

Institute of Science and Technology
Tribhuvan University



Lab Report
On
Discrete Structure (CSC165)

Submitted to:
Mr. Prithvi Raj Paneru

Submitted by:
Kiran Gautam
Roll no: 70

Department of Computer Science and Information Technology
Prithivi Narayan Campus, Pokhara

Lab Index

Name: Kiran Gautam

Roll. No: 70

Level: Bachelors

Year/Sem: 1st/2nd

Subject: Discrete Structure

Faculty: B.Sc. CSIT

Course No: CSC165

Lab.no.	Experiment		Date of Submission	Remarks
1	1.1	To implement conjunction, disjunction, and negation.	2080-04-21	
	1.2	To implement implication and bi-conditional.	2080-04-21	
	1.3	To generate TT of given proposition i) $[p \rightarrow q \wedge q \rightarrow r] \rightarrow (p \rightarrow r)$.	2080-04-21	
	1.4	To check whether the given proposition is tautology, contradiction or contingency.	2080-04-21	
2	2.1	To implement Euclidean Algorithm.	2080-05-15	
	2.2	To implement Extended Euclidean Algorithm.	2080-05-15	
	2.3	To implement Addition and Multiplication Algorithm of two binary numbers.	2080-05-26	
	2.4	To find Boolean join and Boolean product of Boolean Matrix.	2080-05-26	

3	3.1	To solve different types of recursive problems.	2080-05-29	
	3.2	To solve TOH problem.	2080-05-29	
	3.3	To implement merge sort Algorithm.	2080-05-29	

Experiment No: 1.1

To implement conjunction, disjunction and negation.

1. Objectives:

- To implement conjunction, disjunction and negation using C language and Boolean function.
- To utilize the <stdbool.h> library for Boolean data types and operations.

2. Theory:

Conjunction, disjunction, and negation are fundamental logical operations in the field of discrete mathematics and computer science. These operations are widely used in programming, circuit design, and various decision-making processes. In this theoretical explanation, we will discuss the implementation of these operations in the C programming language.

Conjunction (AND):

Given two propositions p and q, the proposition conjunction $p \wedge q$ is the proposition that is true whenever both propositions (p and q) are true, false otherwise.

Truth Table

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

Disjunction (OR)

Given two propositions p and q, the proposition disjunction $p \vee q$ is the proposition that is false whenever both propositions (p and q) are false, true otherwise.

Truth Table

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

Negation

The result is the opposite of the operand. If the operand is true, the result is false, and vice versa. The operation is denoted by !.

Truth Table

p	Negation
T	F
F	T

3. Demonstration:

3.1. Source Code:

/C program to illustrate conjunction, disjunction and negation.

/C program to illustrate conjunction, disjunction and negation.

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

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

```
bool conjunction(bool a, bool b){
```

```

        return a&& b;
    }
    bool disjunction(bool a, bool b){
        return a||b;
    }
    bool negation(bool a){
        return !a;
    }
    void display(bool a){
        if(a==true)
        {
            printf("T\t");
        }
        else
        {
            printf("F\t");
        }
    }
    int main()
    {
        int i;
        bool res;
        bool a[]={true,true,false,false};
        bool b[]={true,false,true,false};
        printf("p\tq\tp^q\tpvq\t!p\n");
        printf("_____ \n");
        for(i=0;i<4;i++)
        {
            display(a[i]);
            display(b[i]);
            res=conjunction(a[i],b[i]);
            display(res);

            res=disjunction(a[i],b[i]);
            display(res);

            res=negation(a[i]);
            display(res);
            printf("\n");
        }
        return 0;
    }

```

3.2. Output and Discussion:

p	q	$p \wedge q$	$p \vee q$	$\neg p$
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

We have generated a truth table for the logical operations of conjunction (AND), disjunction (OR), and negation (NOT) applied to two propositions, denoted as "p" and "q." This truth table was created using the logical operators "&&" (AND), "||" (OR), and "!" (NOT).

4. Conclusion:

In this lab, we've effectively applied the logical operations AND, OR, and NOT in C programming. We employed the `<stdbool.h>` library, which uses true and false for boolean data types, enhancing code clarity and maintainability. Hence, we can illustrate conjunction, disjunction and negation of proposition using truth table in C.

Experiment 1.2

To implement implication and bi-conditional.

1. Objectives:

- To implement implication and bi- conditional.
- To utilize the <stdbool.h> library for Boolean data types and operations.

2. Theory:

Conditional ($p \rightarrow q$):

The proposition implication, represented as ($p \rightarrow q$), is a statement that is only false when the premise (p) is true while the conclusion (q) is false; otherwise, it is true. In simpler terms, it means that if the initial condition (p) is met, then the resulting condition (q) must also be true for the implication as a whole to be true.

Truth table

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Bi-conditional ($p \leftrightarrow q$):

The bi-conditional proposition, represented as ($p \leftrightarrow q$), is a statement that is true when both the premise (p) and the conclusion (q) have the same truth value, and it is false when their truth values differ. In simpler terms, it means that the bi-conditional is true if and only if both the initial condition (p) and the resulting condition (q) are either both true or both false.

