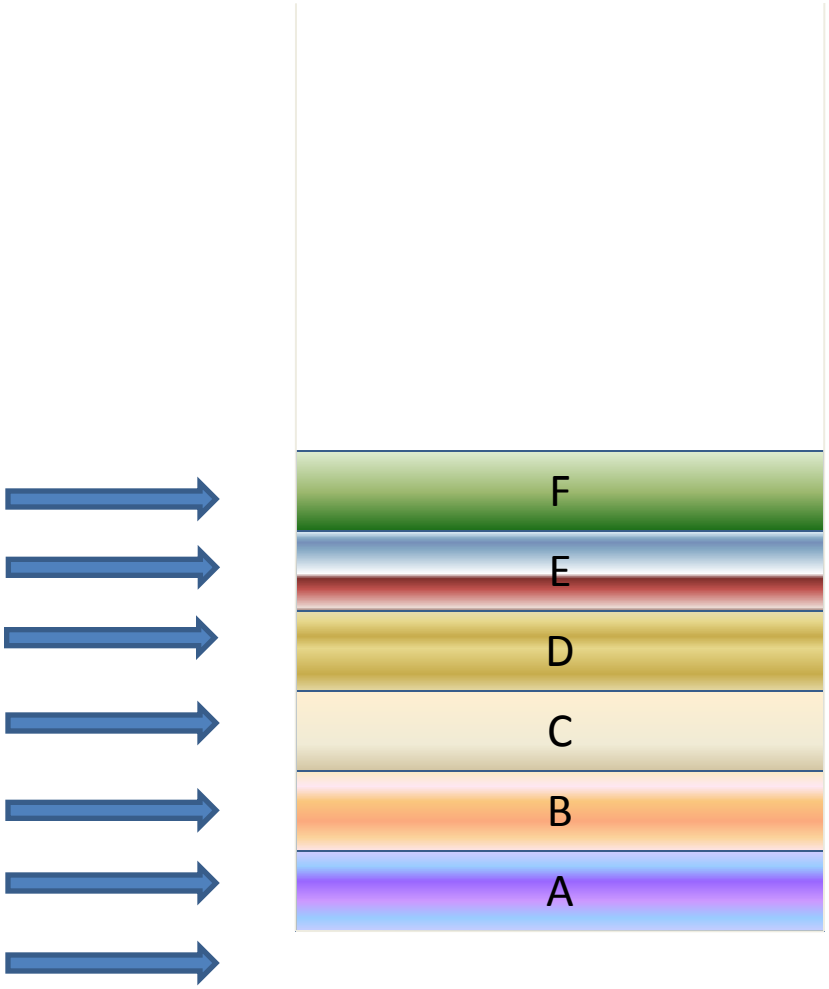
STACKS

Unit 2

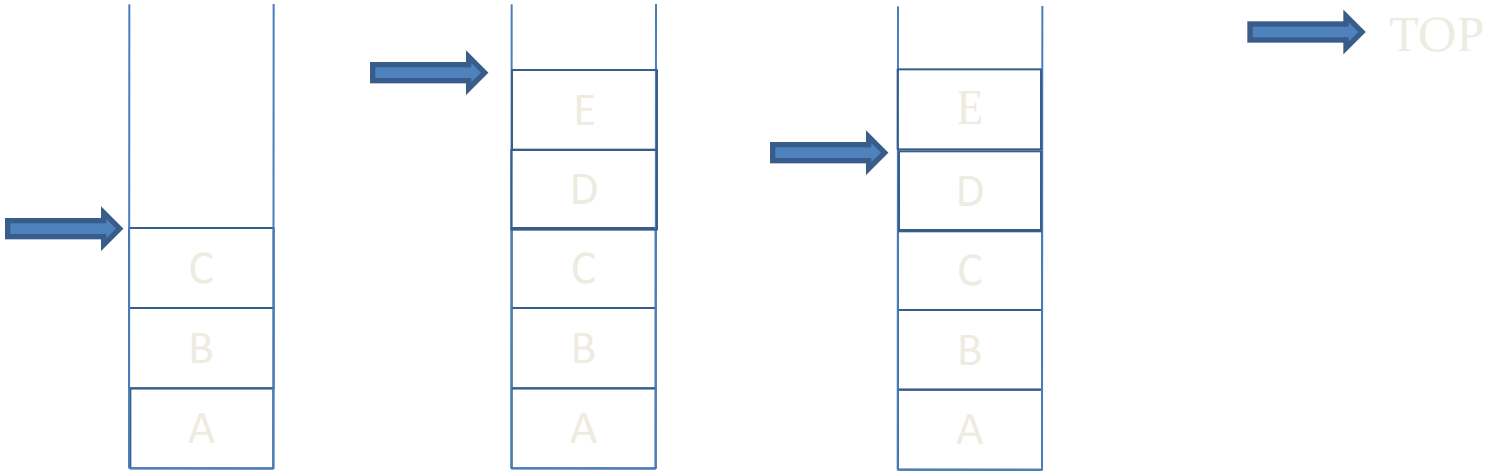
DEFINITION AND EXAMPLES

- A **stack is an ordered collection of items** into which new items may be inserted and from which items may be deleted at one end, called the ***top*** of the stack.
- Unlike that of the array, the definition of the stack provides for the insertion and deletion of items, so that a **stack is dynamic, constantly changing object.**
- A single end of the stack is designated as the **stack top**.
- New items may be put on top of the stack or items which are at the top of the stack may be removed.
- A stack is sometimes called a **last-in, first-out(or LIFO) list.**

Stacks



Stacks



PRIMITIVE OPERATIONS

- The two changes which can be made to a stack are given special names.
- When an item is added to a stack, it is ***pushed*** onto the stack, and when an item is removed is ***popped*** from the stack.
- Given a stack s, and an item i, performing the operation ***push***(s,i) adds the item i to top of stack s.
- Similarly, the operation pop(s) removes the top element and returns it as a function value.
- Thus the assignment operation i = pop(s);

removes the element at the top of s and assign its value to i.

- Because of the push operation which adds elements to a stack, a stack is some times called a ***pushdown*** list.
- **There is no upper limit on the number of items t**hat may be kept in a stack, since the definition does not specify how many items are allowed in the collection.
- If a stack contains a single item and the stack is popped, the resulting stack contains no items and is called the ***empty stack***.

- Although the ***push*** operation is applicable to any stack, the **pop operation cannot be applied to the empty stack**.
- Therefore, before applying the pop operator to a stack, we must ensure that the stack is not empty.
