

Question:1

Write a program to form sparse matrix and print all the non-zero elements with their location address.

Algorithm:

STEP 1: Start

STEP 2: Read number of rows and columns in variables m and n

STEP 3: Declare a matrix of the size m x n

STEP 4: Read the elements of the matrix

STEP 5: Check each element whether they are zero or non-zero. For each non-zero elements, display them with their location i.e. rows and columns

STEP 6: Stop

Example:

Let's suppose that the size of sparse matrix is 3 x 3, and is given by

	3	0	0
M [3][3] =	0	2	0
	1	0	0

In the above sparse matrix, the non-zero elements and their locations are:

Non-zero elements	location
3	(0,0) M[0][0]
2	(1,1) M[1][1]
1	(2,0) M[2][0]

Program:

```
#include <stdio.h>

int main() {
    int m, n;

    // Input rows and columns
    printf("Enter the number of rows: ");
    scanf("%d", &m);
    printf("Enter the number of columns: ");
    scanf("%d", &n);

    // Declare a 2D array for the matrix
    int matrix[m][n];

    // Input the matrix elements
    printf("Enter the elements of the matrix:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            printf("Enter a[%d][%d]= ", i+1, j+1);
            scanf("%d", &matrix[i][j]);
        }
    }
}
```

```

    }
}

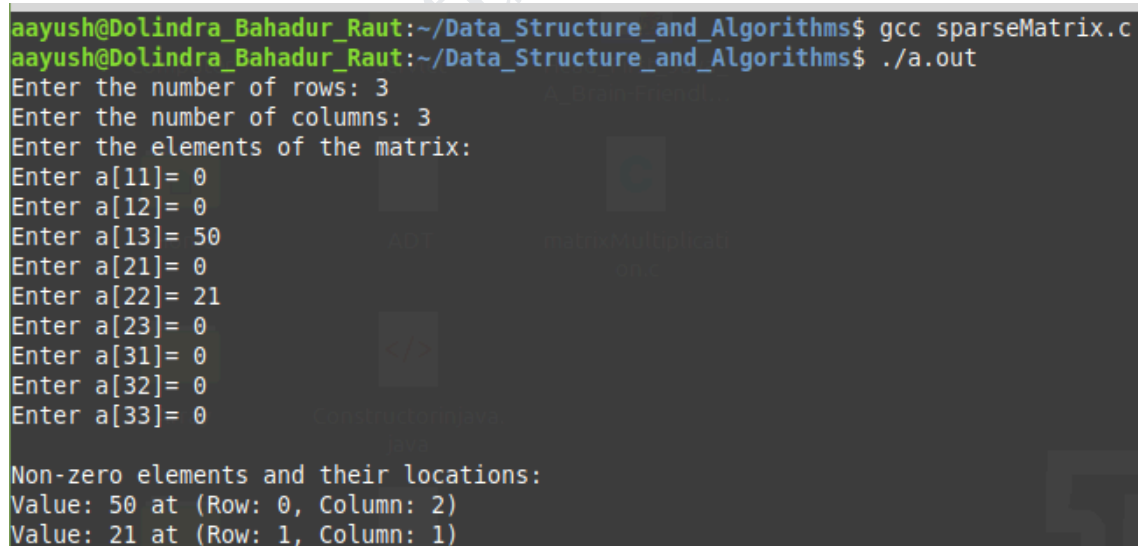
// Print non-zero elements with their locations

printf("\nNon-zero elements and their locations:\n");

for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (matrix[i][j] != 0) {
            printf("Value: %d at (Row: %d, Column: %d)\n",
                matrix[i][j], i, j);
        }
    }
}

return 0;
}

```

Output:


```

aayush@Dolindra_Bahadur_Raut:~/Data_Structure_and_Algorithms$ gcc sparseMatrix.c
aayush@Dolindra_Bahadur_Raut:~/Data_Structure_and_Algorithms$ ./a.out
Enter the number of rows: 3
Enter the number of columns: 3
Enter the elements of the matrix:
Enter a[11]= 0
Enter a[12]= 0
Enter a[13]= 50
Enter a[21]= 0
Enter a[22]= 21
Enter a[23]= 0
Enter a[31]= 0
Enter a[32]= 0
Enter a[33]= 0

Non-zero elements and their locations:
Value: 50 at (Row: 0, Column: 2)
Value: 21 at (Row: 1, Column: 1)

```

Conclusion:

Hence, we successfully created a sparse matrix that takes input from the user and also, the non-zero elements with their location have been displayed.

Question: 2

Define recursion with suitable example. Write algorithm and program with output of the following:

- a. Calculate factorial for n integer number**
- b. Prime number checking**
- c. Fibonacci series**
- d. Tower of Hanoi**

Answer:

Recursion is a programming technique where a function calls itself to solve a problem. It divides the problem into smaller, similar sub problems. Each recursive call works on a simpler version of the original problem. The recursion continues until a base case is reached, which stops further calls. The function then combines the results of the sub problems to solve the original problem.

