Stack:

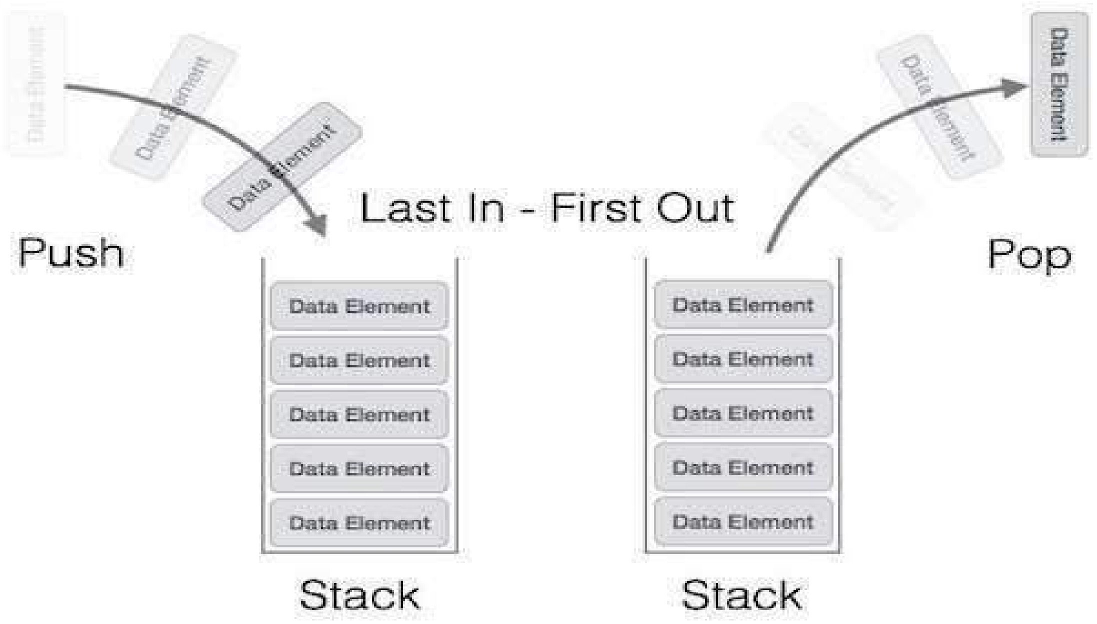
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation:

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations:

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek():

Algorithm of peek() function −

```
begin procedure peek

   return stack[top]

end procedure
```

Implementation of peek() function in C programming language −

```
int peek() {
   return stack[top];
}
```

isfull():

Algorithm of isfull() function −

```
begin procedure isfull


   if top equals to MAXSIZE
      return true
   else
      return
   false endif


end procedure
```

Implementation of isfull() function in C programming language −

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty():

Algorithm of isempty() function −

```
begin procedure isempty


   if top less than 1
      return true
   else
      return
   false endif


end procedure
```

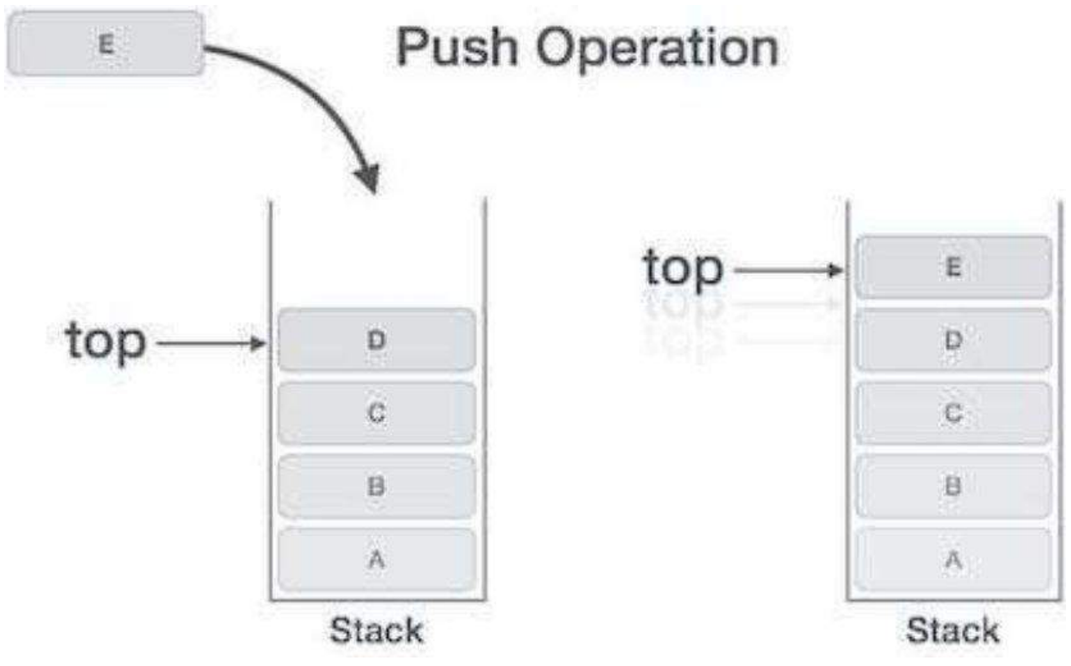
Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation:

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation:

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data


   if stack is full
      return null
   endif


   top ← top + 1


   stack[top] ← data


end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   }else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation:

