Book

A Simplified Approach
to

Data Structures

Prof.(Dr.)Vishal Goyal, Professor, Punjabi University Patiala

Dr. Lalit Goyal, Associate Professor, DAV College, Jalandhar

Mr. Pawan Kumar, Assistant Professor, DAV College, Bhatinda

Shroff Publications and Distributors

Edition 2014

APPLICATIONS OF STACK

Contents for Today's Lecture

- Evaluation of Arithmetic Expression
- Matching Parenthesis
- Quick Sort
- Recursion

Applications of Stack

The stack is used in wide variety of applications. These are extensively used in system programming (compilers and operating system)and application programming .

Evaluation of Arithmetic Expression

An important application of stack is the compilation of arithmetic expression in the programming languages. The compiler must be able to translate the expression which is written in the usual notation known as **infix notation** to a form which is known as **reverse polish notation**. Compilers accomplish this task of notation conversion i.e. infix to postfix with the help of stack.

Infix Notation

The most common notation that is used while expressing arithmetic expression is infix notation. In this notation, operator is placed between **its operands**. For example, to multiply **m** and **n** we write **m** * **n**. It is the notation that is used by most of the people to solve any mathematical/arithmetic expressions. But, while solving the infix notation, the main consideration is the **precedence** of the operators and their **associativity**. For example, consider an expression,

e = q * r + s

In this expression, following the precedence rules, **q** and **r** are first multiplied and then **s** is added. That is, *(multiplication) operator has precedence over the +(addition) operator.

Infix Notation (Continued)

With this notation, we have to distinguish between(q*r)+s and q*(r+s) by using either operator precedence rule or by applying some parenthesis. Such type of expressions cannot be solved accurately if we do not follow the rules of operator of precedence and their associativity.

The main problem with this notation is that the order of operator and operands in the expression does not uniquely decide the order in which operations are to be carried out.

Infix Notation (Continued)

Table showing the precedence and associativity of the various operators.

Priority	Operator	Associativity
1st	Brackets	Inner to Out and Left to Right
2nd	Exponent ^	Left to Right
3rd	* /	Left to Right
4th	+ -	Left to Right
5th	=	Right to Left

Infix Notation (continued)

Consider an expression,

e = 4 – 2 ^ 4 + 8 * 3 + 18 / 3 + 6

^ having highest precedence over the other operators, will be solved first

e = 4 – 16 + 8 * 3 + 18 / 3 + 6

Now, * and / operations will be performed from left to right because both are having same level of precedence.

e = 4 – 16 + 24 + 18 / 3 + 6

e = 4 – 16 + 24 + 6 + 6

Now, + and – operations will be performed from left to right because both are having same level of precedence.

e = - 12 + 24 + 6 + 6

e = 12 + 6 + 6

e = 18 + 6

e=24

Prefix notation

This notation is also popular with the name **polish notation** which is named after polish mathematician **Jan Lukasiewicz**. In 1920's, this polish mathematician developed a system in which mathematical expression can be specified without parenthesis by placing the operator before or after its operands. In prefix notation, operator is placed before **its operands**. For example, to multiply **m** and **n** we write ***mn**.

The main characteristics of this notation is that the order in which the operations are to be carried out is completely determined by the position of operators and operands in the expression. While solving the arithmetic expression which is written in prefix/polish notation, there is no need to take care if any precedence rule and there is no need to put the parenthesis in the expression.

Prefix Notation (continued)

In order to translate an arithmetic expression from infix to polish notation, we will do it by step by step by using [ ] (**square brackets**)to indicate the **partial conversion.** This means the expression within the square brackets will be treated as a single operand.

Example 1:

$$\mathbf{I_{in}} = (a - b)/ c$$
$$= [- ab] / c$$
$$\mathbf{I_{pre} = / - abc}$$

Example 2:

$$\mathbf{I_{in}} = (x - y)*((z + v) / f)$$
$$= [- xy]*([+ zv] / f)$$
$$= [- xy]*[/ + zvf]$$
$$\mathbf{I_{pre} = * - xy / + zvf}$$

Prefix Notation (continued)

Example 3: $\mathbf{I_{in}}$ = ((a + b) / d ^((e – f) + g))

= ([+ab]/ d ^ ([- ef] + g))

= ([+ab] / d ^ [+ - efg])