Truth table

p	q	$P \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

3. Demonstration:

3.1.Source Code:

```
/C program to illustrate implication and bi-conditional
#include<stdio.h>
#include<stdbool.h>
bool implication(bool a, bool b){
    return !a||b;
}
bool biconditional(bool a, bool b){
    return (!a||b)&&(!b||a);
}
void display(bool a){
    if(a==true)
    {
        printf("T\t");
    }
}
```

```

        else
        {
            printf("F\t");
        }
    }
    int main()
    {
        int i;
        bool res;
        bool a[]={true,true,false,false};
        bool b[]={true,false,true,false};
        printf("p\tq\tp->q\tp<->q\n");
        printf("_____ \n");
        for(i=0;i<4;i++)
        {
            display(a[i]);
            display(b[i]);
            res=implication(a[i],b[i]);
            display(res);

            res=biconditional(a[i],b[i]);
            display(res);

            printf("\n");
        }
    }
    return 0;
}

```

3.2. Output and Discussion;

p	q	p->q	p<->q
T	T	T	T
T	F	F	F
F	T	T	F
F	F	T	T

We have presented a truth table that demonstrates the behavior of implication and bi-conditional using logical operators such as AND (&&), OR (||), and NOT (!).

4. Conclusion:

Therefore, we can depict the implication and bi-conditional of propositions using a truth table in the C programming language.

Experiment 1.3

To generate TT of given proposition $[(p \rightarrow q) \wedge (q \rightarrow r)] \rightarrow (p \rightarrow r)$.

1. Objective:

- To generate the truth table of $[p \rightarrow q \wedge q \rightarrow r] \rightarrow (p \rightarrow r)$.

2. Theory:

The given proposition is $[(p \rightarrow q) \wedge (q \rightarrow r)] \rightarrow (p \rightarrow r)$.

To create a truth table for this proposition, we need to break it down into individual parts and solve them separately.

First, we solve $(p \rightarrow q)$, then $(q \rightarrow r)$, and find the conjunction of $(p \rightarrow q)$ and $(q \rightarrow r)$. Next, we take the result of this conjunction and imply it with the outcome of $(p \rightarrow r)$.

To solve these propositional components, we can employ various logical operators, such as $\&\&$ for conjunction, $\|$ for disjunction, and $!$ for negation.

3. Demonstration:

3.1. Source Code:

```
//C program to display truth table of  $[p \rightarrow q \wedge q \rightarrow r] \rightarrow (p \rightarrow r)$ 
#include<stdio.h>
#include<stdbool.h>
void display(bool a){
    if(a==true)
    {
        printf("T\t");
    }
    else
    {
        printf("F\t");
    }
}
int main()
{
    int i;
    bool x,y,res;
    bool a[]={true,true,true,true,false,false,false,false};
    bool b[]={true,true,false,false,true,false,true,false};
    bool c[]={true,false,true,false,true,false,true,false};
    printf("\np\tq\t r\t p->q\t q->r\t (p->q)^(q->r)\tp->r\t [p->q^q->r-->p->r]\n");
    printf("\n");
    for(i=0;i<8;i++)
    {
        display(a[i]);
        display(b[i]);
        display(c[i]);
        display(!a[i]||b[i]);
        display(!b[i]||c[i]);
        printf("\t");
        x=!a[i]||b[i]&&!b[i]||c[i];
```

```

        display(x);
        display(!a[i]||c[i]);
        printf("\t");
        display(!x||!a[i]||c[i]);
        printf("\n");
    }
    return 0;
}

```

3.2. Output and Discussion:

p	q	r	$p \rightarrow q$	$q \rightarrow r$	$(p \rightarrow q) \wedge (q \rightarrow r)$	$p \rightarrow r$	$[p \rightarrow q \wedge q \rightarrow r \rightarrow p \rightarrow r]$
T	T	T	T	T	T	T	T
T	T	F	T	F	F	F	T
T	F	T	F	T	T	T	T
T	F	F	F	T	F	F	T
F	T	T	T	T	T	T	T
F	F	F	T	T	T	T	T
F	T	T	T	T	T	T	T
F	F	F	T	T	T	T	T

We have constructed a truth table for the provided proposition by dividing it into distinct parts and utilizing a variety of logical operators in the C programming language.

4. Conclusion:

Therefore, we can create the truth table for the given proposition by employing multiple logical operators in the C programming language.

Experiment 1.4

To check whether the given proposition is tautology, contradiction or contingency.

1. Objective:

- To check whether the given proposition is tautology, contradiction or contingency.

2. Theory:

To determine whether the given proposition $[(p \rightarrow q) \wedge (q \rightarrow r)] \rightarrow (p \rightarrow r)$ is a tautology, contradiction, or contingency, we need to understand the definitions of these terms.

Tautology:

A tautology is a complex statement that is universally true, regardless of the truth values of its individual atomic components. Example $p \vee \neg p$.

