**PRAVEER RAJ**

**MATHS AND COMPUTING**

**ROLL: 1**

**REG: 230957002**

**DATE: 21-01-2025**

***Non-Recursive Algorithms:***

**L3.1** Implement a non-recursive algorithm to determine the maximum element in an unsorted array of size n and analyze its time efficiency

#include <iostream>

#include <vector>

#include <climits>

int findMax(const std::vector<int>& arr)

{

int maxElement = INT\_MIN;

for (size\_t i = 0; i < arr.size(); ++i)

{

if (arr[i] > maxElement)

{

maxElement = arr[i];

}

}

return maxElement;

}

int main()

{

int rawArray[] = {3, 1, 4, 1, 5, 9, 2};

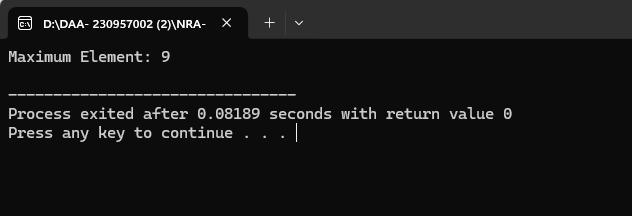
std::vector<int> arr(rawArray, rawArray + sizeof(rawArray) / sizeof(rawArray[0]));

std::cout << "Maximum Element: " << findMax(arr) << std::endl;

return 0;

}

OUTPUT:-



**L3.2** Develop a non-recursive algorithm to identify the unique element in an unsorted array where all other elements appear exactly twice and analyze its time efficiency.

#include <iostream>

#include <vector>

int findUnique(const std::vector<int>& arr) {

int uniqueElement = 0;

for (size\_t i = 0; i < arr.size(); ++i) {

uniqueElement ^= arr[i];

}

return uniqueElement;

}

int main() {

int rawArray[] = {4, 3, 6, 3, 4};

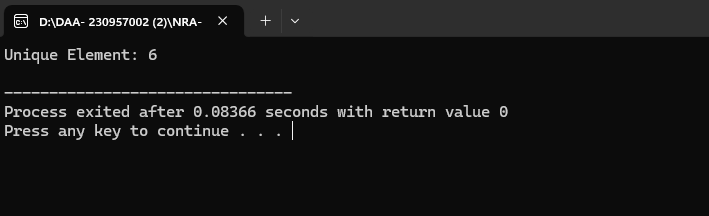
std::vector<int> arr(rawArray, rawArray + sizeof(rawArray) / sizeof(rawArray[0]));

std::cout << "Unique Element: " << findUnique(arr) << std::endl;

return 0;

}

OUTPUT:-



**L3.3** Implement a non-recursive algorithm to multiply two square matrices of order n using the standard matrix multiplication approach and analyze its time efficiency.

#include <iostream>

#include <vector>

std::vector<std::vector<int> > multiplyMatrices(const std::vector<std::vector<int> >& A,

const std::vector<std::vector<int> >& B, int n) {

std::vector<std::vector<int> > result(n, std::vector<int>(n, 0));

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

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

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

result[i][j] += A[i][k] \* B[k][j];

}

}

}

return result;

}

int main() {

int rawA[2][2] = {{2, 2}, {3, 7}};

int rawB[2][2] = {{3, 6}, {6, 8}};

int n = 2;

std::vector<std::vector<int> > A, B;

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

A.push\_back(std::vector<int>(rawA[i], rawA[i] + n));

B.push\_back(std::vector<int>(rawB[i], rawB[i] + n));

}

std::vector<std::vector<int> > result = multiplyMatrices(A, B, n);

std::cout << "Product Matrix:" << std::endl;

for (size\_t i = 0; i < result.size(); ++i) {

for (size\_t j = 0; j < result[i].size(); ++j) {

std::cout << result[i][j] << " ";

