

EAST WEST UNIVERSITY

P R O J E C T R E P O R T

Course Title: Discrete Mathematics

Course Code: CSE106

Sec: 06

**Submitted to:**

**Dr. Mohammad Salah Uddin**

Associate Professor  
Department of Computer Science & Engineering

**Submitted by:**

(1) Md. Saiful Islam

ID: 2022-3-60-045

(2) Erfan Sarker Efte

ID: 2022-3-60-042

(3) Shanghita Naha Sristy

ID: 2022-3-60-311

**Submission Date:**

22th May, 2023

**introduction:**

Using C program, we are going to randomly generate a relational matrix (which represent relation) with dimension n where n is the number of distinct elements on a set. Then it is going to verify the properties of the relation; such as symmetric, anti-symmetric, transitive, and equivalence, etc. Also it will determine computational time in milliseconds. It will check whether the relation represents any function or not.

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define max 700

void symgen(int n, int m[max][max]) {

srand(time(NULL));

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

for (int j = i; j < n; j++) {

if (i == j) {

m[i][j] = 1;

} else {

int value = rand() % 2;

m[i][j] = value;

m[j][i] = value;

}

}

}

}

void antisymgen(int n, int m[max][max]) {

srand(time(NULL));

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

for (int j = 0; j < n; j++) {

if (i == j) {

m[i][j] = 0;

} else {

int value = rand() % 2;

m[i][j] = value;

m[j][i] = 1 - value;

}

}

}

}

void trangen(int n, int m[max][max]) {

srand(time(NULL));

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

for (int j = 0; j < n; j++) {

if (i == j) {

m[i][j] = 1;

} else {

m[i][j] = rand() % 2;

}

}

}

for (int k = 0; k < n; k++) {

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

for (int j = 0; j < n; j++) {

if (m[i][k] == 1 && m[k][j] == 1) {

m[i][j] = 1;

}

}

}

}

}

void printm(int n, int m[max][max]) {

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

for (int j = 0; j < n; j++) {

printf("%d ", m[i][j]);

}

printf("\n");

}

}

int symmetricm(int n,int m[max][max]) {

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

for (int j = 0; j < n; j++) {

if (m[i][j] != m[j][i]) {

return 0;

}

}

}

return 1;

}

int anti\_symmetric(int n, int m[max][max]) {

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

for (int j = 0; j < n; j++) {

if (m[i][j] && m[j][i] && i != j) {

return 0;

}

}

}

return 1;

}

int transitive(int n, int m[max][max]) {

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

for (int j = 0; j < n; j++) {

for (int k = 0; k < n; k++) {

if (m[i][j] && m[j][k] && !m[i][k]) {

return 0;

}

}

}

}

return 1;

}

int equivalence(int n, int m[][max]) {

return symmetricm(n,m) && transitive(n,m);

}

int function(int n, int m[max][max]) {

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

int count = 0;

for (int j = 0; j < n; j++) {

if (m[i][j]) {

count++;

}

}

if (count != 1) {

return 0;

}

}

return 1;

}

int main () {

int n;

clock\_t start,end;

double cpu\_time\_m;

double cpu\_time\_f;

start=clock();

printf("Enter the dimension value(n):");

scanf("%d",&n);

int m[max][max];

srand(time(NULL));

int x=rand()% 3;

switch(x) {

case 0 :

symgen(n,m);

break;

case 1 :

antisymgen(n,m);

break;

case 2 :

trangen(n,m);

break;

}

printf("\nRandom Matrix:");

printf("\n---------------\n\n");

printm(n,m);

printf("\nVerifying the relation metrix:");

printf("\n------------------------------\n\n");

if (symmetricm(n, m)) {

printf("The relation is symmetric.\n");

} else {

printf("The relation is not symmetric.\n");

}

if (anti\_symmetric(n,m)) {

printf("The relation is anti-symmetric.\n");

} else {

printf("The relation is not anti-symmetric.\n");

}

if (transitive(n, m)) {

printf("The relation is transitive.\n");