= [+ ab] / [^ d + - efg]

$\mathbf{I_{pre}}$ = / +ab ^ d + - efg

Example 4: $\mathbf{I_{in}}$=(x*y) + (z + ((a + b - c) * d)) – I * (j / k)

= (x * y) + (z + (([+ ab] – c) * d)) - i * (j / k)

= (x * y) + (z + ([- + abc] * d)) - i * (j / k)

= (x * y) + (z + [* - + abcd]) - i * (j / k)

xy] + [+ z * - + abcd] – [* i / jk]

= [+ * xy + z * + abcd] - [= [* xy] + (z + [* - + abcd]) - i *(j / k)

= [* xy] + [+ z * - + abcd] – i *(j / k)

= [* xy] + [+ z * - + abcd] - i *(/jk)

= [* * I / jk]

$\mathbf{I_{pre}}$ = - + * xy + z * - + abcd * i / jk

Postfix Notation

The postfix notation is also known as **reverse polish notation**. In this notation, operator is placed **after its operands**. Arithmetic operations like +,*, / and ^ will be shown with its operands **m** and n as **mn +, mn -, mn *, mn / and mn ^** respectively. The fundamental characteristics of this notation is that there is no need of parenthesis to designate the hierarchy of operators. In this notation, order of operations is completely determined by the order of operands and its operators.

In order to convert an arithmetic expression from infix to reverse polish notation, we to indicate the **partial conversion**. This means the expression within the square brackets will be treated as single operand.

Conversion from Infix to Postfix Notation

Example 1:

$$\mathbf{I_{in}} = (a - b)/ c$$
$$= [ab -]/ c$$
$$\mathbf{I_{post} = ab - c/}$$

Example 2:

$$\mathbf{I_{in}} = (x - y) X ((z + v)/ f)$$
$$= [xy -] X ([zv +]/ f)$$
$$=[xy -] X [zv + f /]$$
$$\mathbf{I_{post} = xy - zv + f / X}$$

Note: In computer, an infix notation is evaluated in two steps. In the first step, the expression is converted in the reverse polish notation and in the second step, the converted expression is evaluated. The reason behind this notation conversion is that postfix expression is very efficient for the point of view of evaluation done by computers. As postfix expression is scanned from left to right, operands are simply placed into a stack and operators may be immediately applied to operands which are at the top of the stack. By contrast, expression with parenthesis and precedence (infix notation) require the operators to be delayed until some later point. Thus compilers, in all modern computers convert the arithmetic expressions to reverse polish notation for evaluation.

Algorithm: Convert an arithmetic expression 'I' written in infix notation into its equivalent postfix expression 'p'.)

Step 1: Push a left parenthesis (onto the stack.

Step 2: Append a right parenthesis) at the end of Given expression **I.**

Step 3: Repeat steps from 4 to 8 by scanning **I** character by character from left to right until the stack is empty.

Step 4: If the current character in **I** is a white space, simply ignore it.

Step 5: If the current character in **I** is an operand, write it as the next element of the postfix expression **P**

Step 6: If the current character in **I** is a left parenthesis (, push it onto the stack.

Step 7: If the current character in **I** is an operator , Then

Pop operators (if there is any) at the top of stack while they have **equal or higher precedence** than the current operator and put the popped operators in the postfix expression **P.**

Push the currently scanned operator on the stack.

Step 8: If the current character in **I** is a right parenthesis Then

Pop operators from the top of the stack and insert them in the postfix expression **P** until a left parenthesis is encountered at the top of the stack.

Pop and discard left parenthesis (from the stack.

Step 9: Exit

Consider an expression, **I = (6 +2) * 8/4**

Let us transform this infix expression **I** into its equivalent postfix expression **P** using the algorithm.

I = (6 + 2) * 5 – 8 / 4)

Character scanned	**Status of Stack**	**Postfix expression 'P'**
	(	
(	((	
6	((	6
+	((+	6
2	((+	6 2
)	(	6 2 +
*	(*	6 2 +
5	(*	6 2 + 5
-	(-	6 2 + 5 *
8	(-	6 2 + 5 * 8
/	(-/	6 2 + 5 * 8
4	(-/	6 2 + 5 * 8 4
)	Null	6 2 + 5 * 8 4 / -

Evaluation of Postfix notation

Algorithm: Evaluate an arithmetic expression 'P' written in postfix notation and calculates the result of the expression in variable 'Value'.