Truth table

p	$\neg p$	$p \vee \neg p$
T	F	T
F	T	T

Contradiction:

A contradiction is a compound statement that is perpetually false, irrespective of the truth values assigned to its constituent atomic propositions.

Example:

$p \wedge \neg p$

Truth table

p	$\neg p$	$p \wedge \neg p$
T	F	F
F	T	F

Contingency:

A compound proposition that is neither a tautology nor a contradiction is called a contingency.

Example:

$\neg p \wedge q$

Truth table

p	q	$\neg p$	$\neg q$	$\neg p \wedge \neg q$
T	T	F	F	F
T	F	F	T	F
F	T	T	F	F
F	F	T	T	T

3. Demonstration:

Source Code:

//C program to display truth table of $[p \rightarrow q \wedge q \rightarrow r] \rightarrow (p \rightarrow r)$ and to check whether the given program is tautology, contradiction or contingency.

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

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

```
void display(bool a){
```

```
    if(a==true)
```

```

        {
            printf("T\t");
        }
        else
        {
            printf("F\t");
        }
    }
}
int main()
{
    int i,count=0;
    bool x,y,res[8];
    bool a[]={true,true,true,true,false,false,false,false};
    bool b[]={true,true,false,false,true,false,true,false};
    bool c[]={true,false,true,false,true,false,true,false};
    printf("\np\tq\t r\t p->q\t q->r\t (p->q)^(q->r)\tp->r\t [p->q^q->r-->p->r]\n");
    printf("\n");
    for(i=0;i<8;i++)
    {
        display(a[i]);
        display(b[i]);
        display(c[i]);
        display(!a[i]||b[i]);
        display(!b[i]||c[i]);
        printf("\t");
        x=!a[i]||b[i]&&!b[i]||c[i];
        display(x);
        display(!a[i]||c[i]);
        printf("\t");
        display(!x||!a[i]||c[i]);
        printf("\n");
        res[i]=!x||!a[i]||c[i];
    }
    for(i=0;i<8;i++)
    {
        if(res[i]==true)
            count++;
    }
    if(count==8)
    {
        printf("\nThe given proposition is tautology");
    }
    else if(count==0)
    {
        printf("\nThe given proposition is contradiction");
    }
    else

```

```

    {
        printf("\nThe given proposition is contingency");
    }
return 0;
}

```

3.2. Output and Discussion:

p	q	r	p→q	q→r	(p→q)^(q→r)	p→r	[p→q^q→r-->p→r]
T	T	T	T	T	T	T	T
T	T	F	T	F	F	F	T
T	F	T	F	T	T	T	T
T	F	F	F	T	F	F	T
F	T	T	T	T	T	T	T
F	F	F	T	T	T	T	T
F	T	T	T	T	T	T	T
F	F	F	T	T	T	T	T

The given proposition is tautology

Here, first of all the proposition is stored in array and checked whether the given proposition is tautology, contradiction or contingency.

4. Conclusion:

Hence, we have checked whether the given implication is tautology, contradiction or contingency

Experiment 2.1

To implement Euclidean Algorithm

1. Objectives:

- Understand the Euclidean Algorithm and its application in finding the greatest common divisor (GCD) of two integers.
- Implement the Euclidean Algorithm in the C programming language.

2. Theory:

The Euclidean Algorithm is a fundamental algorithm in number theory used to find the greatest common divisor (GCD) of two integers. The GCD of two integers is the largest positive integer that divides both numbers without leaving a remainder.

The algorithm works as follows:

- Start with two positive integers, a and b, where a is greater than or equal to b.
- Divide a by b and find the remainder. Let r be the remainder: $r = a \% b$.
- Replace a with b and b with r: $a = b$, $b = r$.
- Repeat steps 2 and 3 until the remainder r becomes zero.
- The last non-zero remainder obtained in step 2 is the GCD of the original two integers a and b.

3. Demonstration:

3.1. Source Code:

```
#include <stdio.h>

// Function to find the GCD using the Euclidean Algorithm
int findGCD(int a, int b) {
    int r;
    while (b != 0) {
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}

int main() {
    int num1, num2;

    // Input two integers from the user
    printf("Enter the first integer: ");
    scanf("%d", &num1);
    printf("Enter the second integer: ");
    scanf("%d", &num2);

    // Find and display the GCD
    int gcd = findGCD(num1, num2);
    printf("The GCD of %d and %d is %d\n", num1, num2, gcd);
}
```

```
    return 0;  
}
```

3.2. Output and Discussion:

```
Enter the first integer: 421  
Enter the second integer: 111  
The GCD of 421 and 111 is 1  
-----
```

We implemented the Euclidean Algorithm in C to find the GCD of two integers. The algorithm proved to be effective in its task, producing correct results. It iteratively applied the steps of division and replacement until the remainder became zero, ensuring that the final non-zero remainder was the GCD. The Euclidean Algorithm is highly efficient and well-suited for finding the GCD of large integers because it relies on simple arithmetic operations.