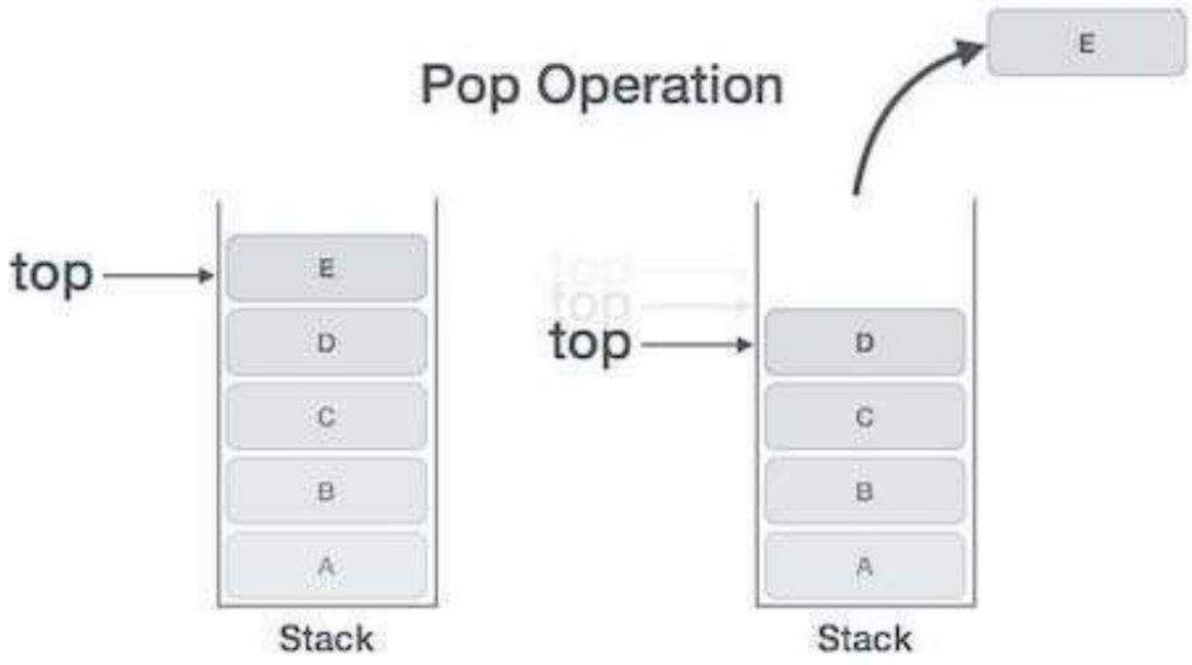
Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.

- **Step 2** − If the stack is empty, produces an error and exit.

- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation:

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]

   top ← top - 1

   return data

end procedure
```

Implementation of this algorithm in C, is as follows −

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   }else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Stack Program in C:

Implementation in C:

```
#include <stdio.h>

int MAXSIZE = 8;
int stack[8];
int top = -1;

int isempty() {

   if(top == -1)
      return 1;
   else
      return 0;
}
```

```
int isfull() {

   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

int peek() {
   return stack[top];
}



int pop() {
   int data;

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   }else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}

int push(int data) {

   if(!isfull()) {
      top = top + 1;
      stack[top] =
   data; }else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

```
int main() {
   // push items on to the
   stack push(3);
   push(5);
   push(9);
   push(1);
   push(12);
   push(15);

   printf("Element at top of the stack: %d\n" ,peek());
   printf("Elements: \n");

    // print stack data
    while(!isempty()) {
      int data = pop();
      printf("%d\n",data);
   }

   printf("Stack full: %s\n" , isfull()?"true":"false");
   printf("Stack empty: %s\n" , isempty()?"true":"false");

   return 0;
}
```

If we compile and run the above program, it will produce the following result −

```
Element at top of the stack: 15
Elements:
15
12
1
9
5
3
Stack full: false
Stack empty: true
```

Polish / Reverse-Polish Expression:

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in **infix** notation, e.g. a-b+c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation (Polish)

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a+b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation (Reversed Polish)

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a+b**.

The following table briefly tries to show the difference in all three notations −

Sr. No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +

5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

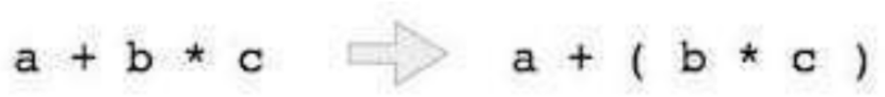
Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −



As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a+b−c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a+b)−c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Sr. No.	Operator	Precedence	Associativity
1	Exponentiation **^**	Highest	Right Associative
2	Multiplication (*****) & Division (**/**)	Second Highest	Left Associative
3	Addition (**+**) & Subtraction (**−**)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a+b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a+b** to be evaluated first, like **(a+b)*c**.

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation −

```
Step 1 – scan the expression from left to right
Step 2 – if it is an operand push it to stack
Step 3 – if it is an operator pull operand from stack and perform operation
Step 4 – store the output of step 3, back to stack
Step 5 – scan the expression until all operands are consumed
Step 6 – pop the stack and perform operation
```

