

Module 7

Recursion

7.1 Recursion as a Different way of Solving Problem

One of the special features of C language is its support to recursion. Very few computer languages will support this feature.

Recursion can be defined as the process of a function by which it can call itself. The function which calls itself again and again either directly or indirectly is known as recursive function.

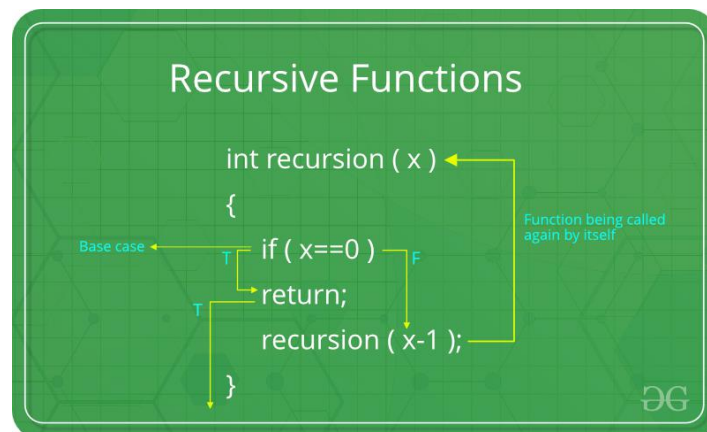
The normal function is usually called by the main () function, by means of its name. But, the recursive function will be called by itself depending on the condition satisfaction.

For Example,

```
main ( )
{
    f1( ) ; ——— Function called by main
    -----
    -----
    -----
}
f1( ) ; ——— Function definition
{
    -----
    -----
    -----
    f1( ) ; ——— Function called by itself
}
```

In the above, the main () function is calling a function named f1() by invoking it with its name. But, inside the function definition f1(), there is another invoking of function and it is the function f1() again.

In programming terms a recursive function can be defined as a routine that calls itself directly or indirectly.



Base Case:

One critical requirement of recursive functions is termination point or base case. Every recursive program must have base case to make sure that the function will terminate. Missing base case results in unexpected behaviour.

How a particular problem is solved using recursion?

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

Why Stack Overflow error occurs in recursion?

If the base case is not reached or not defined, then the stack overflow problem may arise.

7.2 Example Programs

7.2.1 Finding Factorial

Factorial can be calculated using following recursive formula.

$$n! = n * (n-1)!$$
$$n! = 1 \text{ if } n = 0 \text{ or } n = 1$$

Program:

```
#include<stdio.h>

// function to find factorial of given number
unsigned int factorial(unsigned int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}

int main()
{
    int num = 5;
    printf("Factorial of %d is %d", num, factorial(num));
    return 0;
}
```

7.2.2 Fibonacci Series

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,

In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation.

$$F_n = F_{n-1} + F_{n-2}$$

with seed values
 $F_0 = 0$ and $F_1 = 1$.

Program:

```
#include<stdio.h>
#include<conio.h>
int Fibonacci (int);
void main ( )
{
    int i, n; clrscr ( );
    printf ("Enter no. of Elements to be generated" \n)
    scanf ("%d", &n);
    for (i=1; i<=n; i++)
        printf ("%d", Fibonacci (i));
    getch( );
}

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

7.2.3 Reverse a String using Recursion

Recursive function (reverse) takes string pointer (str) as input and calls itself with next location to passed pointer (str+1). Recursion continues this way, when pointer reaches '\0', all functions accumulated in stack print char at passed location (str) and return one by one.

Program:

```
#include <stdio.h>

/* Function to print reverse of the passed string */
void reverse(char *str)
{
    if (*str)
    {
        reverse(str+1);
        printf("%c", *str);
    }
}

int main()
{
    char a[] = "BCE PATNA";
    reverse(a);
    return 0;
}
```

7.2.4 GCD of two Numbers

GCD (Greatest Common Divisor) or HCF (Highest Common Factor) of two numbers is the largest number that divides both of them.

Program:

