DAA LAB 4

Amit Rajkumar Ingle

S.Y.B.Tech Computer

231070020

EXPERIMENTAL TASK-1:

AIM: Consider first/second year course-code choices of 100 students. Find inversion count of these choices. Find students with zero, one, two, three inversion counts comment on your result.

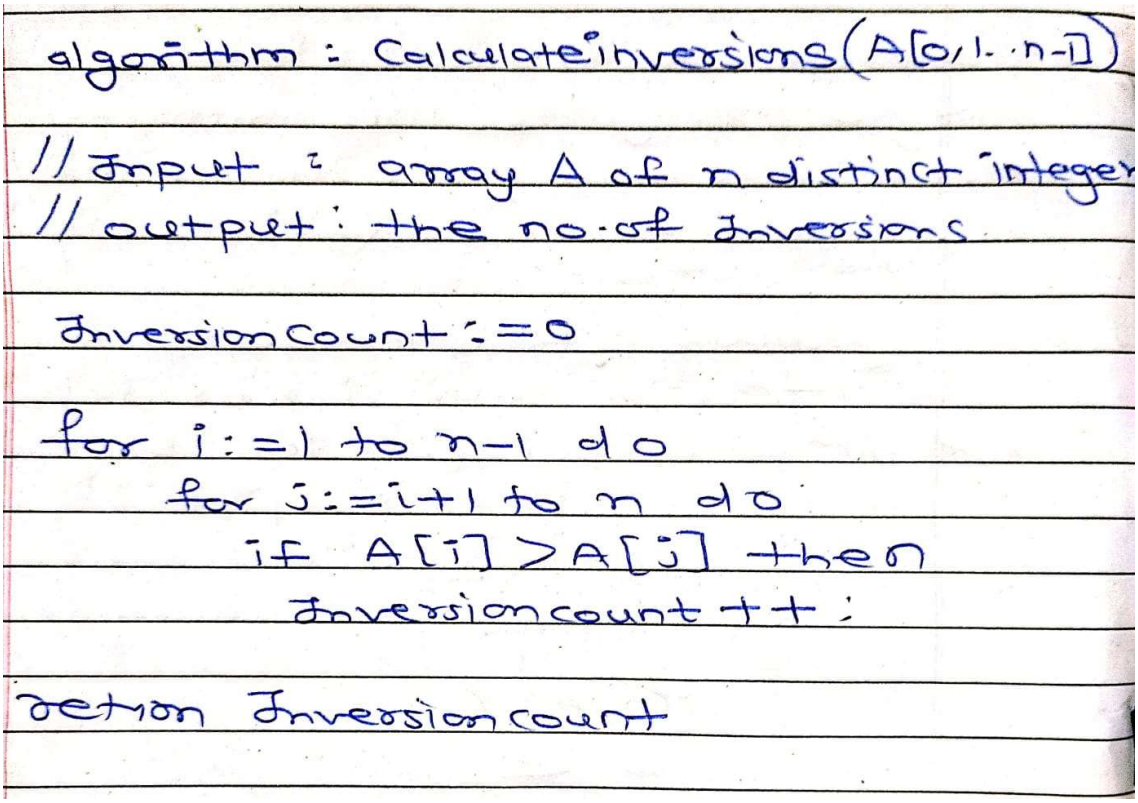
THEORY:

Counting inversions in an array measures its sortedness, defined as pairs ((i, j)) where (i < j) and (A[i] > A[j]). A sorted array has zero inversions.

A naive (O(n^2)) method uses nested loops, but a more efficient approach employs a modified merge sort, achieving O(n log n) time. During merging, if an element from the right half is smaller than one from the left, it indicates inversions with all remaining elements in the left half. This method effectively counts inversions and has applications in sorting analysis and sequence similarity.