Expression Parsing Using Stack

Infix notation is easier for humans to read and understand whereas for electronic machines like computers, postfix is the best form of expression to parse. We shall see here a program to convert and evaluate **infix** notation to **postfix** notation −

```
#include<stdio.h>
#include<string.h>

//char stack
char stack[25];
int top = -1;

void push(char item) {
   stack[++top] = item;
}

char pop() {
   return stack[top--];
}
```

```
//returns precedence of operators
int precedence(char symbol) {

   switch(symbol) {
      case '+':
      case '-':
         return 2;
         break;
      case '*':
      case '/':
         return 3;
         break;
      case '^':
         return 4;
         break;
      case '(':
      case ')':
      case '#':
         return 1;
         break;
   }
}

//check whether the symbol is operator?
int isOperator(char symbol) {

   switch(symbol) {
      case '+':
      case '-':
      case '*':
      case '/':
      case '^':
      case '(':
      case ')':
         return 1;
      break;
```

```
         default:
         return 0;
      }
   }

   //converts infix expression to postfix void
   convert(char infix[],char postfix[]) {
      int i,symbol,j = 0;
      stack[++top] = '#';

      for(i = 0;i<strlen(infix);i++) {
         symbol = infix[i];

         if(isOperator(symbol) == 0)
            { postfix[j] = symbol;
            j++;
         } else {
            if(symbol == '(') {
               push(symbol);
            }else {
               if(symbol == ')') {

                  while(stack[top] != '(') {
                     postfix[j] = pop();
                     j++;
                  }

                  pop();//pop out (.
               } else {
                  if(precedence(symbol)>precedence(stack[top])) {
                     push(symbol);
                  }else {

                     while(precedence(symbol)<=precedence(stack[top]))
                        { postfix[j] = pop();
                        j++;
                     }
                     push(symbol);
                  }
               }
            }
         }
      }

      while(stack[top] != '#') {
         postfix[j] = pop();
         j++;
      }

      postfix[j]='\0';//null terminate string.
   }

   //int stack
   int stack_int[25];
   int top_int = -1;

   void push_int(int item) {
      stack_int[++top_int] = item;
   }

   char pop_int() {
      return stack_int[top_int--];
   }

   //evaluates postfix expression
   int evaluate(char *postfix){

      char ch;
      int i = 0,operand1,operand2;

      while( (ch = postfix[i++]) != '\0') {

         if(isdigit(ch)) {
```

```c
                push_int(ch-'0'); // Push the operand
            }else {
                //Operator,pop two operands
                operand2 = pop_int();
                operand1 = pop_int();

                switch(ch) {
                    case '+':
                        push_int(operand1+operand2);
                        break;
                    case '-': push_int(operand1-
                        operand2); break;

                    case '*':
                        push_int(operand1*operand2);
                        break;
                    case '/':
                        push_int(operand1/operand2);
                        break;
                }
            }
        }

        return stack_int[top_int];
    }

    void main() {
        char infix[25] = "1*(2+3)",postfix[25];
        convert(infix,postfix);

        printf("Infix expression is: %s\n" , infix); printf("Postfix
        expression is: %s\n" , postfix); printf("Evaluated expression
        is: %d\n" , evaluate(postfix));
    }
```

If we compile and run the above program, it will produce the following result −

```
Infix expression is: 1*(2+3)
Postfix expression is: 123+*
Result is: 5
```

Recursion:

Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function **α** either calls itself directly or calls a function **β** that in turn calls the original function **α**. The function **α** is called recursive function.

Example − a function calling itself.

```
int function(int value) {
   if(value < 1)
      return;
   function(value - 1);

   printf("%d ",value);
}
```

Example − a function that calls another function which in turn calls it again.

```
int function(int value) {
   if(value < 1)
      return;
   function(value - 1);

   printf("%d ",value);
}
```

Properties:

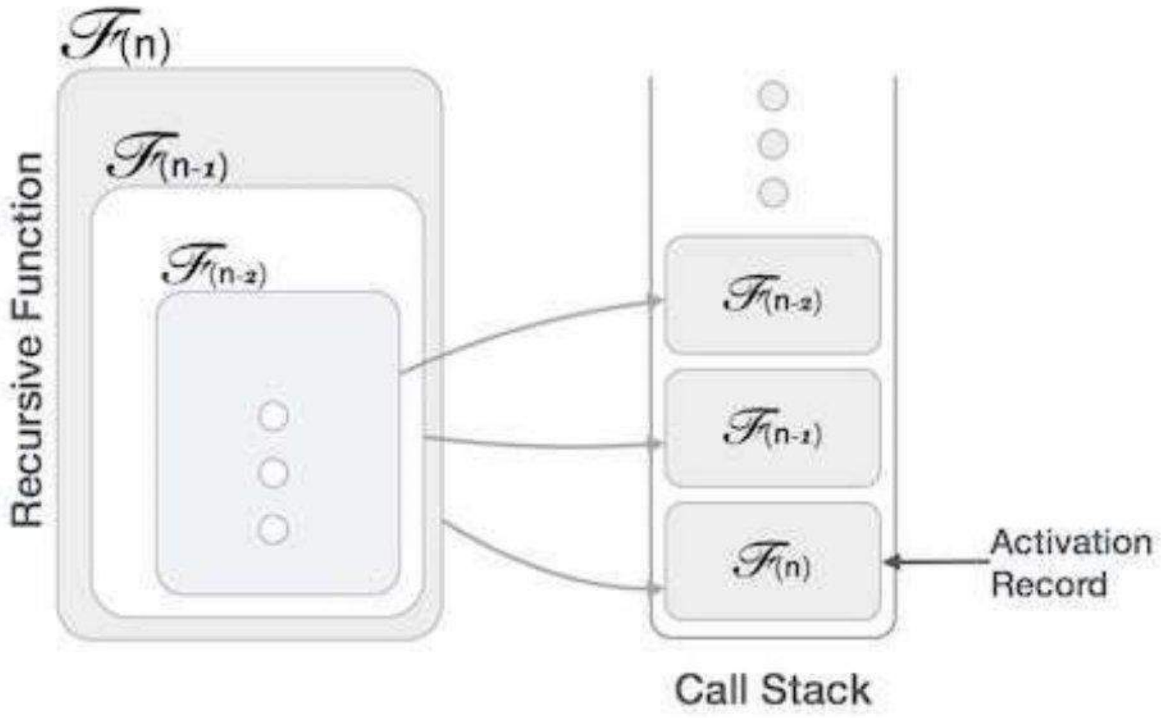
A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

- **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Implementation:

Many programming languages implement recursion by means of **stacks**. Generally, whenever a function (**caller**) calls another function (**callee**) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.