} else {

printf("The relation is not transitive.\n");

}

if (equivalence(n,m)) {

printf("The relation is an equivalence relation.\n");

} else {

printf("The relation is not an equivalence relation.\n");

}

end = clock();

cpu\_time\_m = ((double) (end - start)) / CLOCKS\_PER\_SEC;

start=clock();

if (function(n,m)) {

printf("The relation represents a function.\n");

} else {

printf("The relation does not represent a function.\n");

}

end = clock();

cpu\_time\_f = ((double) (end - start)) / CLOCKS\_PER\_SEC;

printf("\nComputation Time:");

printf("\n------------------\n\n");

printf("verification time for relation: %.2f milliseconds\n", cpu\_time\_m \* 1000);

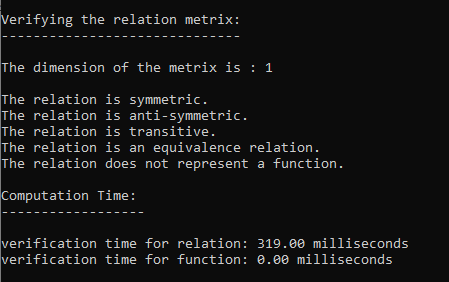
printf("verification time for function: %.2f milliseconds\n", cpu\_time\_f \* 1000);

return 0;