- The operation **empty(s)** determines whether or not stack s is empty.
- if the stack is empty, empty(s) returns the value ***TRUE,*** otherwise returns the value ***FALSE.***

- Another operation that can be performed on a stack is to determine what the ***top*** item on a stack is without removing it.
- This operation is written ***stacktop*** (s) and returns the top element of the stack s.
- it can be decomposed into a pop and a push.

```
i = stacktop(s);
```

is equivalent to

```
i = pop(s); push(s,i);
```

- Like the operation pop, stacktop is not defined for an empty stack.
- The result of an illegal attempt to pop or access an item from an empty stack is called ***underflow.***
- ***Underflow*** can be avoided by ensuring that empty(s) is false before attempting the operation pop(s) or stacktop(s).

Example: **Usage of Stacks in problem solving**

Consider a mathematical Expression containing several sets of nested parenthesis.

We need to ensure that parenthesis are nested correctly.

1. There must be an equal number of right and left parentheses.
2. Every right parenthesis is preceded by a matching left parenthesis.

How to solve it?

Use a **counter** to count the no of parenthesis.

Increment counter when you encounter a left parenthesis and decrement it when you encounter a right parenthesis.

Two conditions that must hold to prove that it forms a valid pattern is as follows:

1. **The parenthesis count at the end of the expression is 0.** It ensures that exactly as many right parentheses as left parenthesis have been found.
2. **The parenthesis count at each point in the expression is nonnegative.** This implies that no right parenthesis is encountered for which a matching left parenthesis had not previously been encountered.

Stacks can be used to solve such problems in an easy manner.

The stack as an Abstract Data Type

- The representation of stack as an abstract data type is straightforward.
- We use ***eltype*** to denote the type of stack element and parameterize the stack type with ***eltype.***

abstract typedef <<eltype>> STACK (eltype);

```
 abstract empty(s)
STACK(eltype) s;
postcondition empty == (len(s) == 0);

abstract eltype pop(s)
STACK(eltype) s;
precondition empty(s) == FALSE;
postcondition pop == first(s');
                          s == sub(s'  1,  len(s')  –  1);
abstract push(s, elt)
STACK (eltype) s;
eltype elt;
postcondition    s == < elt > + s';
```

REPRESENTING STACKS IN C

- A **stack** is an ordered collection of items, and C already contains a data type that is an ordered collection of items: **the array.**
- However, a stack and an array are two entirely different things. **The number of elements in an array is fixed** and is assigned by the declaration for the array. In general the user cannot change this number.
- **A stack,** on the other hand, is fundamentally a **dynamic object** whose size is constantly changing as items are popped and pushed.

- **An array can be declared large enough for the maximum size of the stack.**
- **Two Methods** can be used.

First method: using One dimensional array

Example:

Stack[max_stack_size];

where ***max_stack_size*** is the maximum number of entries.

The first element of the stack is stored in ***stack[0]***, the second in ***stack[1]*** and the i^{th} element in ***stack[i-1]***.

The variable ***top*** points to the top element of the stack.

- Initially ***top=-1*** to indicate an ***empty*** stack.
- The operations on a stack are
  - ***Push*** – ***inserting*** an element into the stack.
  - ***Pop***– ***deleting*** an element from the stack.
- Each time an element is entered in to the stack ***top*** is ***incremented***.
- The elements can be entered into the stack until ***top >= max_stack_size - 1***.
- Beyond this it becomes the ***STACK OVERFLOW*** condition.

- Each time an element is deleted from the stack ***top*** is ***decremented***.
- The elements can be deleted from the stack until ***top = - 1***.
- Beyond this condition it becomes the ***STACK UNDERFLOW*** condition.

Second Method: using structures

A stack in C may be declared as a **structure containing two objects an array** to hold the elements of the stack, and an **integer** to indicate the position if the **current stack top** within the array.

```
#define STACKSIZE 100
struct stack {
int top;
int items[STACKSIZE];
};
```

Once this has been done, an actual stack s may be declared by

```
struct stack s;
```

- The empty stack contains no elements and can therefore be indicated by any equaling -1.
- To initialize a stack ***s*** to empty state. We mainly initially execute ***s.top = -1;***

```
if (s.top == -1)
 /*stack is empty*/
else
/*stack is not empty*/
```

```
int empty(struct stack *ps)
{
  if (ps -> top == -1)
 return(TRUE);
else
return(FALSE);
/*end empty*/
```

Alternatively:

return (ps-> top == -1);

This statement is precisely equivalent to the longer statement.

```
If (empty(&s))
/*stack is empty*/
else
/*stack is not empty*/
```

This style increases the readability

Implementing the pop operation

```
int pop(struct stack *ps)
{
  if (empty(ps)) {
        printf("%", "stack underflow");
        exit(1);
  }/*end if */
return(ps -> items[ps -> top --]);
 }/* end pop*/
```

Alternatively:

```
temp = ps->items[ps->top]
ps->top-- ;
return temp;
```

We can call the function as : X = pop (&s);

```
if(! empty(&s))
X = pop (&s);
else
/*take remedial action*/
```

Implementing the push operation

```
void push (struct stack *ps, int x)
{
  ps-> items[++(ps->top)] = x;
  return;
} /*end push*/
```

But if we use arrays to implement stacks then we have to test for overflow condition.

```
void push (struct stack *ps, int x)
{
  if (ps -> top == STACKSIZE -1) {
          printf(“%s”, “stack overflow”);
          exit(1);
   }
 else
ps->items [++(ps->top)] = x;
return;
} /*end push*/
```

Look into top element of stack without removing it

```
int stacktop(struct stack *ps)
{
  if(empty(ps)) {
        printf("%s", "stack underflow");
        exit(1);
  }
else
return(ps-> items[ps->top]);
} /* end stacktop*/
```

Equivalent to:

```
x = pop(ps)
Push(ps,x)
```

Infix, Prefix and Postfix Expressions

- It is the major application of stacks
- A + B - infix
- + A B - prefix (also called as ***Polish Notation)***
- A B + - postfix ***(*** also called as ***reverse polish notation***)
- The prefixes in, pre, post refer to the relative position of the operator with respect to the two operands
- examples

Infix, Prefix and Postfix Expressions

- Precedence:

  Parenthesis (if any)

  Exponentiation

  Multiplication/division

  Addition/subtraction

- When we have the operators of the same precedence scanned without the parenthesis, the order is assumed **left to right.**
- In the case of **exponentiation,** the order is assumed to be **right** to **left.**

