5 6 2 + * 12 4 / − result: 37

Example: 3 P: 6 3 7 + * 9 + result: 69

// Program to evaluate postfix expression

```
#include<stdio.h>
#include<conio.h>
#include<string.h> <math.h>
float eval_post(char [], float []);
float Push(float);
float pop(); stack[50];
int top=-1;
void main()
{
  int i=0;
  char post[50];
  float value[50], result;
  clrscr();
  printf("\n Enter a valid postfix exp");
  scanf("%s", post);
  while(post[i]!='\0')
  {
    if(isalpha(post[i]))
    { printf("\n Enter the value of %c", post[i]);
      scanf("%f", &value[i]);
    }
    i++;
  }
  result=eval_post(post, value);
  printf("\n The result of %s=%f",
          post, result);
  getch();
}
```

```
float eval_post(char post[], float data[])
{
int i=0;
float op1, op2, result;
char ch;
while(post[i] != '\0')
{
ch = post[i];
if(isalpha(post[i]))
    push(data[i]);
else
  { op2 = pop();
    op1 = pop();
    switch(ch)
    {
      case '+': Push(op1+op2); break;
      case '-': push(op1+op2); break;
      case '*': Push(op1*op2); break;
      case '/': push(op1/op2); break;
      case '^': Push(op1^op2); break;
    }
  } i++;
}
res = pop();
return(res);
}
```

```
// function for Push operation
float push(float num)
{
  top++;
  stack[top] = num;
  return stack[top];
}

// function for POP operation
float pop()
{
  float num;
  num = stack[top];
  top--;
  return(num);
}
```

Read Postfix exp -> Post-string
if isalpha(Post[i])
then read a value
result = eval post(Post, value) -> float

```
eval_Post
{ until(Post[i] != '\0')
{ if(isalpha) — Push->data[i]
  else op2=pop(); op1 = pop();
switch(ch){+,-,*,/,^}
  Push(op1 + op2); break;
  result = pop();
```

```
float push(float num)
{ top++; stack[top] = num;
  return stack[top];
}
float pop()
{ num = stack[top];
  top--;
  return(num);
}
```

Ex:- 78+3*635-1-

Recursion: Two properties of Recursive procedure.
1. There must be certain criteria, called base criteria, for which the procedure does not call itself.
2. Each time the procedure does call itself, it must be closer to the base criteria.

Two properties of Recursive function
1. There must be certain arguments, called base values, for which the function does not refer to itself
2. Each time the function does refer to itself, the argument of the function must be closer to a base value

Ex:- Fibonacci Sequence

Procedure: Fibonacci(Fib, n)
This procedure calculates F_N and returns the value in the first parameter FIB.

1. If n=0 or n=1 then set Fib := N and return
2. Call fibonacci(FIBB, N-2)
3. Call fibonacci(FIBB, n-1)
4. Set FIB := FIBB + PIBB
5. Return

Divide-and-Conquer Algorithm

1. Binary Search

2. Ackermann Function: It is a discovered example of total computable function

Definition:- Let m, n are any non-negative integers $\{0, 1, 2, \cdots\} \in \mathbb{N}_0$

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m \neq 0 \text{ but } n=0 \\ A(m-1, A(m,n-1)) & \text{if } m \neq 0 \text{ and } n \neq 0 \end{cases}$$

$A(0,0) = 1$
$A(0,5) = 6$

$$\begin{aligned} A(1,2) &= A(0, A(1,1)) \\ &= A(0, A(0, A(1,0)) \\ &= A(0, A(0, A(0,1)) \\ &= A(0, A(0,2)) \\ &= A(0,3) \\ &= 4 \end{aligned}$$

$A(4,1) = 65533$

$A(4,2)$ results an integer of 19,729 decimal digits almost $2^{65536} - 3$

$= 2^{2^{65536}} - 3$

```
Ackermann(int m, int n)
{ if(m==0) return n+1;
  if(n==0) return ackermann(m-1, 1);
  return ackermann(m-1, ackermann(m, n-1));
}
```