```
#include <stdio.h>

// Recursive function to return gcd of a and b
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

int main()
{
    int a = 98, b = 56;
    printf("GCD of %d and %d is %d ", a, b, gcd(a, b));
    return 0;
}
```

7.2.5 Ackermann function

Ackermann function is defined as follows

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

Program:

```
#include<stdio.h>
Long int A(int m, int n);

main()
{
    int m,n;
    printf("Enter two numbers :: \n");
    scanf("%d%d",&m,&n);
    printf("\nOUTPUT :: %d\n",A(m,n));
}

Long int A(int m, int n)
{
    if(m==0)
        return n+1;
    else if(n==0)
        return A(m-1,1);
    else
        return A(m-1,A(m,n-1));
}
```

7.2.6 Quick Sort

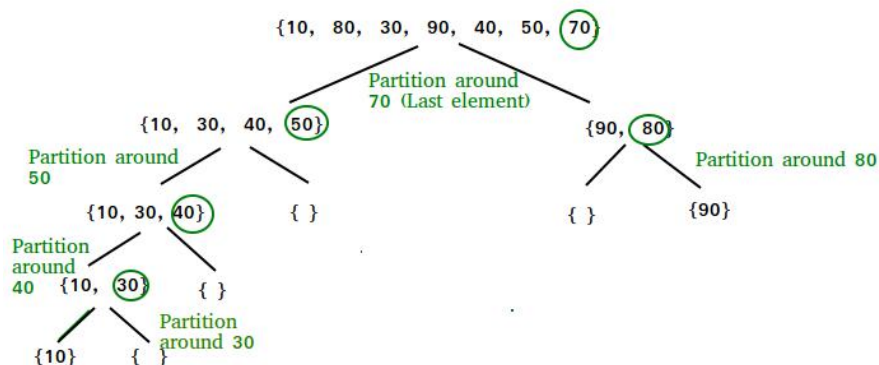
Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Example:



Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

- Step 1 – Choose the highest index value has pivot
- Step 2 – Take two variables to point left and right of the list excluding pivot
- Step 3 – left points to the low index
- Step 4 – right points to the high
- Step 5 – while value at left is less than pivot move right
- Step 6 – while value at right is greater than pivot move left
- Step 7 – if both step 5 and step 6 does not match swap left and right
- Step 8 – if left \geq right, the point where they met is new pivot

Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as –

```
function partitionFunc(left, right, pivot)
    leftPointer = left
    rightPointer = right - 1

    while True do
        while A[++leftPointer] < pivot do
            //do-nothing
        end while

        while rightPointer > 0 && A[--rightPointer] > pivot do
            //do-nothing
        end while

        if leftPointer >= rightPointer
            break
        else
```

```

        swap leftPointer,rightPointer
    end if

end while

swap leftPointer,right
return leftPointer

end function

```

Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows –

Step 1 – Make the right-most index value pivot

Step 2 – partition the array using pivot value

Step 3 – quicksort left partition recursively

Step 4 – quicksort right partition recursively

Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm –

```

procedure quickSort(left, right)

    if right-left <= 0
        return
    else
        pivot = A[right]
        partition = partitionFunc(left, right, pivot)
        quickSort(left,partition-1)
        quickSort(partition+1,right)
    end if

end procedure

```

Program:

```

#include<stdio.h>

// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
    }
}

```

```

        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low  --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

