Name: Arpita Dinesh Singh
2nd-year B.tech
Computer engineering

Roll No:231071005

DESIGN ANALYSIS OF ALGORITHM

LABORATORY 4

EXPERIMENT TASK- 1
(course-code choices of 100 students)

4.CODE
Code for CSV file for 100 students course code choice:

```
import csv
import random

# Function to generate a random 6-digit course code
def generate_course_code():
    return f"{random.randint(0, 999999):06d}"

# Create a list to hold student IDs and course codes
student_data = []

# Generate data for 100 students
for student_id in range(1, 101):
```

```
    course_code = generate_course_code()
    student_data.append((student_id, course_code))

# Write the data to a CSV file
with open('course_codes.csv', mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['StudentID', 'CourseCode'])  # Write the header
    writer.writerows(student_data)  # Write student data

print("CSV file 'course_codes.csv' created successfully with 100
random course codes.")
```

CODE USING DIVIDE AND CONQUER:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <sstream>
#include <random>
#include <algorithm>
#include <map>

using namespace std;

// Function prototypes
int mergeSortAndCount(vector<int>& arr, int left, int right);
int countInversionsInDigits(int number);
int mergeAndCount(vector<int>& arr, int left, int mid, int right);

// Function to count inversions in the digits of a single number
```

```
int countInversionsInDigits(int number) {
    string numStr = to_string(number);
    int n = numStr.size();
    vector<int> digits(n);

    for (int i = 0; i < n; ++i) {
        digits[i] = numStr[i] - '0';  // Convert char to int
    }

    // Count inversions using a modified merge sort
    return mergeSortAndCount(digits, 0, n - 1);
}

// Function to merge two halves and count inversions
int mergeAndCount(vector<int>& arr, int left, int mid, int right) {
    int i = left;
    int j = mid + 1;
    int k = 0;
    vector<int> temp(right - left + 1);
    int inv_count = 0;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
            inv_count += (mid - i + 1);  // Count inversions
        }
    }
```

```
    while (i <= mid) {
        temp[k++] = arr[i++];
    }

    while (j <= right) {
        temp[k++] = arr[j++];
    }

    for (int p = left; p <= right; p++) {
        arr[p] = temp[p - left];
    }

    return inv_count;
}

// Function to count inversions using merge sort
int mergeSortAndCount(vector<int>& arr, int left, int right) {
    int inv_count = 0;
    if (left < right) {
        int mid = left + (right - left) / 2;

        inv_count += mergeSortAndCount(arr, left, mid);
        inv_count += mergeSortAndCount(arr, mid + 1, right);
        inv_count += mergeAndCount(arr, left, mid, right);
    }
    return inv_count;
}

bool isValidCourseCode(int courseCode) {
```

```
    return courseCode >= 100000 && courseCode <= 999999; //
Check if it's a 6-digit number
}

int main() {
    vector<int> courseCodes;
    ifstream inFile("course_codes.csv");

    // Check if the file is open
    if (!inFile.is_open()) {
        cerr << "Error opening file!" << endl;
        return 1;
    }

    string line;
    getline(inFile, line); // Skip header
    while (getline(inFile, line)) {
        stringstream ss(line);
        string studentId;
        int courseCode;
        getline(ss, studentId, ','); // Read student ID
        ss >> courseCode; // Read course code as integer

        if (isValidCourseCode(courseCode)) {
            courseCodes.push_back(courseCode);
        }
    }
    inFile.close();
    // Map to count number of students by inversion counts
    map<int, int> inversionCountMap;
```

```
    vector<pair<int, int>> results; // To store course codes and their
inversion counts
    // Process all course codes and count inversions
    for (int code : courseCodes) {
        int inversionCount = countInversionsInDigits(code);
        inversionCountMap[inversionCount]++;
        results.push_back({code, inversionCount});
    }
    // Write results to CSV
    ofstream outFile("inversion_counts.csv");
    outFile << "Course Code,Inversion Count\n"; // Write CSV
header
    for (const auto& result : results) {
        outFile << result.first << "," << result.second << "\n";
    }
    outFile.close();

    // Randomly select 10 course codes and print their inversion
counts
    random_device rd;
    mt19937 eng(rd());
    shuffle(courseCodes.begin(), courseCodes.end(), eng);

    cout << "Inversion counts for 10 randomly selected course
codes:\n";
    for (int i = 0; i < 10 && i < courseCodes.size(); ++i) {
        int selectedCode = courseCodes[i];
        int inversionCount = countInversionsInDigits(selectedCode);
        cout << "Course Code: " << selectedCode
             << " | Inversion Count: " << inversionCount << endl;
```