}

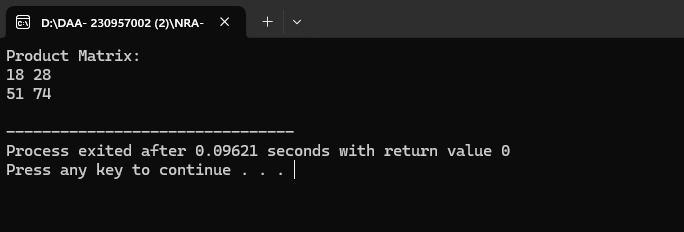
std::cout << std::endl;

}

return 0;

}

OUTPUT:-



**L3.4** Design a non-recursive algorithm to compute the number of binary digits in the binary representation of a given positive decimal integer n and analyze its time efficiency

#include <iostream>

int countBinaryDigits(int n) {

int count = 0;

while (n > 0) {

n /= 2;

++count;

}

return count;

}

int main() {

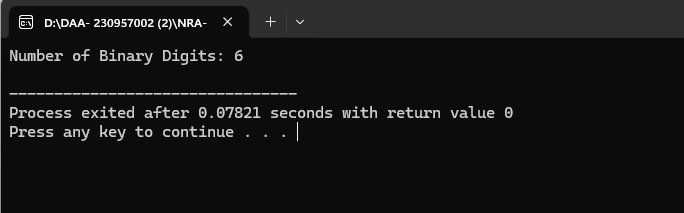
int n = 37;

std::cout << "Number of Binary Digits: " << countBinaryDigits(n) << std::endl;

return 0;

}

OUTPUT:-



**Recursive Algorithms:**

**L3.5** Develop a recursive algorithm to compute the factorial of a given non-negative integer n and analyze its time efficiency

#include <iostream>

using namespace std;

int factorial(int n) {

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

return 1;

}

return n \* factorial(n - 1);

}

int main() {

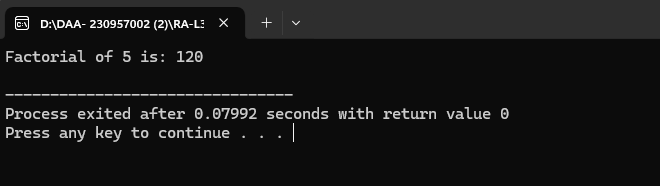
int n = 5;

cout << "Factorial of " << n << " is: " << factorial(n) << endl;

return 0;

}

OUTPUT:-



**L3.6** Implement a recursive algorithm to solve the Tower of Hanoi problem for n disks and analyze its time efficiency.

#include <iostream>

using namespace std;

void towerOfHanoi(int n, char fromRod, char toRod, char auxRod) {

if (n == 1) {

cout << "Move disk 1 from " << fromRod << " to " << toRod << endl;

return;

}

towerOfHanoi(n - 1, fromRod, auxRod, toRod);

cout << "Move disk " << n << " from " << fromRod << " to " << toRod << endl;

towerOfHanoi(n - 1, auxRod, toRod, fromRod);

}

int main() {

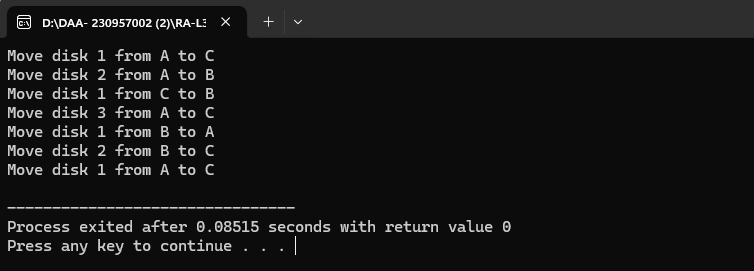
int n = 3;

towerOfHanoi(n, 'A', 'C', 'B');

return 0;

}

OUTPUT:



**L3.7** Implement a recursive algorithm to determine the number of binary digits in the binary representation of a given positive decimal integer n and analyze its time efficiency.

#include <iostream>

using namespace std;

int countBinaryDigits(int n) {

if (n == 0) {

return 0;

}

return 1 + countBinaryDigits(n / 2);

}

int main() {

int n = 37;

cout << "Number of Binary Digits in " << n << " is: " << countBinaryDigits(n) << endl;

return 0;

}

OUTPUT:

