|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SEmester project** | | | | |
|  | | | | |
| Discrete Structure | | | | |
| **Section SE-A**  **Imama Kainat 22F3661**  **Zainab Eman 22F3738** | | | | |
|  |  |  |  |  |

**Question 2**

**//.............................for transvers>1.bfs**

**//...........................................2.dfs**

**//............................0 for blue**

**//............................1 for red**

**#include<bits/stdc++.h>//header files to use vectors, memset, and iostream**

**using namespace std;**

**const int MAXN = 100005;//maximum number of vertices**

**vector<int> adj[MAXN];// vector to store the edges of the graph**

**int color[MAXN];// store the color of each vertex in the graph**

**//recursive function->u=vertex,c=color**

**bool dfs(int u, int c){**

**color[u] = c;//assigns the color c to the vertex u**

**//traverses all its adjacent vertices v**

**//V IS THE ELEMENT ADJACENCY START SE LAST TAK SARE ADJACENCY LIST KE ELEMENTS**

**for(auto v: adj[u]){//auto itself decide datatype yaha pr int he**

**if(color[v] == -1){//color of v is not assigned**

**if(dfs(v, 1 - c) == false){//we call the function //1-1=0 OR 1-0=1**

**//recursively with color 1-c for v.**

**return false;// two adjacent vertices with the same color**

**}**

**}**

**else if(color[v] == color[u]){**

**return false;**

**}**

**}//all the adjacent vertices of u without any issues**

**return true;**

**}**

**bool isBipartite(int n){//number of vertices n as input**

**memset(color, -1, sizeof color);//color array to -1 for all vertices**

**for(int i=1; i<=n; i++){**

**if(color[i] == -1){**

**if(dfs(i, 0) == false){**

**return false;**

**}**

**}**

**}**

**return true;**

**}**

**// Print the graph**

**void printGraph( int v) {**

**cout<<"\nGIVEN GRAPH WAS\n";**

**for (int d = 1; d <= v; ++d) {**

**cout << "\n VERTEX "**

**<< d << ":";**

**for (auto x : adj[d])//used to iterate the values of vectors like loop**

**cout << "-> " << x;**

**cout<<endl;**

**}**

**}**

**int main(){**

**cout<<"...............WELCOME IN BIPARTITE CHECKING GRAPH................\n";**

**int n, m;**

**cout<<"enter vertex\n";**

**cin >> n ;**

**cout<<"enter edges\n";**

**cin >> m;**

**for(int i=0; i<m; i++){**

**cout<<"enter edge "<<"of "<<i+1<<" graph: ";**

**int u, v;**

**cin >> u >> v;**

**//For each edge, we take two integers as input representing the vertices**

**//that the edge connects. We then add this edge to the adjacency list vector**

**//by pushing the vertices to each other's adjacency list.**

**adj[u].push\_back(v);**

**adj[v].push\_back(u);**

**}**

**if(isBipartite(n)){**

**cout << "Graph is Bipartite\n";**

**}**

**else{**

**cout << "Graph is not Bipartite\n";**

**}**

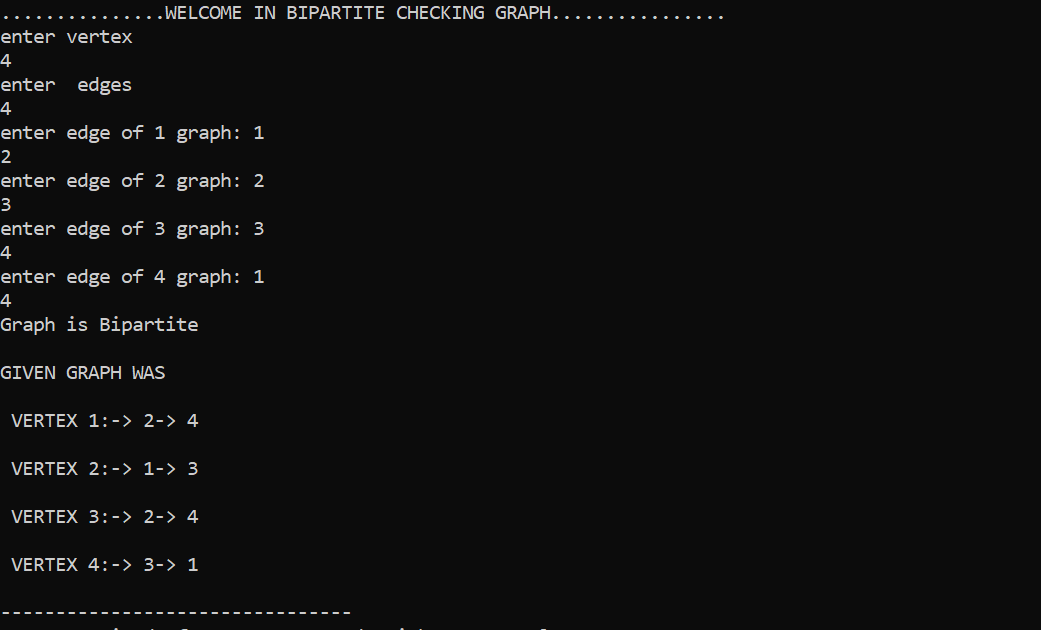
**// We then check its adjacent vertices, 2 and 4. Both are uncolored, so we call dfs(2, 1) and dfs(4, 1).**