This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.



This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.

Analysis of Recursion:

One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

Time Complexity:

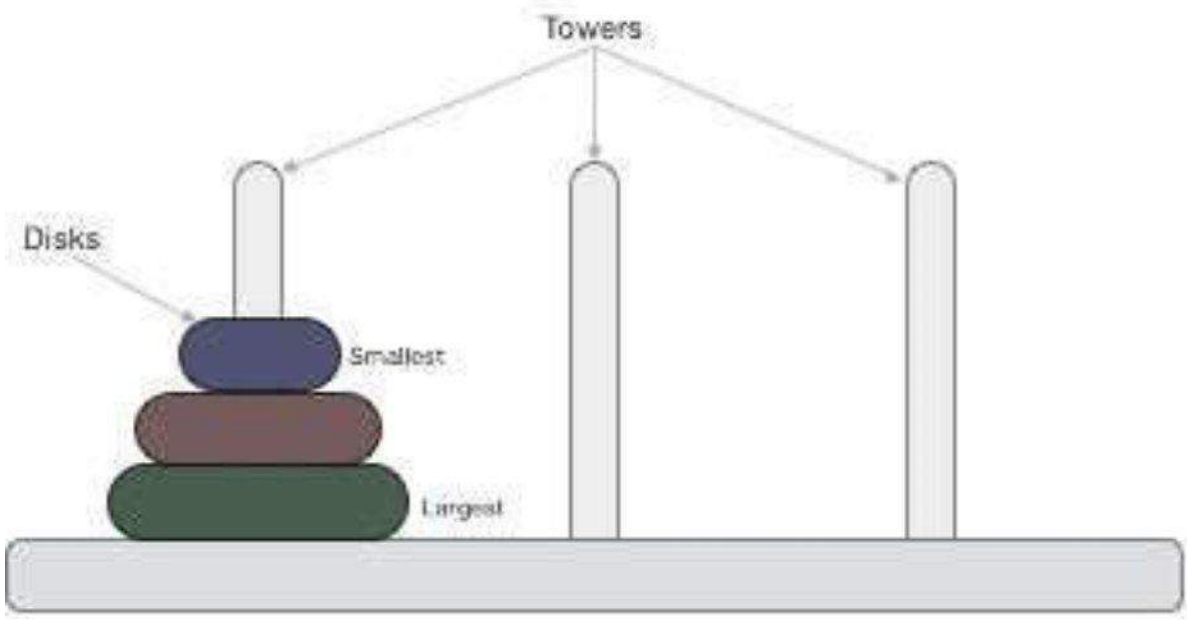
In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is O(1), hence the (n) number of times a recursive call is made makes the recursive function O(n).

Space Complexity:

Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.

Tower of Hanoi:

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −



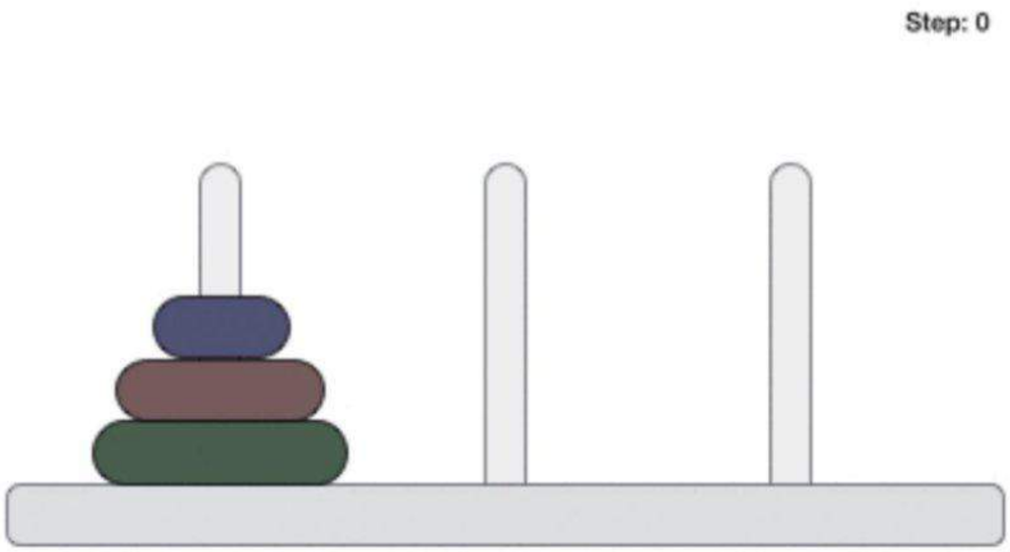
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