4. Conclusion:

The Euclidean Algorithm is a powerful tool for finding the greatest common divisor of two integers. This lab allowed us to implement and test the algorithm in C successfully. Its simplicity and efficiency make it an essential tool in number theory and various mathematical and computational applications.

Experiment 2.2

To implement extended Euclidean Algorithm

1. Objectives:

- Understand the Extended Euclidean Algorithm and its application in finding the greatest common divisor (GCD) of two integers and their Bézout coefficients.
- Implement the Extended Euclidean Algorithm in the C programming language.

2. Theory:

The Extended Euclidean Algorithm is an extension of the Euclidean Algorithm, and it is used to find not only the GCD of two integers but also the Bézout coefficients. The Bézout coefficients are integers x and y such that $ax + by = \gcd(a, b)$.

The algorithm works as follows:

- Start with two positive integers, a and b , where a is greater than or equal to b .
- Apply the Euclidean Algorithm to find the GCD of a and b .
- Perform backward substitution to find x and y , such that $ax + by = \gcd(a, b)$.
- If $b=0$, then set $d=a$, $x=1$, $y=0$, and return (d,x,y) .
- Set $x_2=1$, $x_1=0$, $y_2=0$, $y_1=1$.
- While $b>0$, do
 - $q=\text{floor}(a/b)$, $r=a-qb$, $x=x_2-qx_1$, $y=y_2-qy_1$.
 - $a=b$, $b=r$, $x_2=x_1$, $x_1=x$, $y_2=y_1$, $y_1=y$.
- Set $d=a$, $x=x_2$, $y=y_2$ and return (d,x,y) .

3. Demonstration:

3.1.Source Code:

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

```
// Function to find the GCD using the Euclidean Algorithm
```

```
int findGCD(int a, int b, int *x, int *y) {
```

```
    if (b == 0) {
```

```
        *x = 1;
```

```
        *y = 0;
```

```
        return a;
```

```
    }
```

```
    int x1, y1;
```

```
    int gcd = findGCD(b, a % b, &x1, &y1);
```

```
    *x = y1;
```

```
    *y = x1 - (a / b) * y1;
```

```
    return gcd;
```

```
}
```

```
int main() {
```

```
    int num1, num2, x, y;
```



```

// Input two integers from the user
printf("Enter the first integer: ");
scanf("%d", &num1);
printf("Enter the second integer: ");
scanf("%d", &num2);

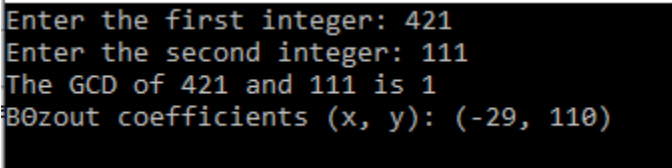
// Find the GCD and Bézout coefficients
int gcd = findGCD(num1, num2, &x, &y);

// Display the results
printf("The GCD of %d and %d is %d\n", num1, num2, gcd);
printf("Bézout coefficients (x, y): (%d, %d)\n", x, y);

return 0;
}

```

3.2. Output and Discussion:



```

Enter the first integer: 421
Enter the second integer: 111
The GCD of 421 and 111 is 1
Bézout coefficients (x, y): (-29, 110)

```

In this lab, we implemented the Extended Euclidean Algorithm in C to find the GCD of two integers and their Bézout coefficients. The algorithm proved to be effective, producing both the GCD and the Bézout coefficients correctly.

The Extended Euclidean Algorithm is useful in various mathematical and cryptographic applications, including modular inverses and solving linear Diophantine equations.

4. Conclusion:

The Extended Euclidean Algorithm is a powerful tool for finding the GCD of two integers and their Bézout coefficients. This lab allowed us to implement and test the algorithm in C successfully. Its ability to provide not only the GCD but also the Bézout coefficients makes it valuable in a wide range of mathematical and computational scenarios.

Experiment 2.3

To implement Addition and Multiplication Algorithm of two binary numbers.

1. Objectives:

- Understand the binary representation of numbers and the principles of binary addition and multiplication.
- Implement binary addition and multiplication algorithms in the C programming language.

2. Theory:

Binary addition and multiplication are fundamental operations in computer systems. Binary numbers consist of only two digits, 0 and 1, and follow similar rules as decimal numbers.

Binary Addition:

Binary addition is carried out just like decimal addition, with the primary difference being that it involves only two digits. The rules for binary addition are as follows:

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0$ (with a carry of 1)

Algorithm works as follow;

Procedure add(a,b: positive integers)

{the binary expansions of a and b are $(a_{n-1}. a_{n-2} \dots a_1.a_0)_2$ and $(b_{n-1}. b_{n-2} \dots b_1.b_0)_2$ respectively}

c=0

for j=0 to n-1

begin

d=/(a_j+b_j+c)/2/

s=a_j+b_j+c-2d

c=d

end

s_n=c

{the binary expansion of the sum is $(s_n. s_{n-1} \dots s_0)_2$ }

Binary Multiplication:

Binary multiplication involves shifting and adding operations, similar to decimal multiplication. The rules for binary multiplication are as follows:

- $0 \times 0 = 0$
- $0 \times 1 = 0$
- $1 \times 0 = 0$
- $1 \times 1 = 1$

Algorithm works as follow;

Procedure multiply(a,b: positive integers)

{the binary expansions of a and b are $(a_{n-1}. a_{n-2} \dots a_1.a_0)_2$ and $(b_{n-1}. b_{n-2} \dots b_1.b_0)_2$ respectively}

c=0

for j=0 to n-1

begin

```

if  $b_j=1$  then  $c_j=-a$  shifted  $j$  places
else  $c_j=0$ 
and
( $c_0c_1\dots\dots, c_{n-1}$  are the partial product)
 $P=0$ 
For  $j=0$  to  $n-1$ 
 $P=P+c_j$ 

```

3. Demonstration

3.1. Source Code:

```

// C program to add to binary numbers:
#include <stdio.h>
int main()
{
    long a,b;
    int i=0, c=0, sum[20];
    printf("Enter first binary number: ");
    scanf("%ld",&a);
    printf("Enter second binary number: ");
    scanf("%ld",&b);
    while (a!=0 || b!=0)
    {
        sum[i++]= (a%10 + b%10 + c)%2;
        c=(a%10 + b%10 + c)/2;
        a=a/10;
        b=b/10;
    }
    if (c!=0)
    {
        sum[i++]=c;
        --i;
    }
    printf("Sum of two binary numbers: ");
    while (i>=0)
    {
        printf("%d",sum[i--]);
    }
    return 0;
}

```

3.2. Program to find product of two binary numbers:

Source Code:

```

#include<stdio.h>
int binaryproduct(int, int);
int main()
{
    long a,b,p=0;

```

```

    int digit, factor=1;
    printf("Enter the first binary number: ");
    scanf("%ld",&a);
    printf("Enter the second binary number: ");
    scanf("%ld",&b);
    while (b!=0)
    {
        digit=b%10;
        if (digit==1)
        {
            a=a*factor;
            p=binaryproduct(a,p);
        }
        else
            a=a*factor;
        b=b/10;
        factor=10;
    }
    printf("Product of two binary numbers: %ld",p);
    return 0;
}

int binaryproduct(int a, int b)
{
    int i=0,c=0,sum[20];
    int bp=0;
    while (a!=0 || b!=0)
    {
        sum[i++]=(a%10 + b%10 + c) % 2;
        c=(a%10 + b%10 + c)/2;
        a=a/10;
        b=b/10;
    }
    if (c!=0)
    {
        sum[i++]=c;
        --i;
    }
    while (i>=0)
    {
        bp=bp*10 + sum[i--];
        return bp;
    }
}

```

3.3. Output and Discussion:

Addition of two binary numbers:

```
Enter first binary number: 1010
Enter second binary number: 1011
Sum of two binary numbers: 10101
PS C:\Users\keerushar\Desktop\vs code>
```

Multiplication of two binary numbers:

```
Enter first binary number: 101
Enter second binary number: 10
Product is 10100
```

We implemented binary addition and multiplication algorithms in C. Both algorithms were effective and produced correct results. Binary addition utilized bitwise operations to carry out the addition of binary numbers efficiently, while binary multiplication involved shifting and adding operations to perform multiplication.

4. Conclusion:

Binary addition and multiplication are fundamental operations in computer systems. This lab allowed us to implement and test these algorithms in C successfully. The efficiency and simplicity of binary operations make them essential in various computational applications and computer hardware design.

Experiment 2.4

To find Boolean join and Boolean product of Boolean Matrix.

1. Objectives:

- Understand the concepts of Boolean join and Boolean product of Boolean matrices.
- Implement algorithms to find the Boolean join and Boolean product of Boolean matrices in the C programming language.

2. Theory:

Boolean matrices consist of elements that can only take on two values, typically denoted as 0 (False) and 1 (True). Boolean join and Boolean product are operations performed on Boolean matrices.

Boolean Join (Logical OR):

The Boolean join of two matrices involves taking the logical OR operation between corresponding elements of the matrices.

The result is a new matrix where each element is the logical OR of the corresponding elements from the original matrices.

Boolean Product (Logical AND):

The Boolean product of two matrices involves taking the logical AND operation between corresponding elements of the matrices.

The result is a new matrix where each element is the logical AND of the corresponding elements from the original matrices.