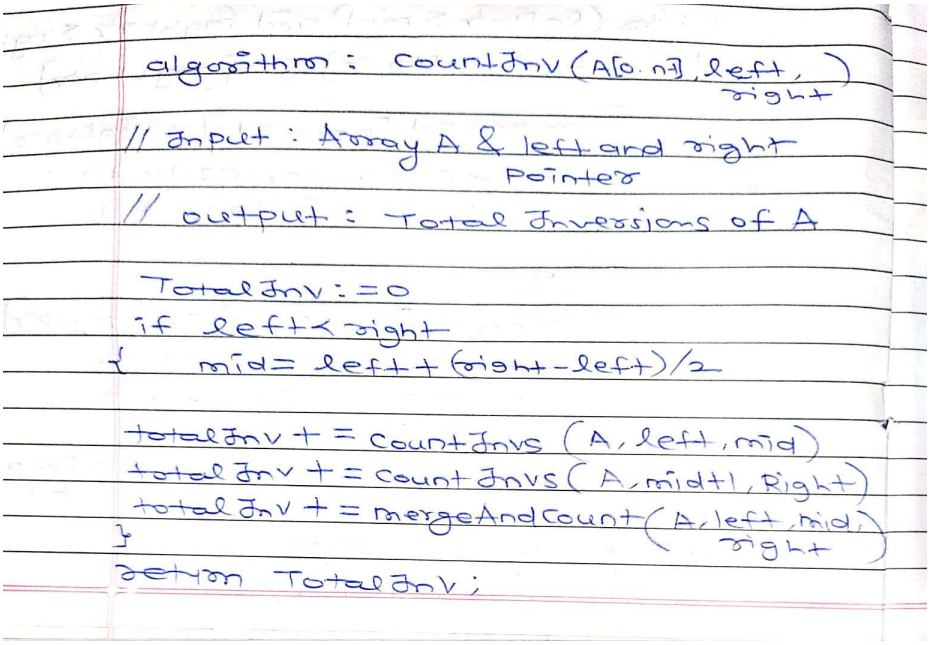
ALGORITHM:

A) Brute force Algorithm :



```
algorithm : CalculateInversions(A[0,1...n-1])

// Input : array A of n distinct integer
// output : the no.of Inversions

InversionCount := 0

for i := 1 to n-1 do
    for j := i+1 to n do
        if A[i] > A[j] then
            Inversioncount++;

return Inversioncount
```

B) Divide and Conquer Algorithm:



```
algorithm : CountInv(A[0...n], left, right)

// Input : Array A & left and right pointer

// output : Total Inversions of A

TotalInv := 0
if left < right
{   mid = left + (right - left)/2

totalInv += CountInvs(A, left, mid)
totalInv += CountInvs(A, midtl, Right)
totalInv += mergeAndCount(A, left, mid, right)
}
return TotalInv;
```

algorithm : mergeandCount(A, left, mid, right)

// Input : Array A. left & right
points to start and end of
A. middle of A is mid.
// output : split-inversions of A

leftptr := left;

rightptr := mid+1;

mergeIdx := 0 ; // Index for merged array

split-invcount := 0

merged (right - left+1) // New array after merging

while (leftptr ≤ mid) and rightptr ≤ right
do if A[leftptr] ≤ A[rightptr]

merged [mergeIdx] = A[leftptr]
mergeIdx = mergeIdx + 1 ;
leftptr = leftptr + 1

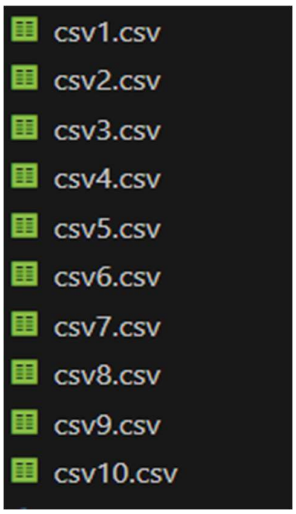
else

merged [mergeIdx] = A[rightptr]
mergeIdx = mergeIdx + 1 ;
rightptr = rightptr + 1 ;
&split-invcount =
split-invcount ++ (mid - leftptr + 1

return split-invcount

TEST CASES:

```
TEST CASES :

CSV FILE 1,2,3,9,10 -> NORMAL
CSV FILE 4 -> DUPLICATES
CSV FILE 5 -> NEGATIVE
CSV FILE 6 -> MISSING DATA
CSV FILE 7 -> OUT OF RANGE
CSV FILE 8 -> EMPTY
```

