

RECURSION

FACTORIAL

Aim: To find the factorial of a given number using recursion.

Theory: Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop. Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

The factorial of a positive number “n” refers to the product of all the descending integers before it (smaller than the number x). The factorial of a number is denoted by the symbol “!”. Thus, the factorial of a number will be “number!”. For instance, we will write the factorial of the non-negative integer x as x!.

Here,

$$x! = x * (x-1) * (x-2) * (x-3) * (x-4) \dots *$$

Let us look at an example,

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

We can also write it as:

$$6! = 6 * 5!$$

Program:

```
#include<stdio.h>                                //preprocessor directives
int fact(int);                                   //function declaration
int main()                                       //main function
{
    int n,f;                                    //initialize variables
    printf("Enter the number:");               //prints output
    scanf("%d",&n);                             //reads value
    f=fact(n);                                  //function call
    printf("factorial=%d",f);                   //prints factorial
}
int fact(int n)                                 //called function
{
    if(n==0)                                    //conditional statement
    {
        return 0;                             //returns 0
    }
    if(n==1)                                    //conditional statement
    {
        return 1;                             //returns 1
    }
    else
        return n*fact(n-1);                   //recursive function call
}
```

Algorithm:

Step 1: Initialize a value 'n'

Step 2: If $n=0$

Then return 0

Go to step 3

If $n=1$

Then return 1

End if

Go to step 3

Else

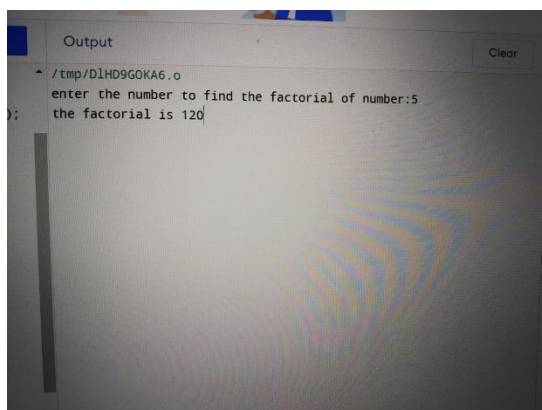
Return $n * \text{fact}(n-1)$

End if

Go to step 3

Step 3: End

Output:



```
Output
/tmp/D1HD9GOKA6.o
enter the number to find the factorial of number:5
the factorial is 120
```

BINARY SEARCH

Aim: To search a number in a list using binary search.

Theory: Binary search is a form of recursion. A binary search algorithm is used to find the position of a specific value contained in a sorted array. Working with the principle of divide and conquer, this search algorithm can be quite fast, but the caveat is that the data has to be in a sorted form. It works by starting the search in the middle of the array and working going down the first lower or upper half of the sequence. If the median value is lower than the target value, that means that the search needs to go higher, if not, then it needs to look on the descending portion of the array.

A binary search is a quick and efficient method of finding a specific target value from a set of ordered items. By starting in the middle of the sorted list, it can effectively cut the search space in half by determining whether to ascend or descend the list based on the median value compared to the target value.

Using binary search, the target only had to be compared to three values. Compared to doing a linear search, it would have started from the very first value and moved up, needing to compare the target to eight values. A binary search is only possible with an ordered set of data; if the data is randomly arranged, then a linear search would yield results all the time while a binary search would probably be stuck in an infinite loop.

Program:

```
#include<stdio.h>          //preprocessor directives
int binarysearch(int array[],int x,int low,int high)  //called function
{
    if(high>=low)          //conditional statement
```

```

{
    int mid=low+(high-low)/2;          //to find the middle element
    if(array[mid]==x)                  //condition check
    {
        return mid;                   //returns mid
    }
    //to search at the left side
    else if(array[mid]>x)
    {
        return binarysearch(array,x,low,mid-1); //recursive function call
    }
    //to search at right side
    else
        return binarysearch(array,x,mid+1,high //recursive function call
    }
    return -1;                        //returns -1
}

int main()
{
    int array[]={ 3,4,5,6,7,8,9};      //array initialization
    int x=4;
    int n=sizeof(array)/sizeof(array[0]); //array size
    int res=binarysearch(array,x,0,n-1); //function call
    if(res==-1)                        //conditional statement if result is invalid
    {
        printf("invalid");
    }
}

```

```
else
    printf("The element is found at %d position",res);    //print output
}
```

Algorithm:

Step 1: Initialize values mid, low, high.

Step 2: if low>high

 return false

 else

 Let $mid = (low + high) / 2$

 end if

Step 3: if $x == arr[mid]$

 return mid

 end if

 if $x > arr[mid]$

 low=mid+1

 else

 high=mid-1

 end if

Step 4: Exit

Output:

```
Output
^ /tmp/D1HD9G0KA6.o
ns The element is found at 1 position
ray
0;
```

TOWER OF HANOI

Aim: To implement tower of Hanoi which uses recursion using stacks.

Theory: Tower of Hanoi is a mathematical puzzle where we have three rods (A, B, and C) and N disks. Initially, all the disks are stacked in decreasing value of diameter i.e., the smallest disk is placed on the top and they are on rod A. The objective of the puzzle is to move the entire stack to another rod (here considered C), obeying the following simple rules:

Only one disk can be moved at a time.

Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

No disk may be placed on top of a smaller disk.

The idea is to use the helper node to reach the destination using recursion. Below is the pattern foris problem:

Shift 'N-1' disks from 'A' to 'B', using C.

Shift last disk from 'A' to 'C'.

Shift 'N-1' disks from 'B' to 'C', using A.

Program:

```
#include<stdio.h>                                //preprocessor directives

void towerofhanoi(int n,char from_rod,char to_rod,char aux_rod)    //function
{
    if(n==1)                                       //conditional statement
    {
        printf("Move disk 1 from rod %c to rod %c",from_rod,to_rod);    //prints
```



```

    return ;
}
towerofhanoi(n-1,from_rod,aux_rod,to_rod);    //recursive function call
printf("Move disk %d from rod %c to rod %c",n,from_rod,to_rod);
towerofhanoi(n-1,aux_rod,to_rod,from_rod);    //recursive function call
}
int main()                                    //main function
{
    int n=4;                                  //initialize n=4
    towerofhanoi(n,'A','C','B');              //function call
    return 0;                                  //returns 0
}

```

Algorithm:

Step 1: procedure hanoi(disk, source, dest, aux)

Step 2: if disk=1 then

 move disk from source to dest

 else

 hanoi(disk-1, source, aux, dest)

 move disk from source to dest

 hanoi(disk-1, aux, dest, source)

 end if

Step 3: Exit

Output:

Run

Output

Clear

▲ /tmp/DlHD9G0KA6.o

```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

to_rod

11

17:36