Step 1: Scan **P** from left to right and Repeat steps 2 and 3 for each scanned character until end of the expression.

Step 2: If scanned character is an **operand**, push it onto the stack.

Step 3: If the scanned character is an **operator** Then

Pop the two top elements **a** and **b** from the stack where **a** is the top element and **b** is the next to top element.

Apply the operator on the operands **b** and **a** and push the result onto the stack.

[End loop]

Step 4: Set **Value = Stack [Top]**

Step 5: Print: "The value of the expression is ": **Value**

Step 6: Exit

Evaluation of Postfix notation (continued)

Consider the previously converted postfix expression **P = 6 2 + 5 * 8 4 / -** which will be evaluated using algorithm as shown below:

Character Scanned	Status of Stack
6	6
2	6 2
+	8
5	8 5
*	40
8	40 8
4	40 8 4
/	40 2
-	38

The calculated result i.e. **38** of the expression **P** is returned.

Matching parenthesis

A stack can be used for syntax verification of the arithmetic expression for ensuring that for each left parenthesis in the expression there is a corresponding right parenthesis. To accomplish this task of parenthesis matching, the expression is scanned from left to right character by character. Whenever a left parenthesis is encountered, we push it onto the stack. The parenthesis encountered can be of any type, square brace [, round brace (, or curly brace{.When we encounter a right parenthesis], or), or}, the status of the stack is checked. If the stack is empty then we have a right parenthesis in the expression that does not have the corresponding left parenthesis in the expression showing the mistake in the expression. If the stack is not empty, we will pop the topmost element from the stack and compare it with the scanned right parenthesis. If both the parenthesis are not of the same type then it shows a mistake in the expression. But, if both the parentheses are of the same type then the same procedure is repeated until the whole expression is scanned and stack is empty.

Algorithm: Syntax verification by scanning an arithmetic expression 'I' from left to right character by character using a stack.

Step 1: Scan the expression **I** from left to right and Repeat steps 2 to 4 for each scanned character until the end of the expression is reached.

Step 2: If the scanned character is **left parenthesis** then push it onto the stack.

Step 3: If the scanned character is an **operator** or **operand** then ignore it.

Step 4: If the scanned character is a **right parenthesis** Then

a) If **Stack[Top] = Null** Then
Print "There is no left parenthesis corresponding to right parenthesis".
Exit

[End If]

Algorithm (continued)

b. Pop the top element from the stack and compare it with currently scanned right parenthesis.

c. If both are **not corresponding** Then

Print “The braces are not in proper order”.

Exit

[End If]

[End Loop]

Step 5: If **Stack [Top] != Null** Then

Print: “There is no right parenthesis corresponding to the left parenthesis”.

Exit

[End If]

Step 6: Exit.

Example

Let us check the order of braces in an arithmetic expression **I** using above algorithm.

I = [(5 + 6) * 7 – {7 / 4} + (3 * 2) – 8]

Character Scanned	Status of Stack
[	[
(	[(
)	[
{	[{
}	[
(	[(
)	[
]	Null

Quick Sort

Quick sort is an important application of stack which was developed by **C.A.R Hoare** in the year **1962**. This sort is also popular with the names **Partition Exchange Sort** or **Hoare's Quick Sort**. Before discussing about the quick sort, let us discuss briefly about general sorting concept .Sorting means, the arrangement of the elements of the list in some logical order. If the elements of the list are numeric numbers then sorting refers to arranging them in increasing or decreasing order. On the other hand, if the list has alphabetic elements then sorting refers to be alphabetic increasing or decreasing arrangements of the elements.

Quick Sort (continued)

Quick sort algorithm which is an important application of stack uses **divide and conquer** policy for sorting the list of elements. In divide and conquer policy, the problem to be solved is divided into sub- problems repeatedly until we reach the smallest size sub-problems whose solution is easy to find. Then solution of these small sub-problems is combined to obtain the solution of the whole problem.

In quick sort strategy, the problem of sorting the given list of element is reduced to sorting the smaller subsets. The quick sort strategy can be better explained by talking an example of a list having unsorted elements.