**//4 vertices and 4 edges: (1,2), (2,3), (3,4), (4,1).**

**printGraph(n);**

**return 0;**

**}**

****

**Question 1 a**

**When enter [(p ∧ q) → r] ↔ (p → (q → r)) ≡ [(p → q) ∧ (q → r)] → (p → r)**

**#include<iostream>**

**using namespace std;**

**bool f1[26];**

**bool f2[26];**

**bool evaluate(){**

**for(int i=0;i<8;i++){**

**if(f1[i]==f2[i])**

**return false;**

**}**

**return true;**

**}**

**void equ1(){**

**cout<<endl;**

**int w=0;**

**cout<<"\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";**

**cout<<"\t\t\* \*\n";**

**cout<<"\t\t\* TRUTH TABLE \*\n";**

**cout<<"\t\t\* \*\n";**

**cout<<"\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";**

**cout << "p\tq\tr\t(p^q)>r\t\t(p>(q>r))\t\t[(p^q)>r]<>(p>(q> r))" << endl;**

**for (int p = 0; p <= 1; p++) {**

**for (int q = 0; q <= 1; q++) {**

**for (int r = 0; r <= 1; r++) {**

**int eq1 = ((p && q) ? r : true);//(p^q)>r**

**int eq2 = (((!p) || q) ? r : true);//(p>(q>r))**

**int eq3 = ((!p || q) && (!q || r)) ? true : false;//[(p^q)>r]<>(p>(q> r))**

**cout << p << "\t" << q << "\t" << r << "\t" << eq1 << "\t\t" << eq2 << "\t\t\t" << eq3 << endl;**

**f1[w]=eq3;**

**w++;**

**}**

**}**

**}**

**}**

**void equ2(){**

**bool f2[26];**

**int w1=0;**

**cout << "p|\tq|\tr|\t(p>q)|\t\t(q>r)|\t\t(p>q)&(q>r)|\t\t(p>r)|\t\t[(p>q)&(q>r)]>(p>r)|" << endl;**

**cout<<"\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"<<endl;**

**for (int p = 0; p <= 1; p++) {**

**for (int q = 0; q <= 1; q++) {**

**for (int r = 0; r <= 1; r++) {**

**int eq1 = ((!p) || q) ? true : false;//(p>q**

**int eq2 = ((!q) || r) ? true : false;//(q>r)**

**int eq3 = eq1 && eq2;//p>q)&(q>r)**

**int eq4 = ((!p) || r) ? true : false;//(p>r)**

**int eq5 = ((!eq3) || eq4) ? true : false;//[(p>q)&(q>r)]>(p>r)**

**cout << p << "|\t" << q << "|\t" << r << "|\t" << eq1 << "|\t\t" << eq2 << "|\t\t" << eq3 << "|\t\t\t" << eq4 << "|\t\t\t" << eq5 << endl;**