```
    }

    // Output the count of students with 0, 1, 2, and 3 inversions
    cout << "\nNumber of students with:\n";
    for (int i = 0; i <= 3; ++i) {
        cout << i << " inversion(s): " << inversionCountMap[i] <<
endl;
    }
   // Write summary to CSV
    ofstream summaryFile("inversion_summary.csv");
    summaryFile << "Inversion Count,Number of Students\n"; //
Write summary header
    for (int i = 0; i <= 3; ++i) {
        summaryFile << i << "," << inversionCountMap[i] << "\n";
    }
    summaryFile.close();
    return 0;
}
```

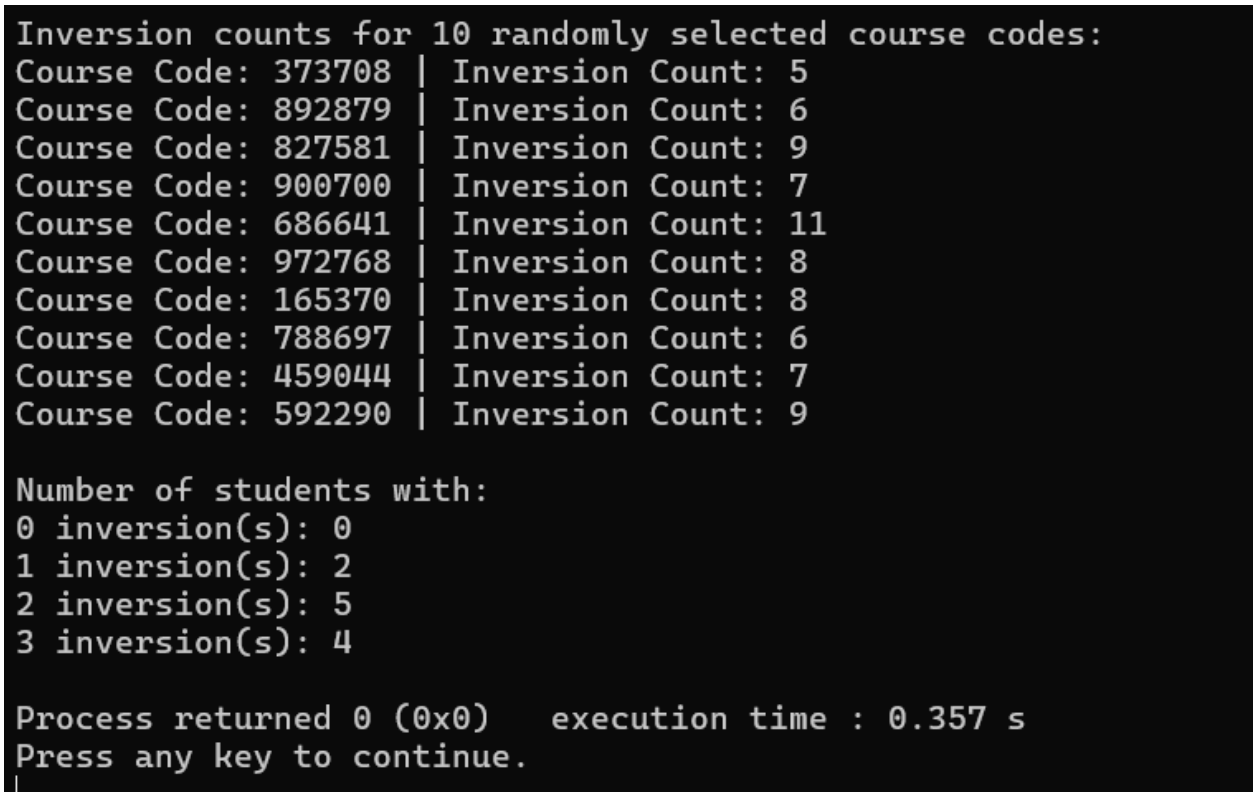
CODE FOR BRUTE FORCE FUNCTION:

```
// Function to count inversions using brute force
int countInversionsBruteForce(int number) {
    string numStr = to_string(number);
    int n = numStr.size();
    int inv_count = 0;

    // Compare every pair of digits
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (numStr[i] > numStr[j]) {
                inv_count++;
```

```
        }
      }
    }
    return inv_count;
}
```

5.OUTPUT:



6.CONCLUSION:

This laboratory exercise explored the analysis of divide-and-conquer and brute force. For counting inversions in course codes, the divide-and-conquer strategy, implemented via a modified merge sort, achieved a time complexity of O(nlogn). This efficiency arose from its ability to break the problem into smaller subproblems and merge results effectively. In contrast, the brute force method, with a time complexity of O(n^2), proved less efficient for larger datasets due to its quadratic growth.

EXPERIMENT TASK- 2
(Integer multiplication)

4.CODE :

CODE USING DIVIDE AND CONQUER:

```
#include <iostream>
#include <cmath>
#include <string>
#include <limits>
using namespace std;

long long karatsubaMultiply(long long x, long long y) {
    // Base case for recursion
    if (x < 10 || y < 10) {
        return x * y;
    }

    // Calculate the size of the numbers
    int n = max(to_string(x).length(), to_string(y).length());
    int half = n / 2;

    // Calculate powers of 10
    long long power = 1;
    for (int i = 0; i < half; ++i) {
        power *= 10; // Calculate 10^half
    }