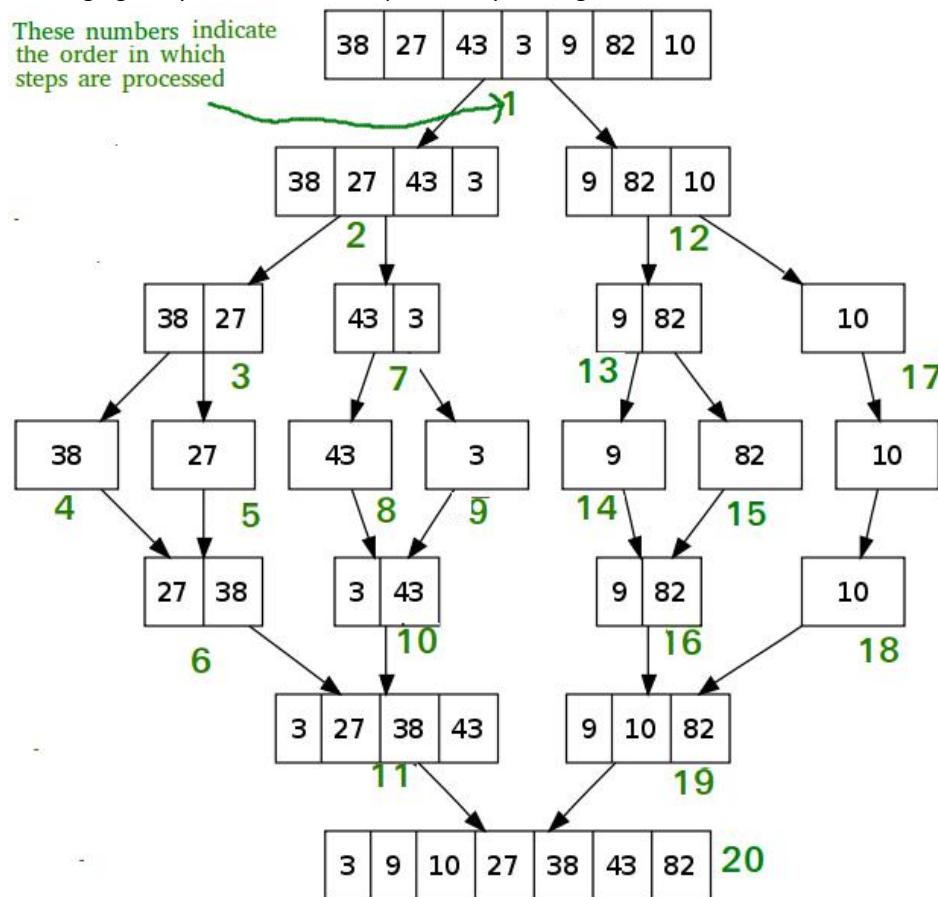
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    printf("Sorted array: n");
    printArray(arr, n);
    return 0;
}

```

7.2.7 Merge Sort

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

The following diagram from shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 – if it is only one element in the list it is already sorted, return.

Step 2 – divide the list recursively into two halves until it can no more be divided.

Step 3 – merge the smaller lists into new list in sorted order.

Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions – divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

```
procedure mergesort( var a as array )
    if ( n == 1 ) return a

    var l1 as array = a[0] ... a[n/2]
    var l2 as array = a[n/2+1] ... a[n]

    l1 = mergesort( l1 )
```



```

l2 = mergesort( l2 )

return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

var c as array
while ( a and b have elements )
    if ( a[0] > b[0] )
        add b[0] to the end of c
        remove b[0] from b
    else
        add a[0] to the end of c
        remove a[0] from a
    end if
end while

while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
end while

while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
end while

return c

end procedure

```

Program:

```

#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];

void merging(int low, int mid, int high) {
    int l1, l2, i;

    for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
        if(a[l1] <= a[l2])
            b[i] = a[l1++];
        else
            b[i] = a[l2++];
    }

    while(l1 <= mid)
        b[i++] = a[l1++];

    while(l2 <= high)

```

```

        b[i++] = a[l2++];

    for(i = low; i <= high; i++)
        a[i] = b[i];
}

void sort(int low, int high) {
    int mid;

    if(low < high) {
        mid = (low + high) / 2;
        sort(low, mid);
        sort(mid+1, high);
        merging(low, mid, high);
    } else {
        return;
    }
}

int main() {
    int i;

    printf("List before sorting\n");

    for(i = 0; i <= max; i++)
        printf("%d ", a[i]);

    sort(0, max);

    printf("\nList after sorting\n");

    for(i = 0; i <= max; i++)
        printf("%d ", a[i]);
}

```