**f2[w1]=eq5;**

**w1++;}**

**}**

**}**

**}**

**int main()**

**{**

**//[(p^q) → r] ↔ (p → (q → r))**

**cout<<"[(p^q)>r]<>(p>(q>r))=[(p>q)^(q>r)]>(p>r)\n";**

**equ1();//.....................................for left side**

**cout<<"\n\n\n";**

**equ2();//......................................for right side**

**if(evaluate()==true)**

**cout<<"BOTH EQUATION ARE EQUAIVALENT TO EACH OTHER\n";**

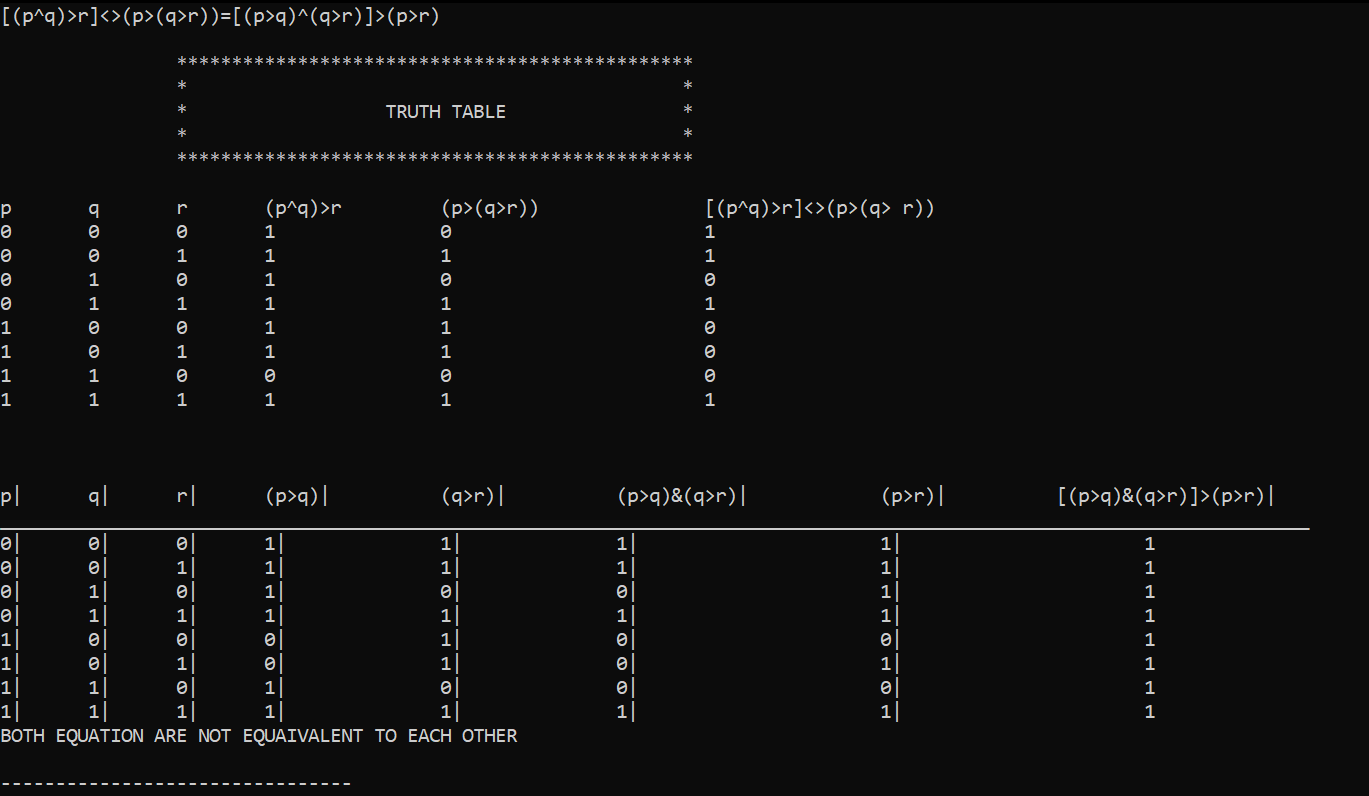
**else**

**cout<<"BOTH EQUATION ARE NOT EQUAIVALENT TO EACH OTHER\n";**

**system(pause)**

**return 0;**

**}**

****

**Generic code for buiding Truthtable**

**This is a program that evaluates logical expressions and prints their truth tables. The program takes a logical statement as input from the user, determines the number of variables in the expression, and then prints a truth table for the expression. The program also splits the expression into sub-expressions to print the truth tables for each of them.**

**The program first reads the logical statement from the user using the getline() function and stores it in the expression variable. It then uses a loop to determine the number of variables in the expression. If a variable is encountered for the first time, it is added to the variable\_names array, and the n variable is incremented. The program also splits the expression into sub-expressions using brackets, and stores them in the strings array.**

**The program then uses a loop to generate the truth table for the expression. It uses another loop to iterate over each row in the truth table, and sets the value of each variable to either true or false in turn. The evaluate() function is then called to evaluate the expression for that row of the truth table, and the result is printed to the console.**

**The evalute() function is a recursive function that takes an expression, an array of boolean variables, and the number of variables as input. It evaluates the expression recursively by splitting it into sub-expressions using brackets and operators, and evaluating each sub-expression recursively. If the sub-expression contains a variable, it looks up its value in the variables array. If the sub-expression contains an operator, it evaluates the left and right sub-expressions recursively and applies the operator. If the sub-expression contains a negation, it evaluates the sub-expression that follows the negation recursively and returns the negation of the result. The function returns the final boolean value of the expression.**

**The print\_truth\_table() function takes the expression, the number of variables, the array of variable names, and the array of boolean variables as input, and prints the truth table to the console. It uses a loop to iterate over each row in the truth table, and uses another loop to iterate over each variable in turn, setting its value to either true or false for that row. It then calls the evaluate() function to evaluate the expression for that row, and prints the result to the console.**

**The split\_expression() function takes an expression as input and splits it into sub-expressions using brackets and operators, and returns them as a vector of strings. It uses a loop to iterate over the expression, and maintains a count of the number of open brackets encountered so far. When it encounters a closing bracket that matches the last open bracket, it extracts the sub-expression enclosed by the brackets and adds it to the vector of sub-expressions.**

**Functions are:**

* **Evaluate**
* **Print truth table**
* **Split expression**

**By using vectors**

**#include <iostream>**

**#include <math.h>**

**#include <string>**

**#include <vector>**

**using namespace std;**

**bool flag=true;**

**bool evaluate(string expression, bool\* variables, int n) {//..............................................1 handle brackets,&,| >,<>,~**

**// Base case: expression contains only one variable**

**if (expression.length() == 1) {**

**int var\_index = expression[0] - 'a';**

**return variables[var\_index];**

**}**

**int num\_brackets = 0;// getting brackets**

**for (int i = 0; i < expression.length(); i++) {**

**if (expression[i] == '(') {**

**num\_brackets++;**

**} else if (expression[i] == ')') {//Bracket**

**num\_brackets--;**

**} else if (num\_brackets == 0 && (expression[i] == '&' || expression[i] == '|' || expression[i] == '>'||expression[i] == '<')) {**