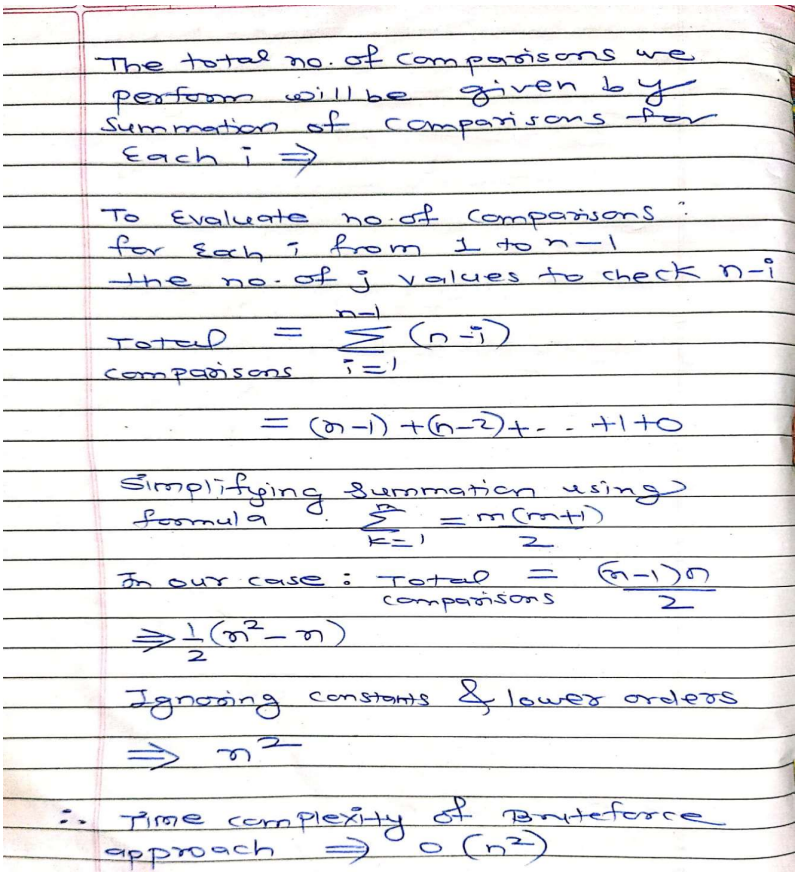


BASIC.cpp > DAA4.cpp > csv1.csv

```
StudentID,Choice1,Choice2,Choice3,Choice4,Choice5
Student_1,3003,3004,3001,3005,3002
Student_2,3002,3005,3001,3004,3003
Student_3,3003,3001,3004,3005,3002
Student_4,3003,3001,3004,3002,3005
Student_5,3002,3003,3005,3004,3001
Student_6,3003,3004,3002,3005,3001
Student_7,3002,3004,3005,3003,3001
Student_8,3004,3002,3001,3003,3005
Student_9,3003,3005,3004,3002,3001
Student_10,3005,3004,3002,3001,3003
Student_11,3001,3005,3002,3004,3003
Student_12,3005,3003,3004,3002,3001
Student_13,3002,3001,3003,3005,3004
Student_14,3004,3005,3003,3001,3002
Student_15,3002,3001,3004,3003,3005
Student_16,3002,3004,3003,3001,3005
Student_17,3001,3003,3005,3002,3004
Student_18,3002,3001,3003,3005,3004
Student_19,3001,3004,3003,3005,3002
Student_20,3001,3005,3004,3002,3003
Student_21,3004,3002,3003,3005,3001
Student_22,3005,3002,3001,3004,3003
Student_23,3003,3002,3004,3005,3001
Student_24,3003,3002,3001,3004,3005
Student_25,3004,3005,3003,3001,3002
Student_26,3004,3003,3001,3002,3005
Student_27,3003,3002,3004,3005,3001
Student_28,3005,3003,3001,3002,3004
Student_29,3004,3002,3001,3003,3005
Student_30,3004,3001,3002,3005,3003
Student_31,3002,3003,3001,3005,3004
Student_32,3005,3001,3004,3002,3003
Student_33,3002,3003,3001,3005,3004
Student_34,3004,3005,3002,3001,3003
```

TIME COMPLEXITY:

1) BRUTEFORCE APPROACH:



The total no. of comparisons we perform will be given by summation of comparisons for Each i ⇒