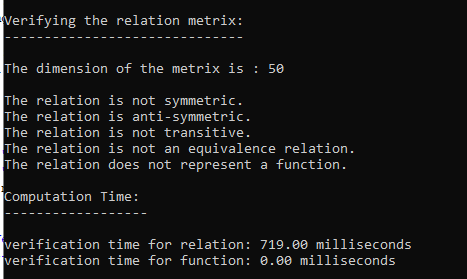
}

**Output:**

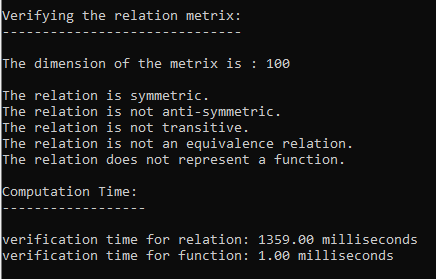
n=1,



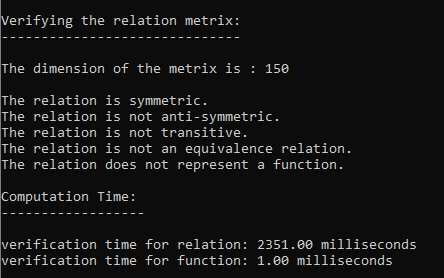
n=50,



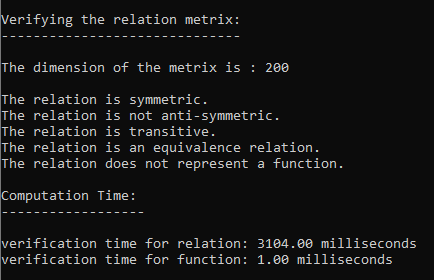
n=100,



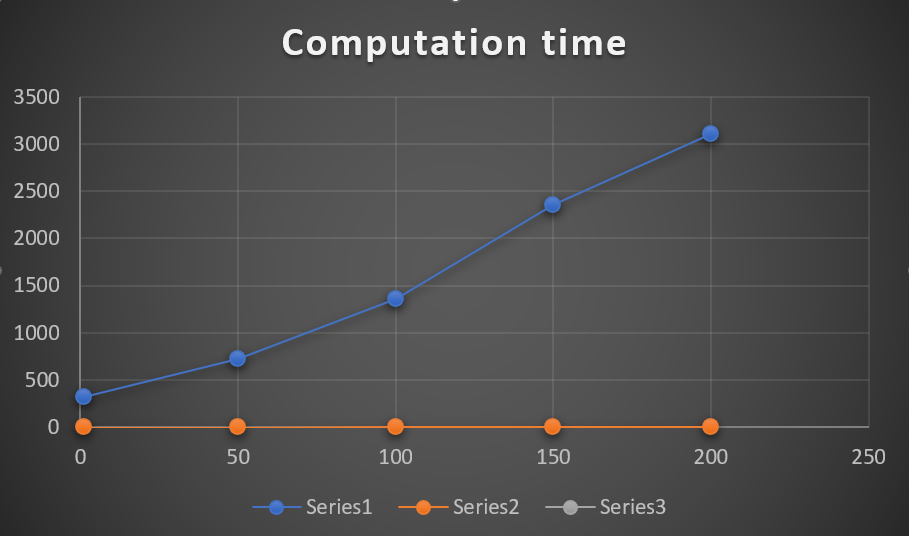
n=150,



n=200,



**Graph:**



**Graph Showing Computational time vs the number of dimension**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Dimension | Computing time of matrix verification | Computing time of function verification | | 1 | 319 | 0 | | 50 | 719 | 0 | | 100 | 1359 | 1 | | 150 | 2351 | 1 | | 200 | 3104 | 1 | |  |
|  |  |
|  |  |
| **TIME COMPLEXITY:**  The time complexity of an algorithm approximates just how much time it would take to solve a task of a specific size. Also, the time complexity of an algorithm may be represented as the number of operations performed by the algorithm when the input is of a certain size. According to the directions for our project, we created a graph of processing time vs n-vertices and compared it to the Big O notation graph. As an outcome, we determined Big O’s estimated time complexity ( n3 ). In the theory, we implemented three nested loops and a couple of extra functions to correctly build the entire program.  **THEORETICAL TIME COMPLEXITY** |  |

|  |  |
| --- | --- |
| **Statement** | **Big O notation** |
| for (int i = 0; i < n; i++) {  for (int j = i; j < n; j++) {  if (i == j) {  m[i][j] = 1;  } else {  int value = rand() % 2;  m[i][j] = value;  m[j][i] = value;  }  }  }  } | f1=n\*(3n+1)+1  =O(n2) |
| for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (i == j) {  m[i][j] = 0;  } else {  int value = rand() % 2;  m[i][j] = value;  m[j][i] = 1 - value;  }  }  }  } | f2=n\*(3n+1)+1  =O(n2) |
| for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (i == j) {  m[i][j] = 1;  } else {  m[i][j] = rand() % 2;  }  }  }  for (int k = 0; k < n; k++) {  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (m[i][k] == 1 && m[k][j] == 1) {  m[i][j] = 1;  }  }  }  }  } | f3=(n\*(3n+1)+1)+(n\*(n\*(2n+1)+1)+1)  =O(n3) |
| for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  printf("%d ", m[i][j]);  }  printf("\n");  } | f4=n\*(n+1)+1  =O(n2) |
| for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (m[i][j] != m[j][i]) {  return 0;  }  }  }  return 1; | f5=n\*(2n+1)+1  =O(n2) |
| for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (i == j) {  m[i][j] = 0;  } else {  int value = rand() % 2;  m[i][j] = value;  m[j][i] = 1 - value;  }  }  } | f6=n\*(3n+1)+1  =O(n2) |
| for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (m[i][j] && m[j][i] && i != j) {  return 0;  }  }  }  return 1; | f7=n\*(2n+1)+1  =O(n2) |
| for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  for (int k = 0; k < n; k++) {  if (m[i][j] && m[j][k] && !m[i][k]) {  return 0;  }  }  }  }  return 1; | f8=n\*(2n+1)+1  =O(n2) |
| for (int i = 0; i < n; i++) {  int count = 0;  for (int j = 0; j < n; j++) {  if (m[i][j]) {  count++;  }  }  if (count != 1) {  return 0;  }  }  return 1; | f9=n\*(3n+1)+1  =O(n2) |
| The big O notation | O (max (n2, n3)) = O(n3) |

Hence, the time complexity of our program is: O(n)=n3

So, we can see that the graph’s time complexity and the program’s time

complexity which we determined have been matched.

**{ T H E E N D }**