**bool left\_value = evaluate(expression.substr(0, i), variables, n);**

**bool right\_value = evaluate(expression.substr(i+1), variables, n);**

**if (expression[i] == '&') {**

**return (left\_value && right\_value);**

**} else if (expression[i] == '|') {//OR**

**return (left\_value || right\_value);**

**} else if (expression[i] == '>') {//Implies**

**return (!left\_value || right\_value);**

**}**

**else if(expression[i] == '<' && expression[i+1]=='>')//Biconditional**

**return(left\_value && right\_value) || (!left\_value && !right\_value);**

**} else if (num\_brackets == 0 && expression[i] == '~') {**

**return !evaluate(expression.substr(i+1), variables, n);**

**}**

**}**

**}**

**//Print Truth Table**

**void print\_truth\_table(string expression, int n, char\* variable\_names, bool\* variables) {**

**cout << " ";**

**for (int i = 0; i < n; i++) {**

**cout << " | " << variable\_names[i];**

**}**

**cout << " | " << expression << endl;**

**for (int i = 0; i < n; i++) {**

**cout << "\_\_";**

**}**

**cout << "\_\_";**

**for (int i = 0; i < expression.length(); i++) {**

**cout << "\_\_\_";**

**}**

**cout << endl;**

**int num\_rows = pow(2, n);**

**for (int i = 0; i < num\_rows; i++) {**

**for (int j = n - 1; j >= 0; j--) {**

**bool value = (i / (int)pow(2, j)) % 2;**

**variables[j] = value;**

**cout << " | " << (value ? "T" : "F");**

**}**

**cout << " | " << (evaluate(expression, variables, n) ? "T" : "F") << endl;**

**}**

**}**

**//Split expression**

**vector<string> split\_expression(string expression) {**

**vector<string> sub\_expressions;**

**int num\_brackets = 0;**

**int start\_pos = 0;**

**for (int i = 0; i < expression.length(); i++) {**

**if (expression[i] == '(') {**

**num\_brackets++;**

**} else if (expression[i] == ')') {**

**num\_brackets--;**

**}**

**if (num\_brackets == 0 && (expression[i] == '&' || expression[i] == '|' || expression[i] == '>')) {**

**sub\_expressions.push\_back(expression.substr(start\_pos, i-start\_pos));**

**start\_pos = i+1;**

**}**

**}**

**sub\_expressions.push\_back(expression.substr(start\_pos));**

**return sub\_expressions;**

**}**

**int main() {**

**string expression;**

**cout << "Enter a logical statement: ";**

**getline(cin, expression);**

**string strings[50];**

**int stringssize = 0;**

**int num\_brackets = 0;**

**int start\_pos\_arr[100] = { 0 };**

**for (int i = 0; i < expression.size(); i++) {**

**if (expression[i] == '(') {**

**start\_pos\_arr[num\_brackets] = i;**

**num\_brackets++;**

**}**

**else if (expression[i] == ')') {**

**num\_brackets--;**

**for (int j = start\_pos\_arr[num\_brackets]; j <= i; j++)**

**{**

**strings[stringssize].push\_back(expression[j]);**

**}**

**stringssize++;**

**}**

**}**

**for (int i = 0; i<stringssize; i++)**

**{**

**cout << strings[i] << endl;**

**}**

**int n = 0;//.................................................number of variables**

**char variable\_names[26];**

**bool variables[26];**

**// Determine the number of variables in the expression**

**for (int i = 0; i < expression.size(); i++) {**

**if (expression[i] >= 'a' && expression[i] <= 'z' || expression[i] >= 'A' && expression[i] <= 'Z') {**

**if (i == 0 || expression[i - 1] != expression[i]) {**

**variable\_names[n] = expression[i];**

**n++;**

**}**

**}**

**}//non repeat premises check if repeat**

**int countvar = n, temp = false;**

**for (int i = 0; i<countvar; i++) {**

**for (int j = 0; j<countvar; j++) {**

**if (variable\_names[i] == variable\_names[j]) {**

**if (i != j) temp = true;**

**}**

**}**

**}**

**if (temp)**

**n -= 1;**

**// Print the truth table**

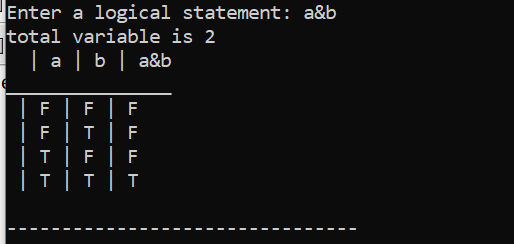
**//dont check the repeating**

**cout << "total variable is " << n << endl;**

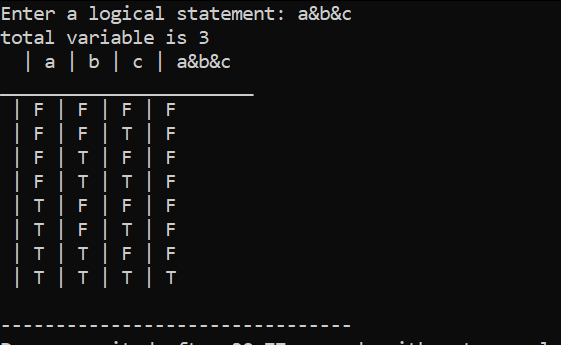
**print\_truth\_table(expression, n, variable\_names, variables);**

**}**

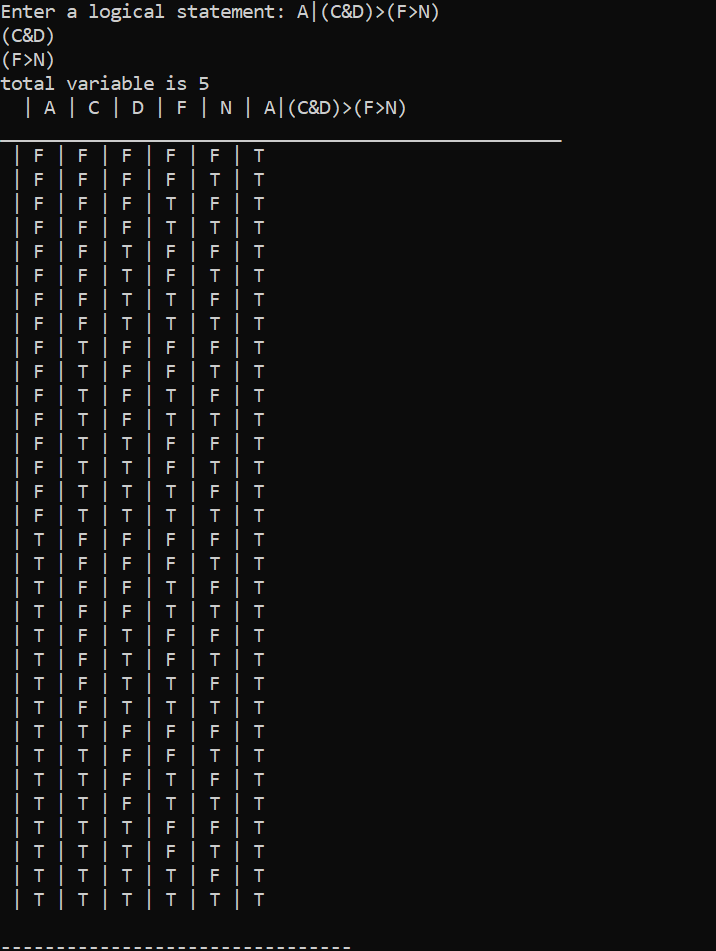
* **When we enter values of two variable**