Applications of stacks

- Evaluating a postfix expression
  - The assumptions are
    - The input should be a valid postfix expression.
    - The expression may contain single non-negative digits and binary operators.
    - No blank space is allowed in between the expression.
    - The allowed operators are +, -, *, /, ^.
    - The allowed digits are 0,1,2,3,4,5,6,7,8,9

- ***P*** is the expression written in the postfix notation and stored in a stack.
  1. Add '***#***' at the end of ***P***.
  2. ***Repeat*** the scanning process of ***P*** from left to right and step ***3*** and ***4*** until '***#***' is encountered.
  3. If an operand is encountered, push it into the stack.

4. If an operator **@** is encountered:
   a. Remove the top two elements of the Stack. ***A*** is the top element and ***B*** is the next element.
   b. Evaluate ***B @ A***.
   c. Push the result of previous step back into the stack.
5. Evaluated value is at the top of the stack.
6. Stop.

Program to evaluate a Postfix Expression

```
#include<math.h>
#define MAXCOLS 80
#define TRUE 1
#define FALSE 0
double eval(char[]);
double pop(struct stack *);
void push(struct stack *, double);
int empty(struct stack *);
int isdigit(char);
double oper(int, double, double);
```

```
struct stack  {
int top;
double items[MAXCOLS];
}
```

main() function

```
int main(void)
{
  char expr[MAXCOLS];
 int position = 0;
while((expr [position++] = getchar()) ! =’\n’)
;
expr[-- position] = ‘\0’;
printf(“%s%s”, “the original postfix expression is”, expr);
printf(“\n%f”, eval(expr));}
return 0;
}    /*end main*/
```

```
double eval(char expr[])
{
  int c, position;
 double opnd1, opnd2, value;
 struct stack opndstk;
opndstk.top = -1;
 for(position = 0; (c = expr[position]) != '\0'; position++)
      if(isdigit(c))
/*operand convert the character representation of the digit
into double and push it onto the stack*/
        push(&opndstk, (double) (c –'0'));
else  {
/* operator */
     opnd2 = pop(&opndstk);
     opnd1 = pop(&opndstk);
     value = oper(c, opnd1, opnd2);
     push  (&opndstk, value);
} /* end else*/
return (pop(&opndstk));
}/* end eval */
```

```
int isdigit(char symb)
{
  return(symb >= ‘0’ && symb <= ‘9’);
}
 double oper(int symb, double op1, double op2)
{switch(symb)  {
    case ‘+’ : return (op1 + op2);
    case ‘-’ : return (op1 - op2);
    case ‘*’ : return (op1 * op2);
    case ‘/’ : return (op1 / op2);
    case ‘$’ : return (pow(op1, op2));
    default : printf(“%s “, “illegal operation”);
                              exit (1);
}/* end switch */
}/* end oper */
```

Conversion from Infix to Postfix

- The assumptions are
  - The given infix expression should be valid.
  - The expression should only contain single character operands i.e. ***a, b,…..z*** and ***0,1,2….9***
  - The valid operators are ***+***, ***-***, *******, ***/***, ***^***
  - The expression may or may not contain parenthesis.

Conversion from Infix to Postfix

- Let ***Q*** be the arithmetic expression written in infix notation. ***P*** is the equivalent postfix expression.
  1. Push the left parenthesis ***(*** on to the stack and add the ***)*** at the end of ***Q***.
  2. Scan ***Q*** from left to right and repeat steps 3 to 6 until the stack is empty.
  3. If an operand is encountered add it to ***P***
  4. If ***(*** is encountered push it into the stack and can be popped only when the ***)*** is encountered.

Conversion from Infix to Postfix

5. If an operator ***@*** is encountered
   a. Repeatedly pop each operator, which has the same precedence or higher precedence than ***@*** from the stack and add it to ***P***.
   b. Push ***@*** to the stack.
6. If ***)*** is encountered
   a. Repeatedly pop each operator from the stack and add it to ***P*** until the ***(***is encountered.
   b. Remove ***(***.
7. Exit.