Quick Sort (continued)

Consider an unsorted list of 10 elements:

5	8	2	11	1	33	4	3	100	7
5	8	2	11	1	33	4	3	100	**7**
									←

In the first pass of the quick sort algorithm, the first element of the list will occupy its correct position in the list. In the above list of numbers, the correct position of the first element in the list will be found and will be occupied by it. This task will be accomplished by scanning the list from the right to left starting from right most position of the list i.e. from element 7. While scanning the list each element will be compared with the first element**,** we will stop the scanning and interchange the first element with the recently scanned element. Here, in the given list while scanning the list from **right to left** the first element 5 will be interchange with the element as shown below:

Quick Sort (continued)

3 8 2 11 1 33 4 **5** 100 7

←

Now, starting from first position i.e. from element **3**, scan the list from **left to right** by comparing each element with **5**.This time meeting an element larger than the element 5, we will stop scanning the list and Here in the given list, the element 5 will be interchanged with the element 8 as shown below.

3 **5** 2 11 1 33 4 **8** 100 7

←

Now, starting from element **8**, scan the list from **right to left** by comparing each element with **5**.

Meeting an element smaller than the element **5**, we stop scanning the list and interchange the currently scanned element with element **5**. Here in the given list, the element **5** will be interchanged with the element **4** as shown below:

3 **4** 2 11 1 33 **5** 8 100 7

→

Quick Sort (continued)

Now, starting from element **4**, scan the list from left to right by comparing each element with **5**. Meeting and interchange the currently scanned element with **5**. Here in the given list, the element 5 will be interchanged with the element as shown below.

3 4 2 **5** 1 33 **11** 8 100 7

←

Now, starting from element **11**, scan the element from **right to left** by comparing each element with **5**. Meeting an element smaller than the element **5**,we stop scanning the list and interchange the currently scanned element with **5**.Here in the given list, the element **5** will be interchange the element 1 as shown below.

3 4 2 **1** **5** 33 11 8 100 7

→

Quick Sort (continued)

Now, starting from element **1**, scan the list from **right to left** by comparing each element with **5**. Meeting an element larger than the element **5**, we stop scanning the list and interchange the element with **5**. Here, in the given list, this time there is no element which is larger than **5**. It means the element **5** is at the correct position in the list and all the elements which are smaller than **5** are on the left side of the **5** and all the elements which are larger than **5** are on the right side of the element **5**.

3 4 2 1 **5** 33 11 8 100 7

Left Sublist **Right Sublist**

Now, the task of sorting is reduced to sorting the two sublists (left sublist and right sublist).

Quick Sort (continued)

The same reduction procedure will be repeated on each sublist having two or more elements. We can process only one sublist at a time, so we have to postpone the processing of other sublist. This postponed processing of sublist can be easily accomplished by sorting the lower and upper indices of each sublist into the two different stacks. The reduction procedure will be applied on the sublists after removing their lower and upper indices from the stacks named **Lbstack** and **Ubstack** respectively.

Quick Sort (continued)

The use of the stack for sorting a linear list **'L'** having **12** numbers is explained as follows:

29	35	42	17	39	12	25	54	10	72	19	85
1	2	3	4	5	6	7	8	9	10	11	12

An Unsorted Array 'L' of 12 Elements

Initially, the lower bound and lower upper bound of given linear list will be pushed onto the two different stacks named **Lbstack** and **Ubstack** respectively as shown below:

Lbtack **Ubstack**

Stacks containing the lower bound and Upper Bound of the list

Quick Sort (continued)

Now, the first reduction step will be performed on the list after popping the indices of the list from both the stacks will become empty.