****

**When we enter value of three variable answer still correct**

****

**But when we enter bracket statement it split the brackets**

****

**Two equations by generic code**

**#include <iostream>**

**#include <math.h>**

**#include <string>**

**#include <vector>**

**using namespace std;**

**bool evaluate(string expression, bool\* variables, int n) {**

**// Base case: expression contains only one variable**

**if (expression.length() == 1) {**

**int var\_index = expression[0] - 'a';**

**return variables[var\_index];**

**}**

**// Evaluate sub-expressions recursively**

**// Evaluate sub-expressions recursively**

**int num\_brackets = 0;**

**for (int i = 0; i < expression.length(); i++) {**

**if (expression[i] == '(') {**

**num\_brackets++;**

**} else if (expression[i] == ')') {**

**num\_brackets--;**

**} else if (num\_brackets == 0 && (expression[i] == '&' || expression[i] == '|' || expression[i] == '>')) {**

**bool left\_value = evaluate(expression.substr(0, i), variables, n);//from strt to i**

**bool right\_value = evaluate(expression.substr(i+1), variables, n);//from i+1 to end**

**if (expression[i] == '&') {**

**return (left\_value && right\_value);**

**} else if (expression[i] == '|') {**

**return (left\_value || right\_value);**

**} else if (expression[i] == '>') {**

**return (!left\_value || right\_value);**

**}**

**} else if (num\_brackets == 0 && expression[i] == '~') {**

**return !evaluate(expression.substr(i+1), variables, n);**

**}**

**}**

**}**

**void print\_truth\_table(string expression, int n, char\* variable\_names, bool\* variables) {**

**cout<<endl;**

**cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";**

**cout<<"\* \*\n";**

**cout<<"\* TRUTH TABLE \*\n";**

**cout<<"\* \*\n";**

**cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";**

**for (int i = 0; i < n; i++) {**

**cout << " | " << variable\_names[i];**

**}**

**cout << " | " << expression << endl;**

**for (int i = 0; i < n; i++) {**

**cout << "\_\_";**

**}**

**cout << "\_\_";**

**for (int i = 0; i < expression.length(); i++) {**

**cout << "\_\_\_";**

**}**

**cout << endl;**

**int num\_rows = pow(2, n);**

**for (int i = 0; i < num\_rows; i++) {**

**for (int j = n - 1; j >= 0; j--) {**

**bool value = (i / (int)pow(2, j)) % 2;**

**variables[j] = value;**

**cout << " | " << (value ? "T" : "F");**

**}**

**cout << " | " << (evaluate(expression, variables, n) ? "T" : "F") << endl;**

**}**

**}**

**vector<string> split\_expression(string expression) {**

**vector<string> sub\_expressions;**

**int num\_brackets = 0;**

**int start\_pos = 0;**

**for (int i = 0; i < expression.length(); i++) {**

**if (expression[i] == '(') {**

**num\_brackets++;**

**} else if (expression[i] == ')') {**

**num\_brackets--;**

**}**

**if (num\_brackets == 0 && (expression[i] == '&' || expression[i] == '|' || expression[i] == '>')) {**

**sub\_expressions.push\_back(expression.substr(start\_pos, i-start\_pos));**

**start\_pos = i+1;**

**}**

**}**

**sub\_expressions.push\_back(expression.substr(start\_pos));**

**return sub\_expressions;**

**}**

**void equivalent(string expression){**

**string strings[50];**

**int stringssize = 0;**

**int num\_brackets = 0;**

**int start\_pos\_arr[100] = { 0 };**

**for (int i = 0; i < expression.size(); i++) {**

**if (expression[i] == '(') {**

**start\_pos\_arr[num\_brackets] = i;**

**num\_brackets++;**

**}**

**else if (expression[i] == ')') {**

**num\_brackets--;**

**for (int j = start\_pos\_arr[num\_brackets]; j <= i; j++)**

**{**

**strings[stringssize].push\_back(expression[j]);**

**}**

**stringssize++;**

**}**

**}**

**cout<<"IF WE SPLIT THE BRACKETS\n";**

**for (int i = 0; i<stringssize; i++)**

**{**

**cout << strings[i] << endl;**

**}**

**int n = 0;//.................................................number of variables**

**char variable\_names[26];**

**bool variables[26];**

**// Determine the number of variables in the expression**

**for (int i = 0; i < expression.size(); i++) {**

**if (expression[i] >= 'a' && expression[i] <= 'z' || expression[i] >= 'A' && expression[i] <= 'Z') {**

**if (i == 0 || expression[i - 1] != expression[i]) {**

**variable\_names[n] = expression[i];**

**n++;**

**}**

**}**

**}//non repeat premises**

**cout << "total variable is " << n << endl;**

**int countvar = n, temp = false;**

**for (int i = 0; i<countvar; i++) {**

**for (int j = 0; j<countvar; j++) {**

**if (variable\_names[i] == variable\_names[j]) {**

**if (i != j) temp = true;**

**}**

**}**

**}**

**if (temp)**

**n -= 