3. Demonstration:

3.1. Source Code

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

```
// Function to perform Boolean join (logical OR) of two matrices
```

```
void booleanJoin(int m, int n, int A[][n], int B[][n], int result[][n]) {  
    for (int i = 0; i < m; i++) {  
        for (int j = 0; j < n; j++) {  
            result[i][j] = A[i][j] || B[i][j];  
        }  
    }  
}
```

```
// Function to perform Boolean product (logical AND) of two matrices
```

```
void booleanProduct(int m, int n, int A[][n], int B[][n], int result[][n]) {  
    for (int i = 0; i < m; i++) {  
        for (int j = 0; j < n; j++) {  
            result[i][j] = A[i][j] && B[i][j];  
        }  
    }  
}
```

```
// Function to display a Boolean matrix
```

```
void displayMatrix(int m, int n, int matrix[][n]) {
```

```

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int m, n;

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

    int matrixA[m][n], matrixB[m][n], resultJoin[m][n], resultProduct[m][n];

    printf("Enter the elements of Matrix A (0 or 1):\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &matrixA[i][j]);
        }
    }

    printf("Enter the elements of Matrix B (0 or 1):\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &matrixB[i][j]);
        }
    }

    // Calculate Boolean join and Boolean product
    booleanJoin(m, n, matrixA, matrixB, resultJoin);
    booleanProduct(m, n, matrixA, matrixB, resultProduct);

    printf("\nBoolean Join (A OR B):\n");
    displayMatrix(m, n, resultJoin);

    printf("\nBoolean Product (A AND B):\n");
    displayMatrix(m, n, resultProduct);

    return 0;
}

```

3.2.Output and Discussion:

```
Enter the number of rows: 2
Enter the number of columns: 3
Enter the elements of Matrix A (0 or 1):
1 0 1
0 1 0
Enter the elements of Matrix B (0 or 1):
0 1 0
1 0 1

Boolean Join (A OR B):
1 1 1
1 1 1

Boolean Product (A AND B):
0 0 0
0 0 0
PS C:\Users\keerushar\Desktop\vs code>
```

We implemented algorithms to find the Boolean join (logical OR) and Boolean product (logical AND) of Boolean matrices. These operations are fundamental in various fields, including computer science, digital circuit design, and database management.

Boolean Join (OR): The Boolean join operation combines corresponding elements of two matrices using the logical OR operation. It is commonly used to combine conditions in Boolean logic.

Boolean Product (AND): The Boolean product operation combines corresponding elements of two matrices using the logical AND operation. It is used to find common elements or conditions between matrices.

Boolean matrices and these operations have applications in database querying, image processing, and circuit design, among others.

4. Conclusion:

In this lab, we successfully implemented algorithms to find the Boolean join and Boolean product of Boolean matrices. These operations are valuable in various fields where binary logic is employed.

Experiment 3.1

To solve various Recursive Problems

1. Objectives:

- Sum of first n natural numbers.
- Factorial of n.
- Value of $1^2+2^2+\dots+n^2$.
- N^{th} Fibonacci number.
- Power (a,b).

2. Theory:

Recursion is a fundamental concept in computer science and mathematics, where a problem is solved by breaking it down into smaller, often identical, sub problems. It is a powerful and elegant problem-solving technique that is widely used in programming and algorithm design. The key idea behind recursion is the use of self-reference, where a function calls itself to solve a problem.

Base Case:

Every recursive algorithm must have a base case. The base case defines the simplest or smallest possible instance of the problem that can be solved directly without further recursion.

Without a base case, the recursion will continue indefinitely, leading to a stack overflow or infinite loop.

Recursive Case:

The recursive case defines how to break down a larger problem into one or more smaller, similar sub problems.

It typically involves making one or more recursive calls to the same function with modified inputs.

3. Demonstration

3.1. Source Code:

3.1.1. To find sum of first n natural numbers:

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

```
// Recursive function to find the sum of the first n natural numbers
```

```
int sum(int n) {
```

```
    // Base case: If n is 0, the sum is 0
```

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

```
        return 0;
```

```
    }
```

```
    // Recursive case: Sum the current number (n) with the sum of the first (n-1) natural numbers
```

```
    else {
```

```
        return n + sum(n - 1);
```

```
    }
```

```
}
```

```
int main() {
```

```
    int n;
```

```
    printf("Enter a positive integer n: ");
```

```
    scanf("%d", &n);
```

```
    int res = sum(n);
```

```

        printf("Sum of the first %d natural numbers = %d\n", n, res);
    return 0;
}

```

3.1.2. To find factorial of n.

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

```
// Recursive function to calculate the factorial of a number
```

```

long long factorial(int n) {
    // Base case: factorial of 0 is 1
    if (n == 0) {
        return 1;
    }
    // Recursive case: n! = n * (n-1)!
    else {
        return (long long)n * factorial(n - 1);
    }
}

```

```

int main() {
    int num;

```

```

    printf("Enter a non-negative integer: ");
    scanf("%d", &num);

```

```

    if (num < 0) {
        printf("Factorial is undefined for negative numbers.\n");
    } else {
        long long result = factorial(num);
        printf("%d! = %lld\n", num, result);
    }

```

```

    return 0;
}

```

3.1.3. Value of $1^2+2^2+\dots+n^2$.

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

```
// Recursive function to calculate the sum of squares from 1 to n
```

```

long long sumOfSquares(int n) {
    // Base case: If n is 1, return 1^2
    if (n == 1) {
        return 1;
    }
    // Recursive case: Sum of squares from 1 to n = n^2 + sum of squares from 1 to (n-1)
    else {
        return (long long)n * n + sumOfSquares(n - 1);
    }
}