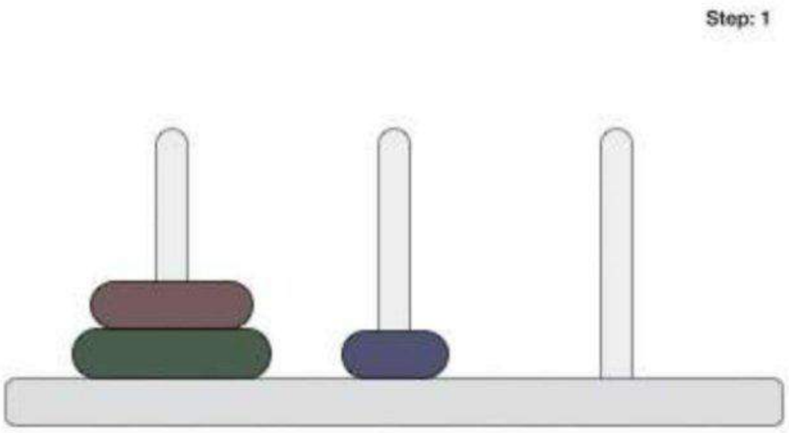
Rules:

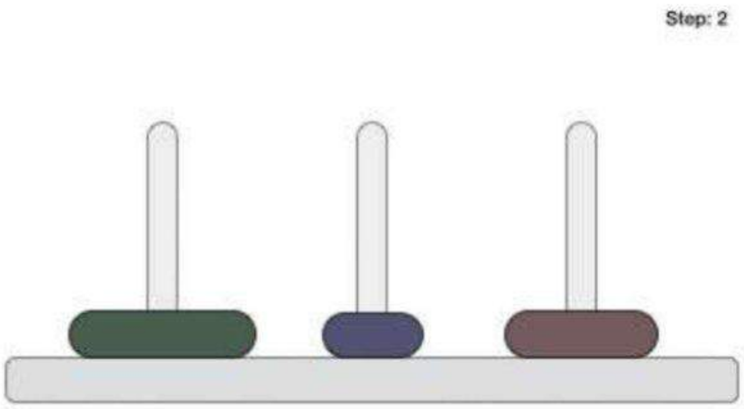
The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

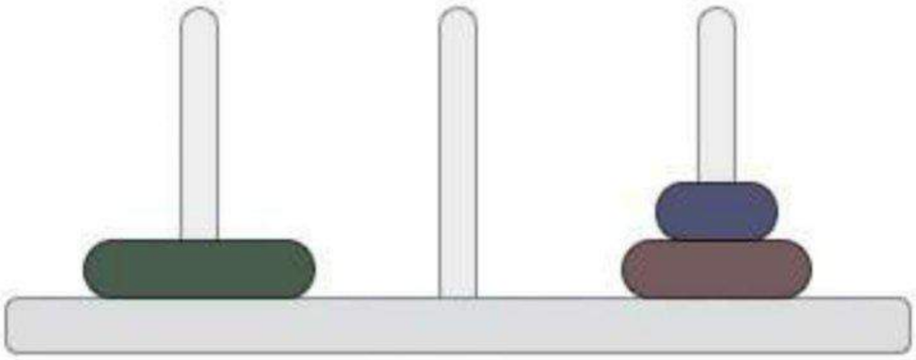
Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