Conversion from Infix to Postfix

```
opstk = the empty stack;
while (not end of input) {
      symb = next input character;
      if (symb is an operand)
           add symb to the postfix string
     else {
       while(!empty(opstk) && prcd(stackop(opstk), symb)) {
         topsymb = pop(opstk);
         add topsymb to the postfix string;
       } /* end while */
       push(opstk, symb)
   }/* end else */
 } /* end while */
/* output any remaining operators */
While (!empty(opstk)) {
     topsymb = pop(opstk);
    add topsymb to the postfix string;
}/* end while*/
```

Examples

- Ex 1: ***Q = A + B * C + (D * E + F) * G***
  - Push the left parenthesis ***(*** on to the stack and add the ***)*** at the end of ***Q***.
  - ***Q = A + B * C + (D * E + F) * G)***

Conversion from infix to postfix- Example

Scan No	***Symbol Scanned***	***Stack***	***P***
		(	
1.	A	(	A
2.	+	(+	A
3.	B	(+	AB
4.	*	(+*	AB
5.	C	(+*	ABC

Conversion from infix to postfix- Example

Scan No	*Symbol Scanned*	*Stack*	*P*
6.	+	(+	ABC*+
7.	(	(+(	ABC*+
8.	D	(+(	ABC*+D
9.	*	(+(*	ABC*+D
10.	E	(+(*	ABC*+DE
11.	+	(+(+	ABC*+DE*

Conversion from infix to postfix- Example

Scan No	***Symbol Scanned***	***Stack***	***P***
12.	F	(+(+	ABC*+DE*F
13.	)	(+	ABC*+DE*F+
14.	*	(+*	ABC*+DE*F+
15.	G	(+*	ABC*+DE*F+G
16.	)		ABC*+DE*F+G*+

Conversion from prefix to postfix

- The assumptions are
  - The given prefix expression should be valid.
  - The expression should only contain single character operands i.e. ***a, b,…..z*** and ***0,1,2….9***
  - The valid operators are ***+***, ***-***, *******, ***/***, ***^***

Conversion from prefix to postfix - Procedure

- Let ***Q*** be the arithmetic expression written in prefix notation. ***P*** is the equivalent postfix expression.
  1. Reverse the contents of Q.
  2. Scan ***Q*** from left to right and repeat steps 3 to 5 until Q is completely scanned.
  3. Store the symbol read into a temp variable.

Conversion from prefix to postfix- Procedure – Contd..

4. If the symbol is an operator
   a. Pop the top element from stack and store it in op1.
   b. Pop the next top element from stack and store it in op2.
   c. Copy op1 into P and join op2 into P.
   d. Join temp into P.
   e. Push P into stack.
5. If it is not an operator
   a. Push temp into stack.
6. Stop

Example

- **Q = -A/B * C ^ DE /* prefix Expression*/**
- **Infix expression is**

  A-(B/(C*(D^E)))
- **Postfix Expression is**

  ABCDE^*/-
- Reverse Q.

  ED^C*B/A-

Symbol Scanned	*Temp*	*op1*	*op2*	*P*	**Stack**
E	E				
D	D				E
^	^	D	E	DE^	E , D
					DE^
C	C				DE^, C
*	*	C	DE^	CDE^*	
					CDE^*

Symbol Scanned	*Temp*	*op1*	*op2*	*P*	**Stack**
B	B				CDE^*, B
/	/	B	CDE^*	BCDE^*/	
					BCDE^*/
A	A				BCDE^*/ , A
-	-	A	BCDE^ */	ABCDE^*/-	

ABCDE^*/-

Recursion

- It is a programming technique that allows the programmer to express operations in terms of themselves.
- A definition which defines an object in terms of a simpler case of itself.
- Recursion is the ability of a function to call itself repeatedly.

Recursion

- Every recursive solution involves two major parts
  - Base Case(s) (stopping condition)
  - Recursive Case(s)
- **base case(s)**, in which the problem is simple enough to be solved directly
- **recursive case(s)**
  - every time go near to the stopping condition

Factorial Function

- *Given an integer **n** the **factorial** is defined as the product of all integers between **n** and 1.*
- *It is denoted by **n!**.*
- *It is represented as **1*2*3*……*(n-1)*n**.*
- *In general the definition for factorial of n is defined as*
- $$n! = \begin{cases} 1 & if \;\; n = 0 \\ n * (n-1)! & if \; n > 0 \end{cases}$$