29 35 42 17 39 12 25 54 10 72 19 **85**

←

19 35 42 17 39 12 25 54 10 72 **29** 85

→

19 **29** 42 17 39 12 25 54 10 72 **35** 85

←

19 **10** 42 17 39 12 25 54 **29** 72 35 85

→

19 10 **29** 17 39 12 25 54 **42** 72 35 85

←

19 10 **25** 17 39 12 **29** 54 42 72 35 85

→

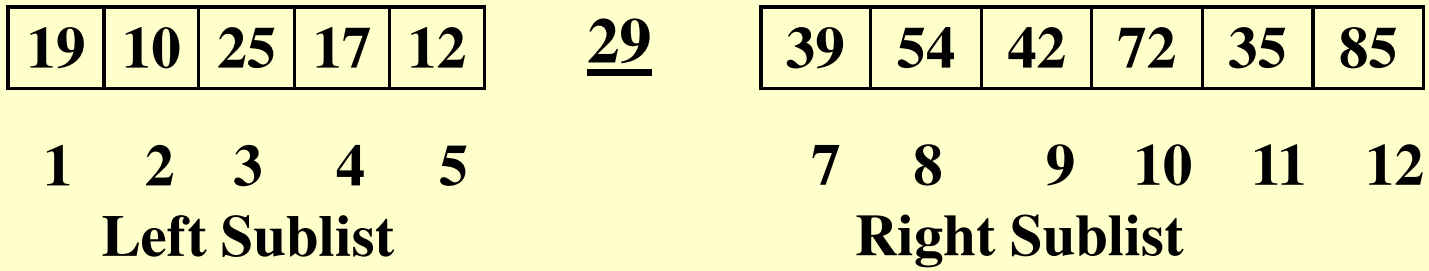
19 10 25 17 **29** 12 **39** 54 42 72 35 85

←

19 10 25 17 **12** **29** 39 54 42 72 35 85

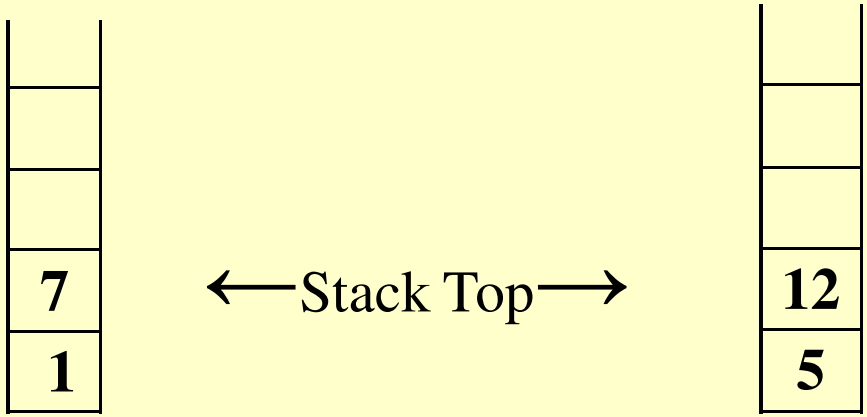
→

Quick Sort (continued)



Two Sublists Created After The First Reduction Step

After the completion of first reduction step, the first element has occupied the correct position in the list and two sublists have been created as shown in figure. The lower and upper indices of newly created sublists will be pushed into the stacks **LbStack** and **UbStack** as shown in figure.



Quick Sort (continued)

Now, the same procedure will be applied on the sublist whose lower and upper indices will be popped from top of the stacks **LbStack** and **UbStack** respectively. Here, in the example, the reduction step will be applied on the right sublist first whose lower and upper indices are 7 and 12 respectively. After the completion of the reduction step, the element at index number 7 i.e. 39 will occupy the correct position in the sublist and the sublist will be divided into two new sublists whose indices will be pushed onto the stacks. This procedure will be r**e**peated until the whole list is stored.

Algorithm: Sort an array 'L' with 'n' elements
Quick sort (L, n)

Step 1: Set **stacktop =** ∅
Step 2: If n > 1 Then
Set **stacktop = 1**
Set **LbStack [stacktop] =1**
Set **UbStack [stacktop] = n**
[End If]
Step 3: Repeat Steps 4 to 7 while **stacktop =**
Step 4: Set **Begin = LbStack [stacktop]**
Set **End = UbStack [stacktop]**
Set **stacktop = stacktop – 1**
Step 5: **Loc = Splitpass (L, Begin, End)**

Algorithm (continued)

Step 6: If **Begin < Loc - 1** Then
Set **stactop = stacktop +1**
Set **Lbstack [stacktop] = Begin**
Set **UbStack [stacktop] = Loc -1**
[End If]
Step 7: If **End > Loc + 1** Then
Set **stacktop = stacktop + 1**
Set **LbStack [stacktop] = Loc + 1**
Set **UbSatck [stacktop] = End**
[End If]
[End Loop]
Step 8: Exit

Algorithm: Put the first element of the sublist 'L' passed to it at its correct position and returns the new location of the first element.