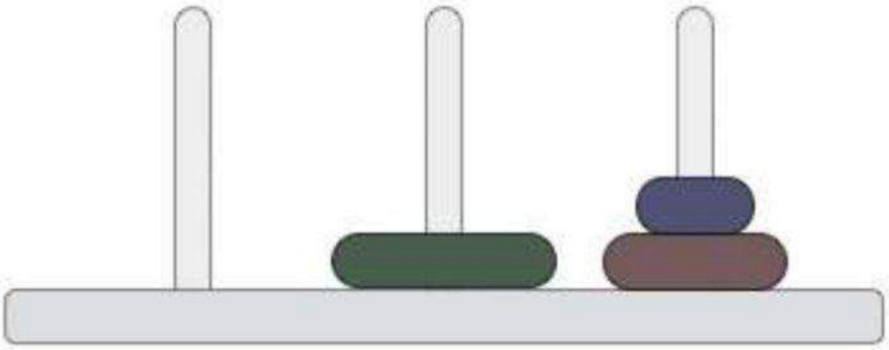




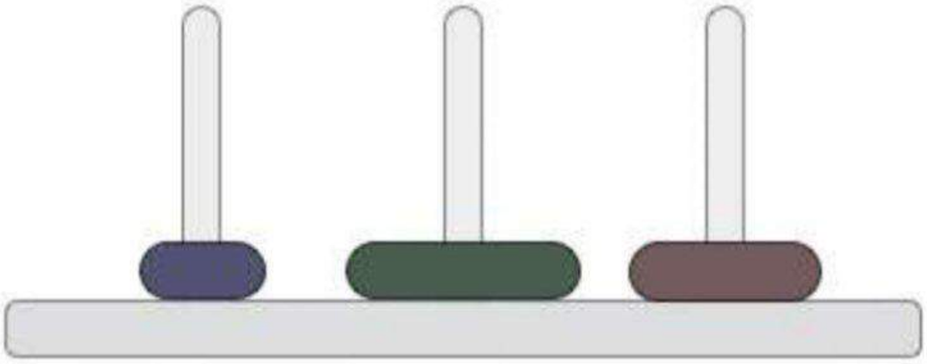
Step: 3

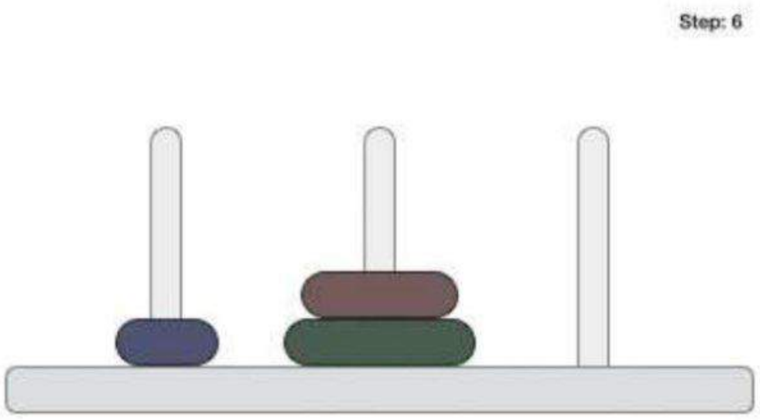


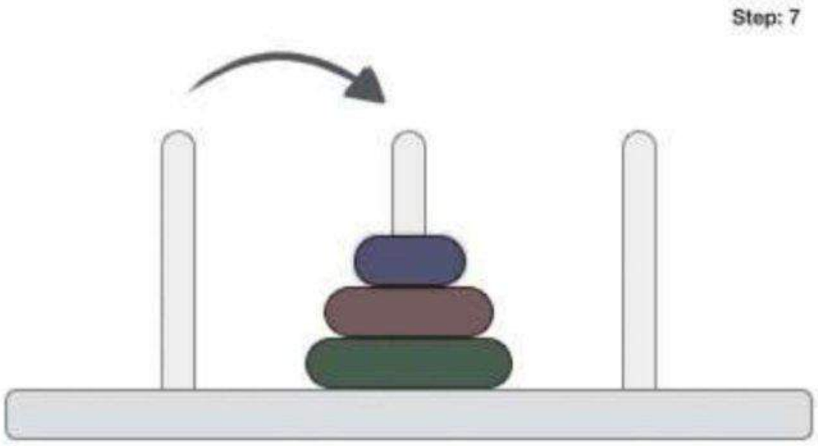
Step: 4

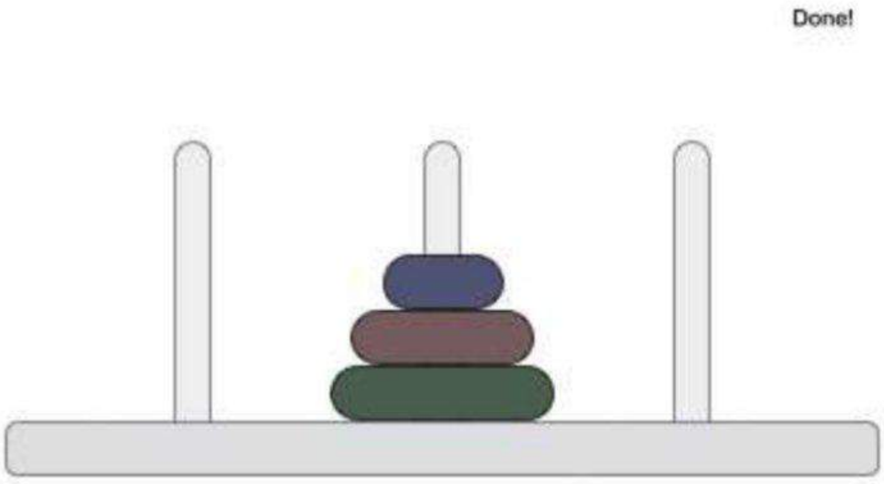


Step: 5









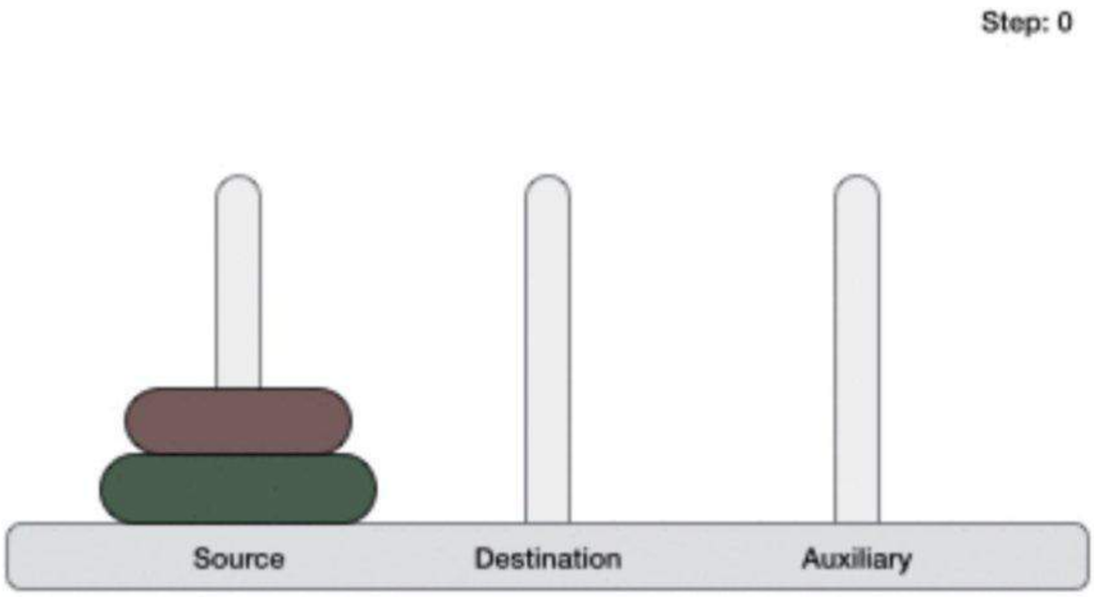
Tower of Hanoi puzzle with n disks can be solved in minimum $\mathbf{2^n-1}$ steps. This presentation shows that a puzzle with 3 disks has taken $\mathbf{2^3-1 = 7}$ steps.