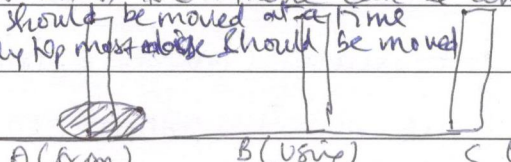
End of the world puzzle
Tower of Bhrahama
64 golden disks
1 move in one second
580 billion years

Towers of Hanoi: invented by French Mathematician Lucas in 1883.
youtu.be/q6RicKIFOWs

There are 3 towers given. and one of the tower is having stack of disks in the decreasing order of disks from the bottom towards top. We have to move all these disks from tower A to C At any moment of time a larger disk should not kept on smaller For help you were given an auxiliary tower called B. It can be used to transfer all disks from A to C. There can be any number of disks in A. Only one disc should be moved at a time

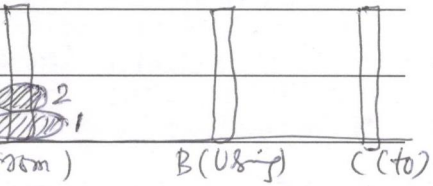
Solution:- Single disc: only top most disc should be moved

Sol: Move a disc from A to C



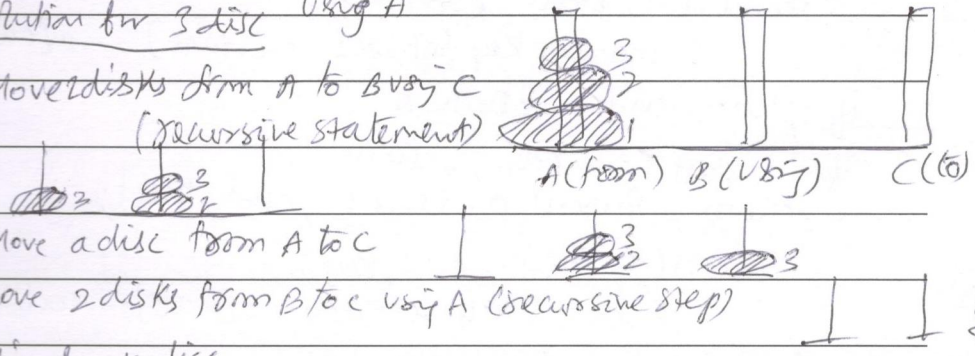
(2) Two disks are given

Sol:- (1) Move a disc from A to B using C
(2) Move a disc from A to C
(3) Move a disc from B to C



(3) Solution for 3 disc using A

1) Move disks from A to B using C (recursive statement)
2) Move a disc from A to C
3) Move 2 disks from B to C using A (recursive step)



Solution for n disc
1) Move n-1 discs from A to B using C
2) Move a disc from A to C
3) Move n-1 discs from B to C using A

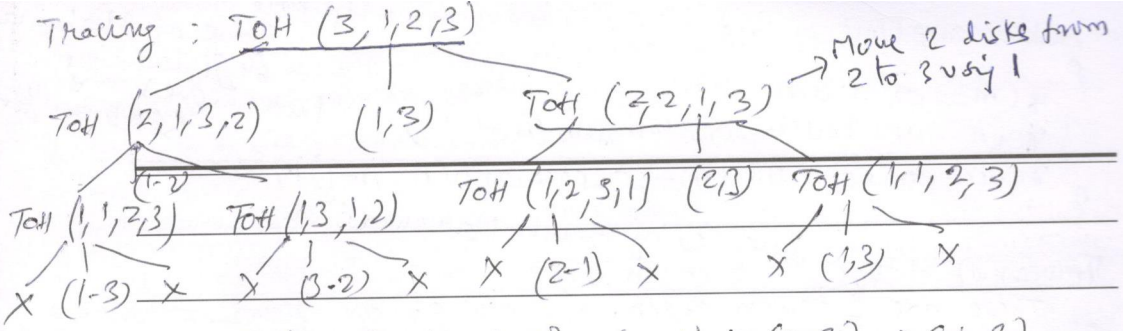
(can be applied for any no of disks.