To Evaluate no. of comparisons: for Each i from 1 to n−1 the no. of j values to check n−i

$$\text{Total comparisons} = \sum_{i=1}^{n-1} (n-i)$$

$$= (n-1) + (n-2) + \cdots + 1 + 0$$

Simplifying Summation using formula $\sum_{k=1}^{n} = \frac{n(n+1)}{2}$

In our case: $\text{Total comparisons} = \frac{(n-1)n}{2}$

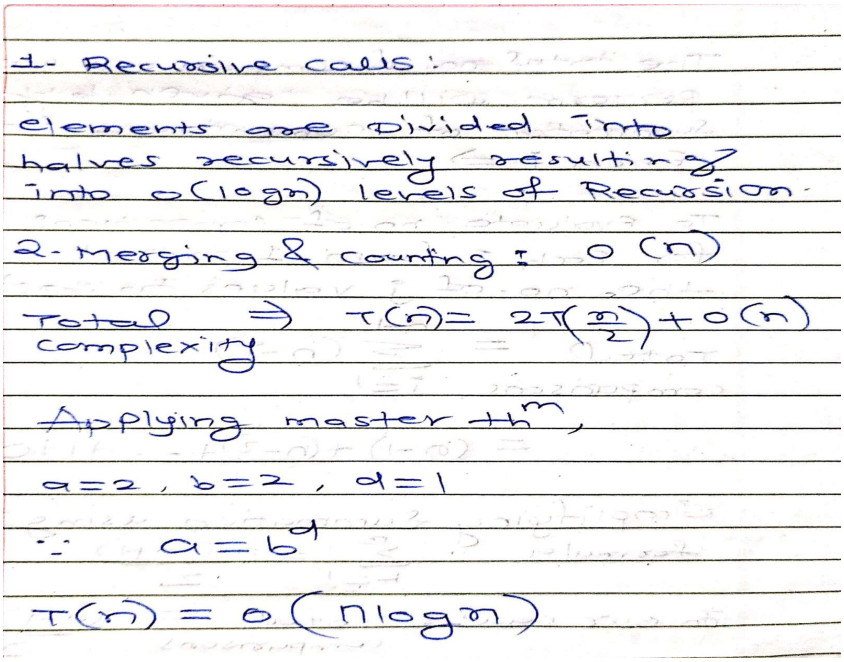
$$\Rightarrow \frac{1}{2}(n^2 - n)$$

Ignoring constants & lower orders

$$\Rightarrow n^2$$

∴ Time complexity of Bruteforce approach ⇒ $O(n^2)$

2) DIVIDE AND CONQUER APPROACH:



1. Recursive calls:

elements are divided into halves recursively resulting into $O(\log n)$ levels of Recursion.

2. Merging & counting: $O(n)$

Total complexity ⇒ $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Applying master th^m,

$a = 2, b = 2, d = 1$

$\therefore a = b^d$

$T(n) = O(n \log n)$

PROGRAM:

1) CSV GENERATOR:

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <string>
#include <algorithm>

const int NUM_STUDENTS = 100;
const int NUM_CHOICES = 5;
const int MIN_CODE = 3001;
const int MAX_CODE = 3005;

// Function to generate unique course choices
std::vector<int> generateUniqueCourseChoices() {
    std::vector<int> choices;
    // Fill the vector with possible course codes
    for (int code = MIN_CODE; code <= MAX_CODE; ++code) {
        choices.push_back(code);
    }

    // Shuffle the choices
    std::random_shuffle(choices.begin(), choices.end());

    // Return the first NUM_CHOICES elements
    return std::vector<int>(choices.begin(), choices.begin() + NUM_CHOICES);
}

// Function to get the next available filename
std::string getNextFilename() {
    int counter = 1;
    std::string filename;

    // Loop to find the next available filename
    while (true) {
        filename = "csv" + std::to_string(counter) + ".csv";
        std::ifstream file(filename); // Try to open the file
        if (!file) { // If the file does not exist
            break;
        }
        counter++;
    }

    return filename;
}

int main() {
    // Seed the random number generator
    std::srand(static_cast<unsigned int>(std::time(0)));

    // Generate a unique filename
    std::string filename = getNextFilename();

    // Create a CSV file
    std::ofstream outputFile(filename);

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

    // Write header to the CSV file
    outputFile << "StudentID,Choice1,Choice2,Choice3,Choice4,Choice5\n";