Factorial Function

- Iterative

```
prod = 1;
for(i=n; i>0; i--)
    prod *=i;
return prod;
```

Recursive:

```
if n == 0
  fact = 1;
else
 return (n * fact(n-1))
```

Factorial Function

- Rewrite the recursive function:

```
if n == 0
  fact = 1;
else {
x = n-1;
y= fact(x);
fact = n * y;
}
return fact;
```

C program:

```
int fact(int n)
{
 int x, y;
if (n == 0)
  return 1;
x = n-1;
y= fact(x);
return  (n * y);
}
```

- Of course, it is much simpler and more straight forward to use the iterative method for evaluation of the factorial function.
- Still need to understand the recursive process

Multiplication of Natural Numbers

The product ***a * b***, where a and b are positive integers, may be defined as ***a*** added to itself ***b*** times.

a * b = a if b ==1

a * b = a * (b-1) + a if b >1

6*3= 6*2+6 =6*1+6 +6 = 6+6+6 =18

Multiplication of Natural Numbers

```
int mult (int a, int b)
{
  int c, d,sum;
  if (b == 1)
   return (a);
   c= b-1;
   d=mult(a,c);
    sum =d+a;
    return(sum);
}/*end mult*/
```

Alternatively:

```
int mult(int a, int b)
{
 return (b == 1? a : mult(a, b-1) + a );
}
```

Fibonacci Sequence

0, 1,1,2,3,5,8,13, ……..

- It is a series of integers where each number in the series is the sum of the two preceding numbers.
- The starting values are always ***0*** and ***1***.

fib(n) = n if n ==0 or n ==1

fib(n) = fib(n-2) + fib(n-1) if n > 1

Fibonacci Sequence

Iterative:

```
if ( n<=1)
 return (n)
 fib1 = 0;
 fib2 = 1;
 for (i = 2; i<=n; i++) {
 x = fib1;
 fib1 = fib2;
 fib2 = x + fib1;
}/* end for */
 return (fib2);
```

Alternatively: Recursive

```
int fibo(int n)
{  int x, y;
if ( n<=1)
 return (n)
  x = fibo(n-1);
 y = fibo(n-2);
return (x + y);
}
```

Observations

- In the case of the factorial function, the same number of multiplications must be performed in computing n! by the recursive and iterative methods.
- The same is true of the number additions in the two methods of computing multiplication.
- However in the case of Fibonacci numbers, the recursive method is far more expensive than the iterative.
- **Note: Recursive Chains – Function A calls Function B and Function B calls Function A**

Binary Search

- We now present a recursive algorithm to search a sorted array a for an element x between a[low] and a[high].

```
if (low > high)
return(-1);
mid = (low + high)/2;
 if(x == a[mid])
 return(mid);
 if(x < a[mid])
search for x in a[low] to a[mid – 1];
 else
 search for x in a [mid + 1] to a [high];
```

Binary Search

```
int  bsearch(int a[], int ele, int low, int high)
{
    int mid;
    if (low > high)
      return -1;
   mid = (low + high)/2;
    if(x == a[mid])
       return(mid);
    if(x < a[mid])
      bsearch( a, ele,  low, mid – 1);
    else
      bsearch (a, ele, mid + 1, high);
}
```

Towers of Hanoi

- The **Towers of Hanoi**, is a mathematical game or puzzle.
- It consists of **three rods or Pegs**, and **a number of disks of different sizes** which can slide onto any rod.
- The puzzle starts with the disks neatly stacked in order of size on one rod, the smallest at the top, thus making a conical shape. (always the larger disk is below the smaller disk)

Towers of Hanoi – Contd..

- The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:
  - Only one disk may be moved at a time.
  - Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
  - No disk may be placed on top of a smaller disk.

Towers of Hanoi – Contd..

- The puzzle was invented by the French mathematician Édouard Lucas in 1883.
- There is a legend about a Vietnamese or Indian temple which contains a large room with three time-worn posts in it surrounded by 64 golden disks.

Towers of Hanoi – Contd..

- The priests of Brahma, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the rules of the puzzle, since that time.
- The puzzle is therefore also known as the Tower of Brahma puzzle.
- According to the legend, when the last move of the puzzle is completed, the world will end.

Towers of Hanoi – Contd..

- If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them $2^{64}-1$ seconds or roughly 600 billion years.
- It would take 18,446,744,073,709,551,615 turns to finish.

Towers of Hanoi – Contd..

- The pseudo code.