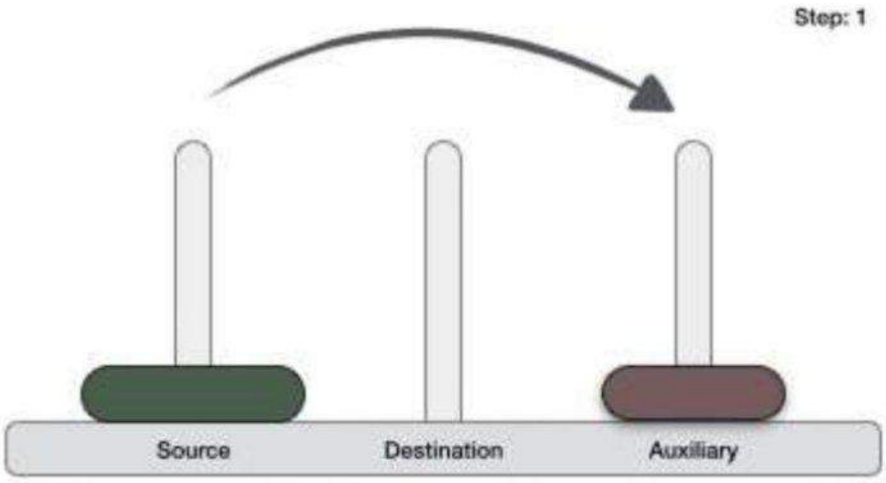
Algorithm:

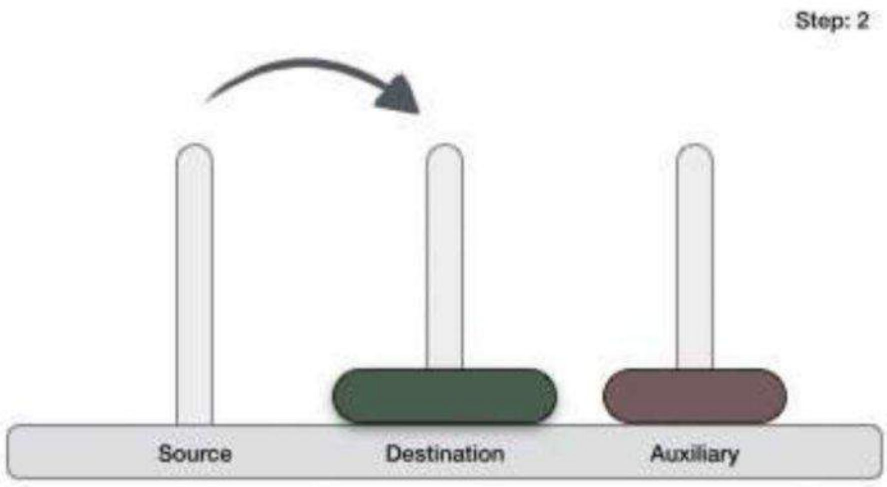
To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

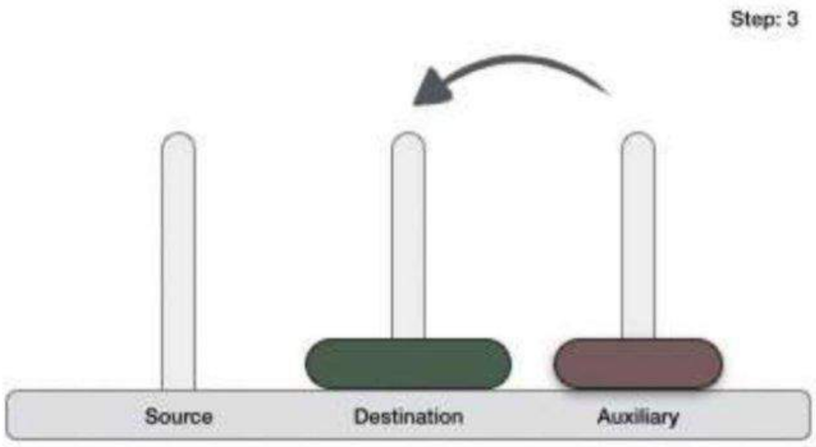
If we have 2 disks –

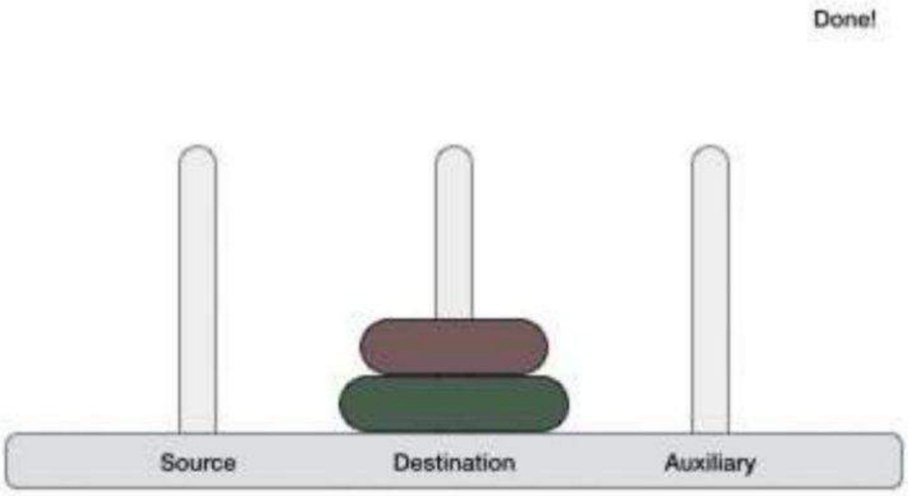
- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.