n disks
A, B, C - Towers

```
void TOH(int n, int A, int B, int C)
{ if(n>0)
  { TOH(n-1, A, C, B);
    printf("Move a disc from %d to %d", A, C);
    TOH(n-1, B, A, C);
  }
}
```

The minimum no of moves necessary to complete the puzzle is $2^n - 1$

Tracing : TOH (3, 1, 2, 3)



(1-3) → (1-2) → (3-2) → (1-3) → (2-1) → (2-3) → (1-3)

= 7 steps $2^n - 1 = 2^3 - 1 = 7$ steps

Linear search: (Recursive approach)

Ex:-1

```
int search (int a[], int n, int Key)
{  if (n==0)  return 0;
   if (a[n-1]==key) return 1;
      return search(a, n-1, key);
}
```

Ex:-2

```
search (a, start, end, key)
      // search key between a[start] .. a[end]
if start > end, return 0;
if a[start] == key, return 1;
return search(a, start+1, end, key);
```

Ex:-3

```
search(a, start, end, key)
      // search key between a[start] ... a[end]
if start > end, return 0;
mid = (start+end)/2;
if a[mid] == key, return 1;
return search(a, start, mid-1, key)
      || search(a, mid+1, end, key)
```

Profiling: In s/w eng, Profiling is a form of dynamic program analysis that measures. Ex:- The space or time complexity of a program, The usage of particular instructions, or the frequency and duration of function calls. Most commonly, Profiling information serves to aid program optimization

Dictionary Meaning: The recording and analysis of a persons psychological and behavioural characteristics so as to assess or predict their capabilities.

Asymptotic Notations, Time, Space tradeoffs

An algorithm is a well-defined list of steps for solving a particular problem. The main objective is to develop efficient algorithms for processing our data. The time and space it uses are two major measures of the efficiency of an algorithm. The complexity of an algorithm is the function which gives the running time and space in terms of the input size.

Sometimes the choice of data structures involves a time-space tradeoff: By increasing the amount of space for storing the data, one may be able to reduce the time needed for processing the data or vice versa.

Example:- Searching Algorithms:
Suppose we are implementing an algorithm that searches for a item in the list. Here we have different possibilities:

1. Best Case: The item that we search is found in the first location itself. In this case the algorithm complexity $C(n) = 1$
2. Worst Case: The item that we wants search is found at the last location $\therefore C(n) = n$
3. Average Case:- The item that we want to search can be present at any location in the list. In this case we do not know at which position it might be. Hence we take an average of all possibilities. Hence assuming for n data, we have a probability of finding any one of them is "$\frac{1}{n}$"

$$\therefore C(n) = 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + 3 \cdot \frac{1}{n} + \dots + n \cdot \frac{1}{n}$$

$$= \frac{1}{n}[1 + 2 + \dots + n]$$

$$= \frac{1}{n}\left[\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

or $f(n) = \left(\frac{n+1}{2}\right)$

$n \rightarrow$ input size

The time and space it uses are two major measures of the efficiency of an algorithm.

1. Space complexity:- It is also known as memory requirement to complete the task. However in high end systems (Servers / latest PCs) more resources are usually allocated for the operation in order to reduce the time taken.
2. Time Complexity: It is also known as performance requirement. Choice of good data structures has greater impact in the time complexity. Efficient DS will definitely reduce the time of running a program.

NOTE: The complexity of the average case of an algorithm is usually much more complicated to analyze than that of the worst case. Moreover, the probabilistic distributions that one assumes for the average case may not actually apply to real situations.

How to balance? The best algorithm to solve a given problem is one that requires less space in memory and takes less time to complete its execution. But in practice it is not always possible to achieve both these objectives.

Approach-1: Take more space but take less time to complete its execution. [Preferred now a days as due to high-end systems]

Approach-2: Take less space but takes more time to complete its execution

We have to sacrifice one at the cost of the other.

Notations used to represent the complexities of algorithms:-

Suppose 'A' is an algorithm and 'n' is the size of input data. Then the complexity of algorithm is $f(n)$ of A increases as 'n' increases.