SplitPass (L, Begin, End)

Step 1: Set Left = **Begin, Right = End, Loc = Begin** And Flag **=False**

Step 2: Repeat steps 3 to 6 While **Flag = False**

Step 3: Repeat While **L [Loc] < =L [Right] And Loc !=Right**

Set **Right= Right – 1**

[End Loop]

Step 4: If **Loc = Right** Then

Set **Flag = True**

Else If **L[Loc] > L[Right]** Then

Interchange L[Loc] and L[Right]

Set **Loc = Right**

[End If]

Algorithm (continued)

Step 5: Repeat While **L[Loc] > = L[Left] AND Loc !=Left**
Set **Left =Left + 1**
[End Loop]
Step 6: If **Loc = Left** Then
Set **Flag= True**
Else If **L[Loc] < L[Left]** Then
Interchange L[Loc] and L[Left]
Set **Loc = Left**
[End If]
[End Loop]
Step 7: Return **Loc**

Complexity Analysis of Quick Sort Algorithm

Complexity of a sorting algorithm is represented by function f(n) i.e. number of comparisons required to sort the list of elements. While analyzing quick sort algorithm, the worst case occurs when after each reduction step, the list is portioned into two sublists with one of them being empty. Such a situation occurs only when the given lists of elements is already sorted. In this situation, the 1^{st} element will be compared with n - 1 elements to remain at its original position. After the completion of first reduction step, one of the two siblists formed will be empty and another will have n - 1 elements. During the second reduction step, 2^{nd} element will be compared with n- 2 elements to remain at its original position and so on. So, in the worst case, the total numbers will be:

$$f(n) = (n-1) + (n-2) + (n-3) + \ldots\ldots + 3 + 2 + 1 = 0(n^2)$$

Complexity Analysis of Quick Sort Algorithm

•The average case occurs while sorting the list, when after each reduction reduction step of the algorithm; it produces two sublists of approximately same size. To calculate the average case complexity of the quick sort algorithm, two assumptions are to be made.

•The size of the list **n** should be the power of 2 i.e. $\mathbf{n = 2^m}$**,** for some positive integer value of **m.**

•After each reduction step sublists formed are approximately equal size.

In average case analysis, there will be exactly **n - 1** comparisons during the first reduction step that produces two sublist of size each. In the second reduction step, there will be approximately comparisons for each sublists that produces for sublists of size each.

Complexity Analysis of Quick Sort Algorithm

This procedure will continue until there is **n** sublists of size 1 each. Here, total number of comparisons **f(n)** will be:

$$f(n) = n + 2 \times \frac{n}{2} + 4 \times \frac{n}{4} + \cdots + n \times \frac{n}{n}$$
$$= n + n + n + \cdots + n$$
$$= m \times n = \log_2 n \times n$$
$$f(n) = n \log_2 n \qquad asm = \log_2 n$$
$$f(n)\ 0 = (n\ \log_2 n)$$

In the general case, where the size of the list is not in the power of 2 and sublists formed may not be of equal size, the complexity can be calculated but the procedure is very complex and is beyond the scope of this book but in that case the resultant complexity will be same i.e. **O(n log2 n).**

Recursion

Recursion is very important and powerful tool for developing algorithms for various problems. Recursion is the ability of a procedure either to call itself or calling to some other procedure may result in call to the original procedure. In computer science, solution of many problems can be best defined recursively. Two very important conditions/ requirements that must be satisfied by any procedure to be defined recursively are:

- There must be a decision criterion that stops the further call to the procedure called **base criteria**.
- Each time a procedure calls itself either directly or indirectly, it must be **nearer to the solution** i.e. nearer to the base criteria.

Recursion (continued)

A procedure having these two properties is called a well defined procedure and can be defined recursively. Recursive procedure can be implemented in various programming languages but compilers of some programming language are not able to handle recursive procedure because they do not have stack mechanism required by the recursive procedures. Programming languages such as PASCAL, ALGOL, C, C++ can be used to implement recursive procedure calls.

Examples of some Recursively defined problems

Most of the readers may be familiar with procedure for calculating the factorial of a positive integer or finding the n^{th} term of a Fibonacci series. Both of these problems can be defined recursively.