```

```

}

int main() {
    int num;

    printf("Enter a positive integer n: ");
    scanf("%d", &num);

    if (num < 1) {
        printf("Please enter a positive integer.\n");
    } else {
        long long result = sumOfSquares(num);
        printf("Sum of squares from 1 to %d = %lld\n", num, result);
    }

    return 0;
}

```

3.1.4. Program to find Nth Fibonacci Number:

```

#include <stdio.h>

// Recursive function to calculate the nth Fibonacci number
long long fibonacci(int n) {
    // Base cases: Fibonacci of 0 is 0, Fibonacci of 1 is 1
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        // Recursive case: Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

int main() {
    int num;
    printf("Enter a non-negative integer n: ");
    scanf("%d", &num);
    if (num < 0) {
        printf("Fibonacci is undefined for negative numbers.\n");
    } else {
        long long result = fibonacci(num);
        printf("Fibonacci(%d) = %llu\n", num, result);
    }
    return 0;
}

```

3.1.5. Program to find Power(a,b):

```
#include <stdio.h>

// Recursive function to calculate power for positive numbers
double power(double a, int b) {
    // Base case: Any number raised to the power of 0 is 1
    if (b == 0) {
        return 1.0;
    } else {
        // Recursive case:  $a^b = a * a^{(b-1)}$ 
        return a * power(a, b - 1);
    }
}

int main() {
    double base;
    int exponent;
    printf("Enter the base (a): ");
    scanf("%lf", &base);
    printf("Enter the positive exponent (b): ");
    scanf("%d", &exponent);
    double result = power(base, exponent);
    printf("%.2lf^%d = %.2lf\n", base, exponent, result);
    return 0;
}
```

3.2. Outputs and Discussion:

3.2.1. Sum of first N natural numbers:

```
Enter a positive integer n: 10
Sum of the first 10 natural numbers = 55
PS C:\Users\keerushar\Desktop\vs code>
```

3.2.2. Factorial of N.

```
Enter a non-negative integer: 5
5! = 120
PS C:\Users\keerushar\Desktop\vs code>
```

3.2.3. Value of $1^2+2^2+.....+n^2$.

```
Enter a positive integer n: 10
Sum of squares from 1 to 10 = 385
PS C:\Users\keerushar\Desktop\vs code>
```

3.2.4. Nth Fibonacci number

```
Enter a non-negative integer n: 6
Fibonacci(6) = 8
PS C:\Users\keerushar\Desktop\vs code>
```

3.2.5. Power(a,b):

```
Enter the base (a): 2
Enter the positive exponent (b): 5
2.00^5 = 32.00
PS C:\Users\keerushar\Desktop\vs code> |
```

3.3.Discussion:

Here for every problem we have write different codes and they have use recursion to solve the problem. Program 1 find the sum of first n natural numbers using recursion, program 2 factorial of n , program 3 sum of square of first n natural numbers, program 4 Fibonacci series and finally last program find the power(a,b) using recursion respectively.

4. Conclusion:

I've successfully solved the given problems by implementing recursive solutions. I've also displayed the program outputs within the Visual Studio Code editor.

Experiment 3.2

To solve TOH problem.

1. Objectives:

- To study and write a program to solve the TOH problem.

2. Theory:

The Tower of Hanoi problem consists of three pegs (A, B, and C) and a number of disks of different sizes, initially stacked in decreasing size on one peg (A). The objective is to move all the disks from peg A to peg C, adhering to the following rules:

- Only one disk can be moved at a time.
- A larger disk cannot be placed on top of a smaller disk.
- Temporary peg (B) can be used as an intermediate step.

Algorithm

The iterative algorithm to solve the Tower of Hanoi problem involves simulating the recursive approach using a stack data structure. Here are the steps:

1. Initialize a stack to keep track of the state.
2. Push the initial state onto the stack, indicating the number of disks and source, auxiliary, and target pegs.
3. Repeat until the stack is empty:
 - Pop the top state from the stack.
 - If the number of disks is 1, move the disk from the source peg to the target peg and continue.
 - Otherwise, push three new states onto the stack in reverse order, simulating the recursive calls with adjusted parameters:
 - 3.1. Move n-1 disks from peg A to peg B, using peg C as the auxiliary peg.
 - 3.2. Move the largest disk from peg A to peg C.
 - 3.3. Move n-1 disks from peg B to peg C, using peg A as the auxiliary peg.