Let us examine $f(n)$ with standard functions such as $g(n)$ $\log_2 n$, $n\log_2 n$, n^2, n^3, 2^n

n / g(n)	log n	n	n log n	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	10^3	10^4	10^6	10^9	10^{300}

The rate of growth in logarithmic function grows most slowly where as in exponential function grows most rapidly.

Suppose $f(n)$ and $g(n)$ are two functions defined on positive integers there exist a positive integer n_0 and a +ve number M such that for all $n > n_0$, we have

$$|f(n)| \leq M|g(n)|$$

This can be written as $f(n) = O(g(n))$ read as "f(n) is of order g(n)"

Ex:- A polynomial $P(n)$ of degree m

$P(n) = 8n^3 - 576n^2 + 832n - 248$ has $O(n^3)$

The time complexities of popular methods are defined as

(a) Linear Search : $O(n)$
(b) Binary Search : $O(\log n)$
(c) Bubble sort : $O(n^2)$
(d) Merge Sort : $O(n \log n)$

Logarithm says that: A quantity representing the power to which a fixed number (the base) must be raised to produce a given number.

Other notations :-

"Big O" defines the upper bound function g(n) for f(n)

2. Omega Notation (Ω) : It is used when the function g(n) defines a lower bound for the function f(n)

Definition: $f(n) = \Omega(g(n))$ iff there exists a positive integer n_0 and a positive number M such that $|f(n)| \geq M.|g(n)|$ for all $n \geq n_0$

Ex:- $f(n) = 18n+9$, $f(n) > 18n$ for all n
Hence $f(n) = \Omega(n)$

$f(n) = 90n^2+18n+6$, $f(n) > 90n^2$ for $n > 0$
Hence $f(n) = \Omega(n^2)$

3. Theta Notation (Θ) :- Used when the function f(n) is bounded both lower and upper bound function of g(n)

Definition :- $f(n) = \Theta(g(n))$ iff there exist two +ve constants c_1 and c_2 and a +ve integer n_0 such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ $\forall$ $n \geq n_0$

Ex:- $f(n) = 18n+9$
$\therefore f(n) > 18n$ and $f(n) \leq 27n$ $\forall$ $n \geq 1$
$\therefore f(n) = \Omega(n)$ and $f(n) = O(n)$
Hence $f(n) = \Theta(n)$

4 Little Oh Notation (o)

Definition :- $f(n) = o(g(n))$ iff $f(n) = O(g(n))$ and $f(n) \neq \Omega g(n)$

Ex:- $f(n) = 18n+9 \Rightarrow f(n) = O(n^2)$
but $f(n) \neq \Omega(n^2)$
Hence $f(n) = o(n^2)$ However $f(n) \neq o(n)$

Ex:- Selection sort : $O(n^2)$

```
for(x=0; x<n; x++)
{
  min=x;
  for(y=x; y<n; y++)
  {
    if(array[y]<array[min])
      min=y;
  }
  temp=array[x];
  array[x]=array[min];
  array[min]=temp;
}
```

outer loop O(n)
Inner loop roughly O(n) since it is based on the value of x, average n/2 and we ignore the constant. $\therefore O(n)$
$O(n) \cdot O(n) \Rightarrow O(n^2)$

Binary search : $O(\log n)$

Example : 16 array elements (Search element = 13)
1, 3, 5, 8, 12, 13, 15, 16, 18, 20, 22, 30, 40, 50, 55, 67
Selecting middle element as pivot = (length/2) = $16 \times \frac{1}{2} = 8$
$\because$ 13 < pivot (First Half)

4 times Division to get target element
$\therefore 16 \times (\frac{1}{2})^4 = 1$

for n elements : $n * (\frac{1}{2})^k = 1$

$n * \frac{1}{2^k} = 1$

$n = 2^k$

$\log_2 n = k$

$8 \times \frac{1}{2} = 4 \Rightarrow 4 \times \frac{1}{2} = 2 \Rightarrow 2 \times \frac{1}{2} = 1$