    // Generate course choices for 100 students
    for (int i = 0; i < NUM_STUDENTS; ++i) {
        outputFile << "Student_" << (i + 1);
        std::vector<int> choices = generateUniqueCourseChoices();
        for (int j = 0; j < NUM_CHOICES; ++j) {
            outputFile << "," << choices[j];
        }
        outputFile << "\n";
    }

    // Close the file
    outputFile.close();

    std::cout << "Course choices have been written to " << filename << std::endl;

    return 0;
}
```

2) BRUTEFORCE :

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>

const int MIN_COURSE_CODE = 3001; // Minimum valid course code
const int MAX_COURSE_CODE = 3005; // Maximum valid course code

// Function to calculate the number of inversions in a list of course choices
int calculateInversions(const std::vector<int>& courseChoices) {
    int inversionCount = 0;
    for (size_t i = 0; i < courseChoices.size(); ++i) {
        for (size_t j = i + 1; j < courseChoices.size(); ++j) {
            if (courseChoices[i] > courseChoices[j] && courseChoices[i] != 0 && courseChoices[j] != 0) {
                inversionCount++;
            }
        }
    }
    return inversionCount;
}

// Function to validate course choices
bool validateCourseChoices(const std::vector<int>& courseChoices) {
    for (int choice : courseChoices) {
        if (choice < 0 || choice == 0 || choice < MIN_COURSE_CODE || choice > MAX_COURSE_CODE) {
            return false;
        }
    }
    return true;
}

int main() {
    std::ifstream inputFile("csv8.csv"); // Input file name
    std::string currentLine;
    std::map<int, int> inversionStats; // Map to count students by inversion count
    int invalidChoiceCount = 0; // Counter for invalid course choices

    // Skip the header line
    std::getline(inputFile, currentLine);

    while (std::getline(inputFile, currentLine)) {
        std::stringstream lineStream(currentLine);
        std::string studentID;
        std::vector<int> courseChoices;
        int courseChoice;

        std::getline(lineStream, studentID, ','); // Read the Student ID

        // Read course choices
        while (lineStream >> courseChoice) {
            courseChoices.push_back(courseChoice);
            if (lineStream.peek() == ',') lineStream.ignore(); // Skip the comma
        }

        // Validate course choices
        if (!validateCourseChoices(courseChoices)) {
            invalidChoiceCount++;
            continue; // Skip to the next student if choices are invalid
        }

        // Calculate inversions for valid choices
        int inversions = calculateInversions(courseChoices);

        // Record the number of students with this inversion count
        inversionStats[inversions]++;
    }

    inputFile.close();

    // Output the inversion statistics
    std::cout << "BRUTEFORCE APPROACH "<<"\n"<<"Number of students by inversion count:\n";
    for (const auto& entry : inversionStats) {
        std::cout << "Inversions: " << entry.first << " -> Number of Students: " << entry.second << "\n";
    }

    std::cout << "-----------------------------------------" << "\n";

    // Output the count of students with invalid choices
    std::cout << "Number of students with invalid choices: " << invalidChoiceCount << std::endl;

    return 0;
}
```

3) DIVIDE AND CONQUER:

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>

const int MIN_CODE = 3001; // Minimum valid course code
const int MAX_CODE = 3005; // Maximum valid course code

// Function to merge two halves and count split inversions
int mergeAndCount(std::vector<int>& codes, int left, int mid, int right) {
    int leftPtr = left;       // Starting index for left subarray
    int rightPtr = mid + 1;   // Starting index for right subarray
    int mergeIdx = 0;         // Index for the merged array
    int splitInvCount = 0;
    std::vector<int> merged(right - left + 1); // Temporary array for merging

    // Merging the two halves
    while (leftPtr <= mid && rightPtr <= right) {
        if (codes[leftPtr] <= codes[rightPtr]) {
            merged[mergeIdx++] = codes[leftPtr++];
        } else {
            merged[mergeIdx++] = codes[rightPtr++];
            splitInvCount += (mid - leftPtr + 1); // Count split inversions
        }
    }

    // Copy remaining elements from the left half
    while (leftPtr <= mid) {
        merged[mergeIdx++] = codes[leftPtr++];
    }
    // Copy remaining elements from the right half
    while (rightPtr <= right) {
        merged[mergeIdx++] = codes[rightPtr++];
    }