1;**

**// Print the truth table**

**//dont check the repeating**

**cout << "total variable is " << n << endl;**

**print\_truth\_table(expression, n, variable\_names, variables);**

**}**

**int main() {**

**string expression1,expression2;**

**cout << "Enter statement 1\n";**

**cin>>expression1;**

**equivalent(expression1);**

**cout << "Enter statement 2\n";**

**cin>>expression2;**

**equivalent(expression2);**

**system("pause");**

**return 0;**

**when we enter two equations **

**CODE WITHOUT USING VECTOR USING SIMPLE STRINGodeod**

**#include <iostream>**

**#include <string>**

**#include <cmath>**

**using namespace std;**

**// Function to evaluate a logical expression with no brackets**

**bool evaluate(string expression, int n, bool\* variables) {**

**// Base case: expression contains only one variable**

**//ASCII value of the character 'a' from the ASCII value of**

**//the "variable" character and then casts the result to an integer.**

**//This will give a unique integer value for each character entered by the user,**

**//starting with 0 for 'a', 1 for 'b', and so on.**

**if (expression.size() == 1) {//single variable**

**int var\_index = expression[0] - 'a';**

**return variables[var\_index];**

**}// bracket system**

**//VALUES SUBSTR('hello', 2)**

**//The result is 'ello'.**

**// Evaluate the expression using the logical operators**

**for (int i = 0; i < expression.size(); i++) {**

**if (expression[i] == '&') {**

**bool left\_value = evaluate(expression.substr(0, i), n, variables);**

**bool right\_value = evaluate(expression.substr(i + 1), n, variables);**

**return left\_value && right\_value;**

**}//or**

**else if (expression[i] == '|') {**

**bool left\_value = evaluate(expression.substr(0, i), n, variables);**

**bool right\_value = evaluate(expression.substr(i + 1), n, variables);**

**return left\_value || right\_value;**

**}//implies**

**else if (expression[i] == '>') {**

**bool left\_value = evaluate(expression.substr(0, i), n, variables);**

**bool right\_value = evaluate(expression.substr(i + 1), n, variables);**

**return !left\_value || right\_value;**

**}//negation**

**else if (expression[i] == '~') {**

**bool value = !evaluate(expression.substr(i + 1), n, variables);**

**return value;**

**}**

**//biconditional**

**else if (expression[i] == '<' && expression[i + 1] == '>') {**

**bool left\_value = evaluate(expression.substr(0, i), n, variables);**

**bool right\_value = evaluate(expression.substr(i + 2), n, variables);**

**return (left\_value && right\_value) || (!left\_value && !right\_value);**

**}**

**}**

**return false; // Should never reach here**

**}**

**//Function to split a logical expression into its bracketed parts**

**string split\_expression(string expression, int n, bool\* variables) {**

**//Find the position of the outermost pair of brackets**

**int num\_brackets = 0;**

**int start\_pos = -1;**

**for (int i = 0; i < expression.length(); i++) {**

**if (expression[i] == '(') {**

**if (num\_brackets == 0) start\_pos = i;**

**num\_brackets++;**

**}**

**else if (expression[i] == ')') {**

**num\_brackets--;**

**if (num\_brackets == 0) {**

**// Evaluate the subexpression within the brackets recursively**

**bool value = evaluate(expression.substr(start\_pos + 1, i - start\_pos - 1), n, variables);**

**// Replace the subexpression with its truth value in the expression**

**expression.replace(start\_pos, i - start\_pos + 1, value ? "1" : "0");**

**// Reset the loop index to the beginning of the expression**

**i = -1;**

**}**

**}**

**}**

**return expression;**

**}**

**// Function to print the truth table for a logical statement**

**void print\_truth\_table(string expression, int n, char\* variable\_names, bool\* variables) {**

**cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";**

**cout << "\* \*\n";**

**cout << "\* TRUTH TABLE \*\n";**

**cout << "\* \*\n";**

**cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";**

**// cout << " ";**

**for (int i = 0; i < n; i++) {**

**cout << " | " << variable\_names[i];**

**}**

**cout << " | " << expression << endl;**

**//cout << " ";**

**for (int i = 0; i < n; i++) {**

**cout << "\_\_";**

**}**

**cout << "\_\_";**

**for (int i = 0; i < expression.length(); i++) {**

**cout << "\_\_\_";**

**}**

**cout << endl;**

**//2^n formula**

**int num\_rows = pow(2, n);**

**// cout<<"total number of rows will be "<<num\_rows<<endl;**

**//variable use for aik sari row ki mtlb input ki sari values ko store kerta he jaha se next final ko solve kia gata he**