Here is an example of recursion to find the sum of the first n natural numbers:

Problem: The sum of the first n natural numbers can be represented as:

$$S(n)=1+2+3+...+n$$

This can be defined recursively as:

- Base case: $S(1)=1$ (the sum of the first 1 natural number is just 1).
- Recursive case: $S(n)=n+S(n-1)$ (the sum of the first n numbers is n plus the sum of the first n-1 numbers).

This can be implemented using c as below:

```
int sumOfNaturalNumbers(int n) {
    if (n == 1) {
        return 1;
    }
    else {
        return n + sumOfNaturalNumbers(n - 1);
    }
}
```

Calculate factorial for n integer number**Algorithm:**

Step1: Start

Step2: Define a recursive function calcfactorial(n):

If (n == 0 || 1): return 1

else return n * calcfactorial(n - 1)

Step3: Take input n from the user.

Step4: Pass the n as parameter into the recursive function calcfactorial(n)

Step5: Print the result.

Step6: Stop.

Example:

Let suppose, n = 4

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1!$

Hence, $4! = 4 * 3 * 2 * 1$

Program:

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

```
int calcfactorial(int n){
```

```
    if(n==0 || n==1){
```

```
        return 1;
```

```
    }
```

```
    else{
```

```
        return n * calcfactorial(n-1);
```

```
    }
```

```
}
```

```

int main(){
    int num;

    printf("Enter any positive number: ");
    scanf("%d",&num);

    printf("factorial of %d = %d\n",num,calcfactorial(num));

    return 0;
}

```

Output:

```

aayush@Dolindra_Bahadur_Raut:~/DSA$ gcc factorial.c -o factorial
aayush@Dolindra_Bahadur_Raut:~/DSA$ ./factorial
Enter any positive number: 4
factorial of 4 = 24
aayush@Dolindra_Bahadur_Raut:~/DSA$

```

Conclusion:

Therefore, we have used the recursion to find the factorial of the given integer n.

b. Prime number checking**Algorithm:**

Step 1: Start

Step 2: Define a recursive function checkprime(n, i), where:

n is the number to check.

i is the divisor (starting from n-1 and decreasing).

Step 3: Base conditions:

If n is less than or equal to 1, return 0 (not prime).

If i is equal to 1, return 1 (no divisors found, so prime).

Step 4: Check divisibility:

If n is divisible by i, return 0 (not prime).

Otherwise, call checkprime(n, i - 1) (check next smaller number).

Step 5: Call the function with checkprime(n, n - 1).

Step 6: Print "Prime" if the function returns 1, otherwise print "Not Prime".

Step 7: Stop.

Example: Checking if 3 is Prime or not**Step 1:** Call checkprime(3, 2):

n = 3, i = 2

Check if $3 \% 2 == 0$. This is false (3 is not divisible by 2), so we proceed to the next step and call checkprime(3, 1).

Step 2: Call checkprime(3, 1):

n = 3, i = 1

We reach the base case where $i == 1$, so return 1, meaning 3 is prime.

Result:

Since the function reaches the base case without finding any divisors other than 1, it concludes that **3 is prime**.

Program:

```
#include <stdio.h>

// Recursive function to check if a number is prime
int checkprime(int n, int i) {
    if (n <= 1)
        return 0; // Not prime
    if (i == 1)
        return 1; // Prime (no divisors found)
    if (n % i == 0)
        return 0; // Not prime

    return checkprime(n, i - 1); // Check next divisor
}

int main() {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
```

```

if (checkprime(n, n - 1))
    printf("%d is a Prime Number.\n", n);
else
    printf("%d is Not a Prime Number.\n", n);

return 0;
}

```

Output:

```

aayush@Dolindra_Bahadur_Raut:~/DSA$ gcc checkprime.c -o checkprime
aayush@Dolindra_Bahadur_Raut:~/DSA$ ./checkprime
Enter a number: 3
3 is a Prime Number.
aayush@Dolindra_Bahadur_Raut:~/DSA$ 

```

Conclusion:

Hence we gave a recursive solution to check whether the given number is prime or not.

c. Fibonacci series**Algorithm:**

Step 1: Start.

Step 2: Define a recursive function calculateFibonacci(n):

If $n == 0$, return 0 as the base case.

If $n == 1$, return 1 as the second base case.

Otherwise, return $\text{calculateFibonacci}(n-1) + \text{calculateFibonacci}(n-2)$.

Step 3: Prompt the user to enter a number n (which represents the number of terms to display in the Fibonacci sequence).

Step 4: Use a loop to call the calculateFibonacci(n) function for each number starting from 0 to $n-1$, and print the result for each call.

Step 5: End.

Example:

Let suppose, we have to find 8 terms of fibonacci series, then:

- $f(0) = 0$
- $f(1) = 1$
- $f(2) = f(1) + f(0) = 1 + 0 = 1$
- $f(3) = f(2) + f(1) = 1 + 1 = 2$
- $f(4) = f(3) + f(2) = 2 + 1 = 3$
- $f(5) = f(4) + f(3) = 3 + 2 = 5$
- $f(6) = f(5) + f(4) = 5 + 3 = 8$
- $f(7) = f(6) + f(5) = 8 + 5 = 13$

Thus, the Fibonacci series of the first 8 terms is: 0 1 1 2 3 5 8 13.

Program:

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

```
// Recursive function to calculate Fibonacci number
```

```
int calcfibonacci(int n) {
    if (n == 0) {
        return 0; // Base case: f(0) = 0
    } else if (n == 1) {
        return 1; // Base case: f(1) = 1
    } else {
        return calcfibonacci(n - 1) + calcfibonacci(n - 2); // Recursive case
    }
}
```

```

int main() {
    int terms;

    printf("Enter the no.of terms for Fibonacci series: ");

    scanf("%d", &terms);

    printf("Fibonacci series of %d terms:\n", terms);

    for (int i = 0; i < terms; i++) {
        printf("%d ", calcfibonacci(i)); // Print Fibonacci number for each term
    }

    printf("\n");

    return 0;
}

```

Output:

```

aayush@Dolindra_Bahadur_Raut:~/DSA$ gcc calcfibonacci.c
aayush@Dolindra_Bahadur_Raut:~/DSA$ ./a.out
Enter the no.of terms for Fibonacci series: 8
Fibonacci series of 8 terms:
0 1 1 2 3 5 8 13
aayush@Dolindra_Bahadur_Raut:~/DSA$ 

```

Conclusion:

Hence we gave recursive solution to generate fibonacci series of n terms where n is given by the user.

d. Tower of Hanoi**Algorithm:**

Step1: Start.

Step2: Define a recursive function: towerOfHanoi (n, from_rod, to_rod, aux_rod).

If (n == 1) : move the disk from (from_rod) to (to_rod). //Base case

else: //Recursive case

Call towerOfHanoi(n - 1, from_rod, aux_rod, to_rod).

Move the disk from from_rod to to_rod.

Call towerOfHanoi(n - 1, aux_rod, to_rod, from_rod).

Step3: Take input n (number of disks).

Step4: Call the recursive function towerOfHanoi(n, from_rod, to_rod, aux_rod).

Step5: Stop.

Example:

Let suppose user gives 3 for the value of n (no. of disks) then the recursive solution to the problem of Tower of Hanoi for 3 disks is given by,

1. Disk 1 moved from rod A to rod C.
 - The smallest disk (disk 1) is moved from rod A to rod C.
2. Disk 2 moved from rod A to rod B.
 - The second disk (disk 2) is moved from rod A to rod B.
3. Disk 1 moved from rod C to rod B.
 - The smallest disk (disk 1) is moved from rod C to rod B to allow room for disk 2.
4. Disk 3 moved from rod A to rod C.
 - The largest disk (disk 3) is moved from rod A to rod C.
5. Disk 1 moved from rod B to rod A.
 - The smallest disk (disk 1) is moved back from rod B to rod A.
6. Disk 2 moved from rod B to rod C.
 - Disk 2 is moved from rod B to rod C, on top of disk 3(largest disk).
7. Disk 1 moved from rod A to rod C.

- Finally, disk 1 is moved from rod A to rod C, completing the puzzle of tower of hanoi.

So, all 3 disks are successfully moved from rod A to rod C in the correct order (with the largest disk at the bottom and the smallest disk at the top).

Program:

```
#include <stdio.h>

// Recursive function to solve Tower of Hanoi

void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {

    if (n == 1) {

        printf("Disk 1 is being shifted from rod %c to rod %c.\n", from_rod, to_rod);

        return;

    }

    //Move top n-1 disks from 'from_rod' to 'aux_rod' using 'to_rod' as auxiliary
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);

    //Move the nth disk from 'from_rod' to 'to_rod'
    printf("Disk %d is being shifted from rod %c to rod %c.\n", n, from_rod, to_rod);

    //Move n-1 disks from 'aux_rod' to 'to_rod' using 'from_rod' as auxiliary
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);

}

int main() {

    int n ; //number of disks

    printf("Enter the no. of disks: ");

    scanf("%d", &n);

    printf("\nRecursive solution to the Tower of Hanoi problem with %d disks:\n", n);
```

```
towerOfHanoi(n, 'A', 'C', 'B'); // Call the recursive function

return 0;

}
```

Output:

```
aayush@Dolindra_Bahadur_Raut:~/DSA$ gcc TOH.c
aayush@Dolindra_Bahadur_Raut:~/DSA$ ./a.out
Enter the no. of disks: 3

Recursive solution to the
Tower of Hanoi problem with 3 disks:
Disk 1 is being shifted from rod A to rod C.
Disk 2 is being shifted from rod A to rod B.
Disk 1 is being shifted from rod C to rod B.
Disk 3 is being shifted from rod A to rod C.
Disk 1 is being shifted from rod B to rod A.
Disk 2 is being shifted from rod B to rod C.
Disk 1 is being shifted from rod A to rod C.
aayush@Dolindra_Bahadur_Raut:~/DSA$
```

Conclusion:

Hence we gave a recursive solution to the problem of the tower of hanoi.