    // Copy back the merged elements into the original array
    for (int i = 0; i < merged.size(); ++i) {
        codes[left + i] = merged[i];
    }

    return splitInvCount;
}

// Function to count inversions using the merge sort approach
int countInvs(std::vector<int>& codes, int left, int right) {
    int totalInvs = 0;
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively count inversions in the left half
        totalInvs += countInvs(codes, left, mid);
        // Recursively count inversions in the right half
        totalInvs += countInvs(codes, mid + 1, right);
        // Count split inversions while merging
        totalInvs += mergeAndCount(codes, left, mid, right);
    }
    return totalInvs;
}

// Function to validate course choices
bool isValid(const std::vector<int>& choices) {
    for (int choice : choices) {
        if (choice < 0 || choice == 0 || choice < MIN_CODE || choice > MAX_CODE) {
            return false; // Invalid choice
        }
    }
    return true; // All choices are valid
}
```

```cpp
int main() {
    std::ifstream inputFile("csv8.csv"); // Input file name
    std::string line;
    std::map<int, int> invStats; // Map to count students by inversion count
    int invalidCount = 0; // Counter for invalid choices

    // Skip the header line
    std::getline(inputFile, line);

    while (std::getline(inputFile, line)) {
        std::stringstream ss(line);
        std::string studentID;
        std::vector<int> choices;
        int choice;

        std::getline(ss, studentID, ','); // Read the Student ID

        // Read course choices
        while (ss >> choice) {
            choices.push_back(choice);
            if (ss.peek() == ',') ss.ignore(); // Skip the comma
        }

        // Validate course choices
        if (!isValid(choices)) {
            invalidCount++;
            continue; // Skip to the next student if choices are invalid
        }

        // Count inversions for valid choices
        int inversions = countInvs(choices, 0, choices.size() - 1);

        // Record the number of students with this inversion count
        invStats[inversions]++;
    }

    inputFile.close();

    // Output the inversion statistics
    std::cout << "DIVIDE AND CONQUER APPROACH "<<"\n"<<"Number of students by inversion count:\n";
    for (const auto& entry : invStats) {
        std::cout << "Inversions: " << entry.first << " -> Number of Students: " << entry.second << "\n";
    }

    std::cout << "-----------------------------------------" << "\n";

    // Output the count of students with invalid choices
    std::cout << "Number of students with invalid choices: " << invalidCount << std::endl;

    return 0;
}
```

OUTPUT:

CSV FILE 1 OUTPUT:

```
BRUTEFORCE APPROACH
Number of students by inversion count:
Inversions: 0 -> Number of Students: 1
Inversions: 1 -> Number of Students: 3
Inversions: 2 -> Number of Students: 8
Inversions: 3 -> Number of Students: 9
Inversions: 4 -> Number of Students: 20
Inversions: 5 -> Number of Students: 20
Inversions: 6 -> Number of Students: 16
Inversions: 7 -> Number of Students: 8
Inversions: 8 -> Number of Students: 11
Inversions: 9 -> Number of Students: 4
-----------------------------------------
Number of students with invalid choices: 0
```

```
DIVIDE AND CONQUER APPROACH
Number of students by inversion count:
Inversions: 0 -> Number of Students: 1
Inversions: 1 -> Number of Students: 3
Inversions: 2 -> Number of Students: 8
Inversions: 3 -> Number of Students: 9
Inversions: 4 -> Number of Students: 20
Inversions: 5 -> Number of Students: 20
Inversions: 6 -> Number of Students: 16
Inversions: 7 -> Number of Students: 8
Inversions: 8 -> Number of Students: 11
Inversions: 9 -> Number of Students: 4
-----------------------------------------
Number of students with invalid choices: 0
```

CSV FILE 2 OUTPUT:

```
BRUTEFORCE APPROACH
Number of students by inversion count:
Inversions: 0 -> Number of Students: 1
Inversions: 1 -> Number of Students: 4
Inversions: 2 -> Number of Students: 5
Inversions: 3 -> Number of Students: 13
Inversions: 4 -> Number of Students: 13
Inversions: 5 -> Number of Students: 18
Inversions: 6 -> Number of Students: 22
Inversions: 7 -> Number of Students: 16
Inversions: 8 -> Number of Students: 7
Inversions: 9 -> Number of Students: 1
-----------------------------------------
Number of students with invalid choices: 0
```