**for (int i = 0; i < num\_rows; i++) {**

**for (int j = n - 1; j >= 0; j--) {**

**bool value = (i / (int)pow(2, j)) % 2;**

**variables[j] = value;//value bool type he correct to true and galat to false**

**cout << " | " << (value ? "T" : "F");**

**}**

**cout << " | " << (evaluate(expression, n, variables) ? "T" : "F") << endl;**

**}**

**}**

**int main() {**

**// Get input from user**

**string expression;**

**cout << "Enter a logical statement: ";**

**getline(cin, expression);**

**string strings[50];**

**int stringssize = 0;**

**int num\_brackets = 0;**

**int start\_pos\_arr[100] = { 0 };**

**for (int i = 0; i < expression.size(); i++) {**

**if (expression[i] == '(') {**

**start\_pos\_arr[num\_brackets] = i;**

**num\_brackets++;**

**}**

**else if (expression[i] == ')') {**

**num\_brackets--;**

**for (int j = start\_pos\_arr[num\_brackets]; j <= i; j++)**

**{**

**strings[stringssize].push\_back(expression[j]);**

**}**

**stringssize++;**

**}**

**}**

**for (int i = 0; i<stringssize; i++)**

**{**

**cout << strings[i] << endl;**

**}**

**int n = 0;//.................................................number of variables**

**char variable\_names[26];**

**bool variables[26];**

**// Determine the number of variables in the expression**

**for (int i = 0; i < expression.size(); i++) {**

**if (expression[i] >= 'a' && expression[i] <= 'z' || expression[i] >= 'A' && expression[i] <= 'Z') {**

**if (i == 0 || expression[i - 1] != expression[i]) {**

**variable\_names[n] = expression[i];**

**n++;**

**}**

**}**

**}//non repeat premises**

**cout << "total variable is " << n << endl;**

**int countvar = n, temp = false;**

**for (int i = 0; i<countvar; i++) {**

**for (int j = 0; j<countvar; j++) {**

**if (variable\_names[i] == variable\_names[j]) {**

**if (i != j) temp = true;**

**}**

**}**

**}**

**if (temp)**

**n -= 1;**

**// Print the truth table**

**//dont check the repeating**

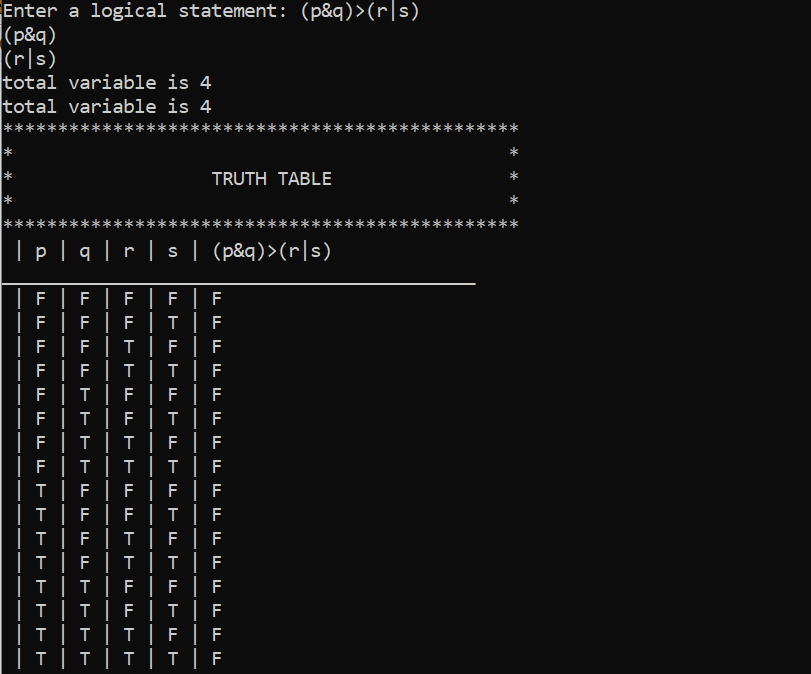
**cout << "total variable is " << n << endl;**

**print\_truth\_table(expression, n, variable\_names, variables);**

**system("pause");**

**return 0;**

**}**

****

**Question 1 b**

**#include <iostream>**

**using namespace std;**

**//in this quesion many time implication use> to find out it**

**bool implies(bool p, bool q) {**

**return !p || q;**

**}**

**//this fuction is to get value of all premises in the quesion**

**bool validity(bool p, bool q, bool r, bool s, bool t, bool u) {**

**bool premise1 = implies(p && t, r || s);**

**bool premise2 = implies(q, u && t);**

**bool premise3 = implies(u, p);**

**bool premise4 = !s;**

**bool conclusion = implies(q, r);**

**cout<< p <<"|\t"<< q <<"|\t"<< r<<"|\t"<< s <<"|\t"<< t <<"|\t" <<u <<"|\t"<<premise1<<"|\t\t"<<premise2<<"|\t\t"<<premise3<<"|\t\t"<<premise4<<"|\t\t"<<conclusion<<"|\t\t\n";**