    // Split the digits
    long long a = x / power;      // Left half of x
    long long b = x % power;      // Right half of x
```

```
    long long c = y / power;       // Left half of y
    long long d = y % power;       // Right half of y

    // 3 recursive calls
    long long ac = karatsubaMultiply(a, c);          // a * c
    long long bd = karatsubaMultiply(b, d);          // b * d
    long long ad_plus_bc = karatsubaMultiply(a + b, c + d) - ac -
bd; // (a + b)(c + d) - ac - bd

    // Combine the results using integers
    return ac * power * power + ad_plus_bc * power + bd;
}

int main() {
    long long x, y;

    for (int i = 0; i < 10; ++i) {
        while (true) {
            cout << "Enter pair " << (i + 1) << " for Karatsuba
multiplication (two integers): ";
            cin >> x >> y;

            // Check for valid input
            if (cin.fail()) {
                cin.clear(); // Clear the error state
                cin.ignore(numeric_limits<streamsize>::max(), '\n'); //
Discard invalid input
                cout << "Invalid input. Please enter two integers." <<
endl;
            } else {
```

```
            break; // Valid input, exit the loop
        }
    }

    long long result = karatsubaMultiply(x, y);
    cout << "Karatsuba Multiplication of " << x << " and " << y <<
": " << result << endl;
  }

  return 0;
}
```

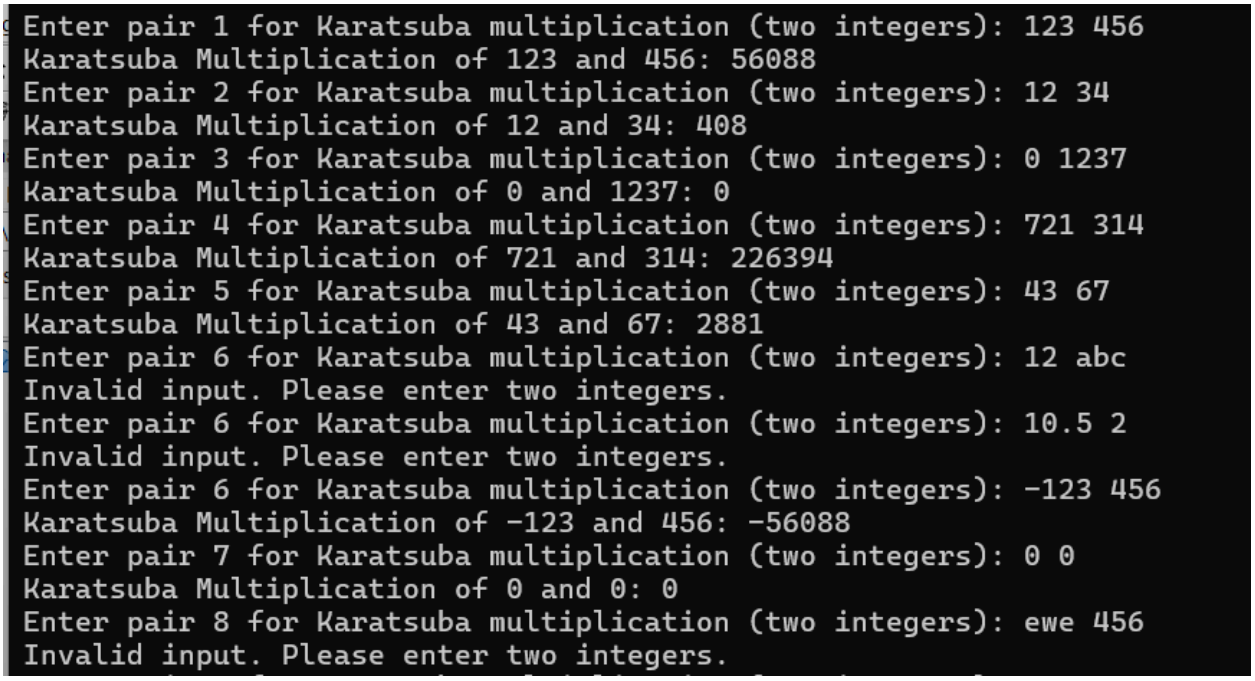
CODE FOR BRUTE FORCE FUNCTION:

```
long long bruteForceMultiply(long long x, long long y) {
  long long result = 0;
  long long shift = 0;
  while (y > 0) {
    // Multiply the last digit of y with x
    long long last_digit = y % 10;
    long long current_product = x * last_digit;
    // Shift the current product according to the position
    result += current_product * pow(10, shift);

    // Prepare for the next iteration
    y /= 10; // Remove the last digit of y
    shift++; // Increase the shift for the next digit
  }

  return result;
}
```

5.OUTPUT:



6.CONCLUSION:

We also examined integer multiplication through the Karatsuba algorithm, which operates in O(n^{1.585}) and a brute force method with O(n^2) complexity. Consider large integers of size 10, 50, 100, 500, and 1000 digits; the Karatsuba algorithm demonstrated significant performance advantages, especially as the size of the numbers increased.