3. Demonstration:

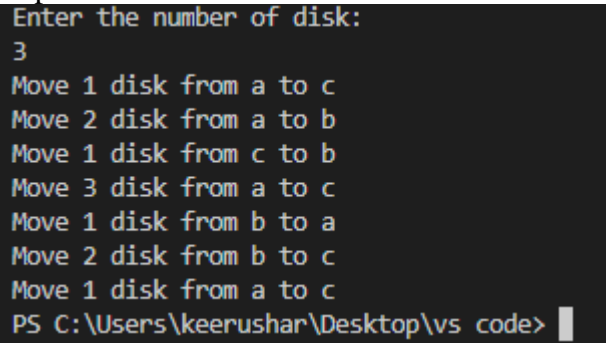
3.1. Source Code:

C program to solve TOH problem

```
#include<stdio.h>
void TOH(int n,char a, char b, char c){
if(n>0)
{
    TOH(n-1,a,c,b);
    printf("Move %d disk from %c to %c\n",n,a,c);
    TOH(n-1,b,a,c);
}
}
int main()
{
    int n;
    char a='a',b='b',c='c';
```

```
    printf("Enter the number of disk:\n");
    scanf("%d",&n);
    TOH(n,a,b,c);
    return 0;
}
```

3.2. Output and Discussion:



```
Enter the number of disk:
3
Move 1 disk from a to c
Move 2 disk from a to b
Move 1 disk from c to b
Move 3 disk from a to c
Move 1 disk from b to a
Move 2 disk from b to c
Move 1 disk from a to c
PS C:\Users\keerushar\Desktop\vs code>
```

The C program successfully solves the Tower of Hanoi problem using a recursive approach. It follows the algorithm described earlier, moving disks step by step while adhering to the rules. The output demonstrates the sequence of moves required to solve the problem for a given number of disks.

The time complexity of this algorithm is (2^n) , where n is the number of disks. This exponential time complexity is due to the recursive nature of the algorithm.

4. Conclusion:

In this lab, we implemented a C program to solve the Tower of Hanoi problem using recursion. The program effectively moves disks from one peg to another while obeying the specified rules.

Experiment 3.3

To implement Merge Sort Algorithm.

1. Objectives:

- To implement merge sort algorithm.
- To implement algorithm in C programming.

2. Theory:

Merge sort is a sorting technique utilized to arrange arrays by following a sequence of steps. It starts by breaking the array into smaller subarrays, sorting each of these subarrays independently, and then merging them back together to construct the final sorted array.

This method is especially efficient when sorting larger arrays in comparison to more basic sorting methods like bubble sort or insertion sort. Importantly, merge sort maintains the order of elements with equal values, ensuring that their relative positions are preserved during the sorting process.

The algorithm functions by repeatedly dividing the array into halves until it's no longer possible to divide further. In other words, it halts when the array becomes empty or contains just a single element. For arrays with multiple elements, it splits them into halves and proceeds to apply the merge sort recursively to each of these halves. Once both halves are sorted independently, the merge operation is executed. This merge operation entails combining two smaller sorted arrays into one larger, sorted array.

ALGORITHM FOR MERGE SORT:

step1: Start.

step2: declare array and left, right, mid-value.

step3: perform merge functions:

if left < right

return

mid = (left + right) / 2;

mergesort(array, left, mid)

mergesort(array, mid + 1, right)

mergesort(array, left, mid, right)

step4: Stop.

3. Demonstration:

Source Code:

```
#include <stdio.h>
void Merge(int a[], int L, int mid, int H)
{
    int n1 = mid - L + 1;
    int n2 = H - mid;
    int i, j, k;
    int A[n1], B[n2];
    for ( i = 0; i < n1; i++)
        A[i] = a[L + i];
    for ( j = 0; j < n2; j++)
```



```

    B[j] = a[mid + 1 + j];
    i = 0;
    j = 0;
    k = L;
    while (i < n1 && j < n2) {
        if (A[i] <= B[j]) {
            a[k] = A[i];
            i++;
        } else {
            a[k] = B[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        a[k] = A[i];
        i++;
        k++;
    }
    while (j < n2) {
        a[k] = B[j];
        j++;
        k++;
    }
}

void MergeSort(int a[],int L,int H) {
    if(L<H)
    {
        int mid=(L+H)/2;
        MergeSort(a,L,mid);
        MergeSort(a,mid+1,H);
        Merge(a,L,mid,H);
    }
}

int main() {
    int n,Array[100],i=0;
    printf("Enter no. of elements in array:\n");
    scanf("%d",&n);
    printf("Enter elements of Array:\n");
    for(i=0;i<n;i++)
        scanf("%d",&Array[i]);
    printf("Array Without Sort:\n");
    for(i=0;i<n;i++)
        printf("%d\t",Array[i]);
    MergeSort(Array,0,n-1);
    printf("\nArray with Sorting:\n");
    for(int i=0;i<n;i++)
        printf("%d\t",Array[i]);

```

```
    return 0;  
}
```

3.2. Output and Discussion:

```
Enter no. of elements in array:  
10  
Enter elements of Array:  
1 27 8 16 25 49 81 64 4 9  
Array Without Sort:  
1      27      8      16      25      49      81      64      4      9  
Array with Sorting:  
1      4      8      9      16      25      27      49      64      81  
PS C:\Users\keerushar\Desktop\vs code>
```

Here, we have define functions for merging and performing merge sort on an array of integers. It then sorts an example array and prints both the original and sorted arrays.

4. Conclusion:

To sum up, merge sort algorithm was successfully studied and implemented in C programming to sort number of elements stored in array.