Recursion (continued)

Factorial Function

The factorial of a posititve number **n** is the product of positive integers from **1 to n** The Factorial of a number is represented symbolically by placing a symbol '!' next to it. The factorial of a positive integer **n** will be defined as:

$$n! = 1 \times 2 \times 3 \times 4 \times \ldots\ldots\ldots \times (n-1) \times n$$

The value of the factorial function for zero is taken as 1. Let us know calculate the factorial of some positive integers.

$$3! = 3 \times 2 \times 1 = 6$$

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$$

Recursion (continued)

The factorial for the same numbers can also be defined as:

$$4! = 4 \times 3!$$

$$5! = 5 \times 4!$$

$$6! = 6 \times 5!$$

Thus, the formal definition of the factorial function can be given as:

$$n! = \begin{cases} 1 & if\ n = 0 \\ n \times (n-1)! & if\ n \geq 1 \end{cases}$$

This definition of calculating the factorial of a number is recursive as it meets both the conditions which are required to solve a problem recursively.

Recursion (continued)

Algorithm: Calculate the value of n! recursively

Factorial (n)
If **n = 0** Then
Set **Fact = 1**
Return
Else $n \times Factorial(n-1)$
Set **Fact =**
Return
[End If]

Recursion (continued)

Fibonacci Series

A Fibonacci series is a sequence of numbers which is usually denoted by $F_0, F_1, F_2 F_3, \ldots\ldots, F_n$. The series is as shown below:
0, 1, 2, 3, 5, 8, 13, 21,……
Here, $F_0=0$, $F_1=1$ *and* $F_2=F_0+F_1$, $F_3= F_1+F_2$, $F_4= F_2+F_3$ and so on.
Generally speaking, in a Fibonacci series, each succeeding term is a sum of two preceding terms. The recursive procedure for finding the nth term of the Fibonacci series can be defined as:

$$Fibo(n) = \begin{cases} n & if\ n = 0\ or\ 1 \\ Fibo(n-1) + Fibo(n-2) & if\ n > 1 \end{cases}$$

Recursion (continued)

Algorithm: Find the n^{th} term of a Fibonacci series recursively.

Fibonacci (n)

If **n =0** Then

    Set **Fibo = 0**

Return

Else if **n = 1** Then

    Set **Fibo = 1**

  Return

Else

  Set **Fibo = Fibonacci (n – 1) + Fibonacci (n -**2)

        Return

[End If]

When to use Recursion

Recursion is generally used for repetitive computations in which each action is defined in terms of previous result.
While making a decision about whether to use recursive procedure or non- recursive procedure, there is not any hard and fast rule for the selection. There are many factors that affect the choice of procedure for solving a given problem:

- Computer Memory Required
- Processing Time Required
- Time Required for developing the Algorithm
- Time Required for Debugging

It is always advisable to consider a tree structure for a given problem. If the tree structure is simple then use of non-recursive procedure is suitable. If the tree structure appears quite bushy with duplication of tasks, then recursive procedure is suitable.

Demerits of recursion

1. Many programming languages do not support recursion. Hence recursive mathematical function is implemented using interactive methods.
2. Even though mathematical functions can be easily implemented using recursion it is always at the cost of execution time and memory space. For example, the recursion tree for generating 6 numbers in a Fibonacci series generation is given in previous figure. A Fibonacci series is of the form 0, 1, 2, 3, 5, 8, 13, …..etc, where a number is the sum of preceding two numbers. It can be noticed from the figure that, f(n-2) is computed twice, f(n-3) is computed thrice, f(n-4) is computed 5 times.
3. A recursive procedure can be called from within or outside itself and to show its proper functioning it has to save the return addresses in same order so that, a return to the proper location will result when the return to a calling statement is made.
4. A special care is required to put a stopping condition at which the recursive function will stop.

Demerits of iterative methods

1. Mathematical functions such as factorial and Fibonacci series generation can be easily implemented using rather than iteration.
2. In iterative techniques looping of statement is very much necessary.

Recursion is a **top down** approach to problem solving. It divides the problem into pieces or selects one key step postponing the rest.

Iteration is a bottom up approach. It begins with, what is known and from this constructs the solution step by step. The interactive function obviously uses time that is O(n) where as recursive function has an exponential time complexity.

It is always true that recursion can be replaced by iteration and stack.

It is also true that stack can be replaced by a recursive program with no stack.