**return premise1 && premise2 && premise3 && premise4 && conclusion;**

**}**

**int main() {**

**bool valid = true;**

**cout<<"Premises: (p^t)>(rvs), q>(u^t), u>p, and ~s\n";**

**cout<<"Conclusion: q>r\n";**

**//total 6 loop for finding p,q,r,s,t,u**

**cout<<"\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";**

**cout<<"p|\tq|\tr|\ts|\tt|\t u|\t(p^t)>(rvs)|\tq>(u^t)|\tu>p|\t~s|\t conclusion|\t \n";**

**cout<<"\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n";**

**for (int p = 0; p <= 1; p++) {**

**for (int q = 0; q <= 1; q++) {**

**for (int r = 0; r <= 1; r++) {**

**for (int s = 0; s <= 1; s++) {**

**for (int t = 0; t <= 1; t++) {**

**for (int u = 0; u <= 1; u++) {**

**if (!validity(p, q, r, s, t, u)) {**

**valid = false;**

**break;**

**}**

**}**

**}**

**}**

**}**

**}**

**}**

**if (valid) {**

**cout << "The argument is valid." << endl;**

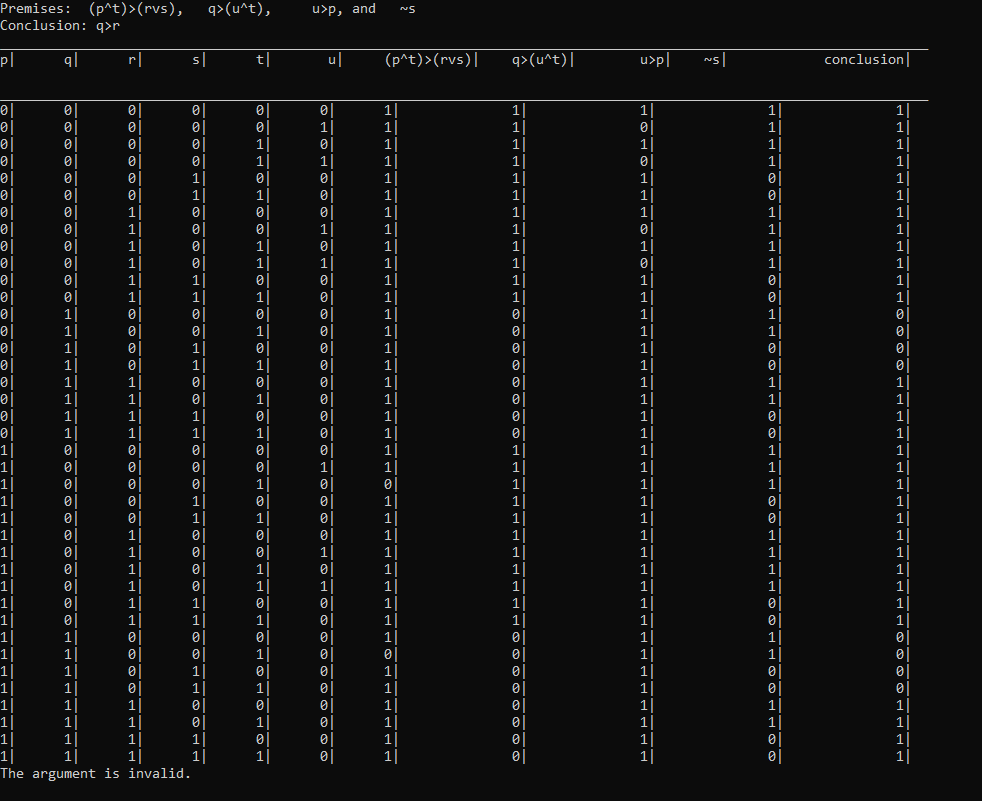
**} else {**

**cout << "The argument is invalid." << endl;**

**}**

**return 0;**

**}**

****

**Generic**

**#include <iostream>**

**using namespace std;**

**//all premises true conclusion true argument is valid**

**//all premises true conclusion false argument is invalid**

**//if premises have true ,false value no argument**

**bool evaluate(bool p, bool q, bool r, bool s, bool t, bool u, string expr) {**

**bool result;**

**if (expr == "~s") {**

**result = !s;**

**} else if (expr == "p&t") {**

**result = p && t;**

**}**

**else if (expr == "p|t") {**

**result = p ||t;**

**}**

**else if (expr == "r|t") {**

**result = r ||t;**

**//we can expand the else if**

**} else if (expr == "r|s") {**

**result = r || s;**

**} else if (expr == "u&t") {**

**result = u && t;**

**} else if (expr == "u|t") {**

**result = u || t;**

**}else if (expr == "u>p") {**

**result = !u || p;**

**} else if (expr == "q>(u&t)") {**

**result = !q || (u && t);**

**} else if (expr == "q>(r|s)") {**

**result = !q || (r || s);**

**} else {**

**cout << "Invalid expression: " << expr << endl;**

**result = false;**

**}**

**return result;**

**}**

**int main() {**

**int p, q, r, s, t, u;**

**bool conclusion, premises;**

**string premise1, premise2, premise3, conclusionEq;**

**// Read in the premises and conclusion from the user**

**cout << "Enter premise 1: ";**

**cin >> premise1;**

**cout << "Enter premise 2: ";**

**cin >> premise2;**

**cout << "Enter premise 3: ";**

**cin >> premise3;**

**cout << "Enter conclusion equation: ";**

**cin >> conclusionEq;**

**// Evaluate the premises and conclusion for all possible truth values of p, q, r, s, t, u**

**for (p = 0; p <= 1; p++) {**

**for (q = 0; q <= 1; q++) {**

**for (r = 0; r <= 1; r++) {**

**for (s = 0; s <= 1; s++) {**

**for (t = 0; t <= 1; t++) {**

**for (u = 0; u <= 1; u++) {**

**premises = evaluate(p, q, r, s, t, u, premise1) && evaluate(p, q, r, s, t, u, premise2) && evaluate(p, q, r, s, t, u, premise3);**

**conclusion = evaluate(p, q, r, s, t, u, conclusionEq);**

**if (premises && !conclusion) {**

**cout << "The argument is not valid." << endl;**

**cout << "Counter!!!! p=" << p << ", q=" << q << ", r=" << r << ", s=" << s << ", t=" << t << ", u=" << u << ", conclusion=" << conclusion<< endl;**

**return 0;**

**}**

**}**

**}**

**}**

**}**

**}**

**}**

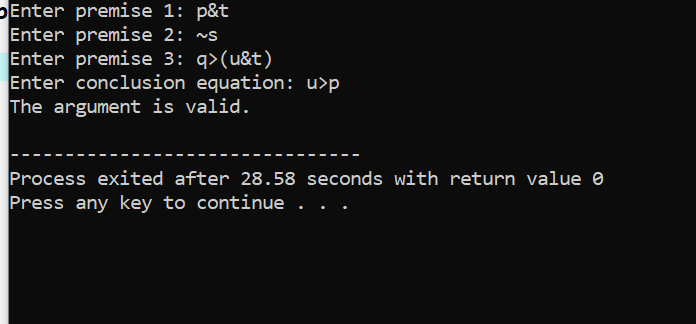
**// If no counterexample is found, the argument is valid**

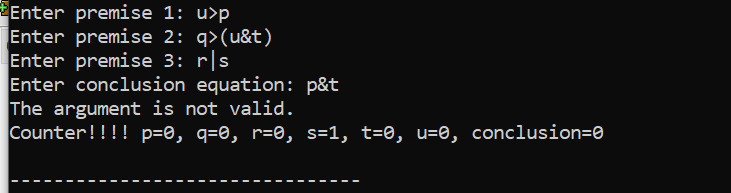
**//if premises sari true na b ho to b invlid**

**cout << "The argument is valid." << endl;**

**return 0;**

**}**

****

****