Step: 2
Source
Destination
Auxiliary


Step: 3
Source
Destination
Auxiliary


Done!
Source
Destination
Auxiliary

So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other (n-1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are −

```
Step 1 − Move n-1 disks from source to aux
Step 2 − Move nth  disk from source to dest
Step 3 − Move n-1 disks from aux to dest
```

A recursive algorithm for Tower of Hanoi can be driven as follows −

```
START
Procedure Hanoi(disk, source, dest, aux)

   IF disk == 0, THEN
      move disk from source to dest
   ELSE
      Hanoi(disk - 1, source, aux, dest)     // Step 1
      move disk from source to dest          // Step 2
      Hanoi(disk - 1, aux, dest, source)     // Step 3
   END IF

END Procedure
STOP
```

Tower of Hanoi in C:

Program:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 10

int list[MAX] = {1,8,4,6,0,3,5,2,7,9};

void display(){
   int i;
   printf("[");

   // navigate through all items
   for(i = 0; i < MAX; i++){
      printf("%d ",list[i]);
   }

   printf("]\n");
}

void bubbleSort() {
   int temp;
   int i,j;
   bool swapped = false;

   // loop through all numbers
   for(i = 0; i < MAX-1; i++) {
      swapped = false;

      // loop through numbers falling
      ahead for(j = 0; j < MAX-1-i; j++) {
         printf("Items compared: [ %d, %d ] ", list[j],list[j+1]);
         // check if next number is lesser than current no
         //   swap the numbers.
         //  (Bubble up the highest number)

         if(list[j] > list[j+1]) {
            temp = list[j];
            list[j] = list[j+1];
            list[j+1] = temp;

            swapped = true;
            printf(" => swapped [%d, %d]\n",list[j],list[j+1]);
         }else {
            printf(" => not swapped\n");
         }
      }

      // if no number was swapped that means
      //   array  is  sorted  now,  break  the
      loop. if(!swapped) {
         break;
      }

      printf("Iteration %d#: ",(i+1));
      display();
   }
}



main(){
   printf("Input Array: ");
   display(); printf("\n");
   bubbleSort();
   printf("\nOutput Array:
   "); display();
}
```

If we compile and run the above program, it will produce the following result –

```
Input Array: [1 8 4 6 0 3 5 2 7 9 ]

     Items compared: [ 1, 8 ]  => not swapped
     Items compared: [ 8, 4 ]  => swapped [4, 8]
     Items compared: [ 8, 6 ]  => swapped [6, 8]
     Items compared: [ 8, 0 ]  => swapped [0, 8]
     Items compared: [ 8, 3 ]  => swapped [3, 8]
     Items compared: [ 8, 5 ]  => swapped [5, 8]
     Items compared: [ 8, 2 ]  => swapped [2, 8]
     Items compared: [ 8, 7 ]  => swapped [7, 8]
     Items compared: [ 8, 9 ]  => not swapped
Iteration 1#: [1 4 6 0 3 5 2 7 8 9 ]
     Items compared: [ 1, 4 ]  => not swapped
     Items compared: [ 4, 6 ]  => not swapped
     Items compared: [ 6, 0 ]  => swapped [0, 6]
     Items compared: [ 6, 3 ]  => swapped [3, 6]
     Items compared: [ 6, 5 ]  => swapped [5, 6]
     Items compared: [ 6, 2 ]  => swapped [2, 6]
     Items compared: [ 6, 7 ]  => not swapped
     Items compared: [ 7, 8 ]  => not swapped
Iteration 2#: [1 4 0 3 5 2 6 7 8 9 ]
     Items compared: [ 1, 4 ]  => not swapped
     Items compared: [ 4, 0 ]  => swapped [0, 4]
     Items compared: [ 4, 3 ]  => swapped [3, 4]
     Items compared: [ 4, 5 ]  => not swapped
     Items compared: [ 5, 2 ]  => swapped [2, 5]
     Items compared: [ 5, 6 ]  => not swapped
     Items compared: [ 6, 7 ]  => not swapped
Iteration 3#: [1 0 3 4 2 5 6 7 8 9 ]
     Items compared: [ 1, 0 ]  => swapped [0, 1]
     Items compared: [ 1, 3 ]  => not swapped
     Items compared: [ 3, 4 ]  => not swapped
     Items compared: [ 4, 2 ]  => swapped [2, 4]
     Items compared: [ 4, 5 ]  => not swapped
     Items compared: [ 5, 6 ]  => not swapped
```

```
Iteration 4#: [0 1 3 2 4 5 6 7 8 9 ]
     Items compared: [ 0, 1 ]  => not swapped
     Items compared: [ 1, 3 ]  => not swapped
     Items compared: [ 3, 2 ]  => swapped [2, 3]
     Items compared: [ 3, 4 ]  => not swapped
     Items compared: [ 4, 5 ]  => not swapped
Iteration 5#: [0 1 2 3 4 5 6 7 8 9 ]
     Items compared: [ 0, 1  ] => not swapped
     Items compared: [ 1, 2  ] => not swapped
     Items compared: [ 2, 3  ] => not swapped
     Items compared: [ 3, 4  ] => not swapped

Output Array: [0 1 2 3 4 5 6 7 8 9 ]
```

References:

1. "Data Structures & Algorithm" , Tutorials Point India (https://www.tutorialspoint.com)