```
DIVIDE AND CONQUER APPROACH
Number of students by inversion count:
Inversions: 0 -> Number of Students: 1
Inversions: 1 -> Number of Students: 4
Inversions: 2 -> Number of Students: 5
Inversions: 3 -> Number of Students: 13
Inversions: 4 -> Number of Students: 13
Inversions: 5 -> Number of Students: 18
Inversions: 6 -> Number of Students: 22
Inversions: 7 -> Number of Students: 16
Inversions: 8 -> Number of Students: 7
Inversions: 9 -> Number of Students: 1
-----------------------------------------
Number of students with invalid choices: 0
```

CSV FILE 3 OUTPUT:

```
BRUTEFORCE APPROACH
Number of students by inversion count:
Inversions: 1 -> Number of Students: 3
Inversions: 2 -> Number of Students: 5
Inversions: 3 -> Number of Students: 9
Inversions: 4 -> Number of Students: 12
Inversions: 5 -> Number of Students: 18
Inversions: 6 -> Number of Students: 17
Inversions: 7 -> Number of Students: 14
Inversions: 8 -> Number of Students: 11
Inversions: 9 -> Number of Students: 11
-----------------------------------------
Number of students with invalid choices: 0
```

```
DIVIDE AND CONQUER APPROACH
Number of students by inversion count:
Inversions: 1 -> Number of Students: 3
Inversions: 2 -> Number of Students: 5
Inversions: 3 -> Number of Students: 9
Inversions: 4 -> Number of Students: 12
Inversions: 5 -> Number of Students: 18
Inversions: 6 -> Number of Students: 17
Inversions: 7 -> Number of Students: 14
Inversions: 8 -> Number of Students: 11
Inversions: 9 -> Number of Students: 11
-----------------------------------------
Number of students with invalid choices: 0
```

CSV FILE 4 OUTPUT:

```
BRUTEFORCE APPROACH
Number of students by inversion count:
Inversions: 0 -> Number of Students: 2
Inversions: 1 -> Number of Students: 10
Inversions: 2 -> Number of Students: 13
Inversions: 3 -> Number of Students: 17
Inversions: 4 -> Number of Students: 21
Inversions: 5 -> Number of Students: 9
Inversions: 6 -> Number of Students: 14
Inversions: 7 -> Number of Students: 6
Inversions: 8 -> Number of Students: 4
Inversions: 9 -> Number of Students: 4
-----------------------------------------
Number of students with invalid choices: 0
```

```
DIVIDE AND CONQUER APPROACH
Number of students by inversion count:
Inversions: 0 -> Number of Students: 2
Inversions: 1 -> Number of Students: 10
Inversions: 2 -> Number of Students: 13
Inversions: 3 -> Number of Students: 17
Inversions: 4 -> Number of Students: 21
Inversions: 5 -> Number of Students: 9
Inversions: 6 -> Number of Students: 14
Inversions: 7 -> Number of Students: 6
Inversions: 8 -> Number of Students: 4
Inversions: 9 -> Number of Students: 4
-----------------------------------------
Number of students with invalid choices: 0
```

CSV FILE 5 OUTPUT:

```
BRUTEFORCE APPROACH
Number of students by inversion count:
Inversions: 1 -> Number of Students: 3
Inversions: 2 -> Number of Students: 5
Inversions: 3 -> Number of Students: 12
Inversions: 4 -> Number of Students: 11
Inversions: 5 -> Number of Students: 12
Inversions: 6 -> Number of Students: 17
Inversions: 7 -> Number of Students: 14
Inversions: 8 -> Number of Students: 7
Inversions: 9 -> Number of Students: 1
----------------------------------------
Number of students with invalid choices: 18
```

```
DIVIDE AND CONQUER APPROACH
Number of students by inversion count:
Inversions: 1 -> Number of Students: 3
Inversions: 2 -> Number of Students: 5
Inversions: 3 -> Number of Students: 12
Inversions: 4 -> Number of Students: 11
Inversions: 5 -> Number of Students: 12
Inversions: 6 -> Number of Students: 17
Inversions: 7 -> Number of Students: 14
Inversions: 8 -> Number of Students: 7
Inversions: 9 -> Number of Students: 1
----------------------------------------
Number of students with invalid choices: 18
```

CSV FILE 6 OUTPUT:

```
BRUTEFORCE APPROACH
Number of students with inversion counts:
Inversions: 0 -> Number of Students: 1
Inversions: 1 -> Number of Students: 3
Inversions: 2 -> Number of Students: 8
Inversions: 3 -> Number of Students: 8
Inversions: 4 -> Number of Students: 18
Inversions: 5 -> Number of Students: 16
Inversions: 6 -> Number of Students: 15
Inversions: 7 -> Number of Students: 6
Inversions: 8 -> Number of Students: 10
Inversions: 9 -> Number of Students: 4
----------------------------------------
Number of students with invalid choices: 11
```

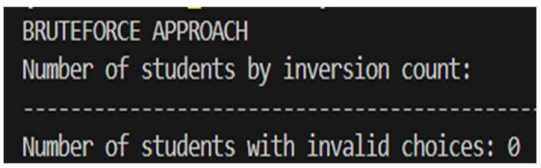
```
DIVIDE AND CONQUER APPROACH
Number of students with inversion counts:
Inversions: 0 -> Number of Students: 1
Inversions: 1 -> Number of Students: 3
Inversions: 2 -> Number of Students: 8
Inversions: 3 -> Number of Students: 8
Inversions: 4 -> Number of Students: 18
Inversions: 5 -> Number of Students: 16
Inversions: 6 -> Number of Students: 15
Inversions: 7 -> Number of Students: 6
Inversions: 8 -> Number of Students: 10
Inversions: 9 -> Number of Students: 4
----------------------------------------
Number of students with invalid choices: 11
```

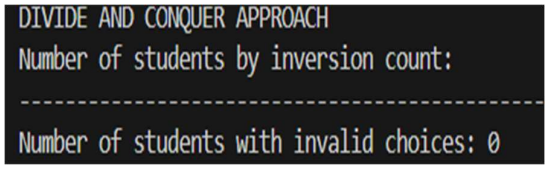
CSV FILE 7 OUTPUT:

```
BRUTEFORCE APPROACH
Number of students by inversion count:
Inversions: 0 -> Number of Students: 1
Inversions: 1 -> Number of Students: 3
Inversions: 2 -> Number of Students: 7
Inversions: 3 -> Number of Students: 9
Inversions: 4 -> Number of Students: 18
Inversions: 5 -> Number of Students: 16
Inversions: 6 -> Number of Students: 15
Inversions: 7 -> Number of Students: 8
Inversions: 8 -> Number of Students: 10
Inversions: 9 -> Number of Students: 4
----------------------------------------
Number of students with invalid choices: 9
```

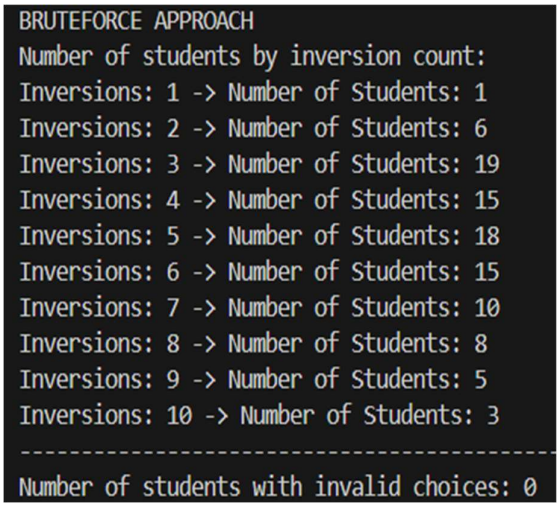
```
DIVIDE AND CONQUER APPROACH
Number of students by inversion count:
Inversions: 0 -> Number of Students: 1
Inversions: 1 -> Number of Students: 3
Inversions: 2 -> Number of Students: 7
Inversions: 3 -> Number of Students: 9
Inversions: 4 -> Number of Students: 18
Inversions: 5 -> Number of Students: 16
Inversions: 6 -> Number of Students: 15
Inversions: 7 -> Number of Students: 8
Inversions: 8 -> Number of Students: 10
Inversions: 9 -> Number of Students: 4
----------------------------------------
Number of students with invalid choices: 9
```

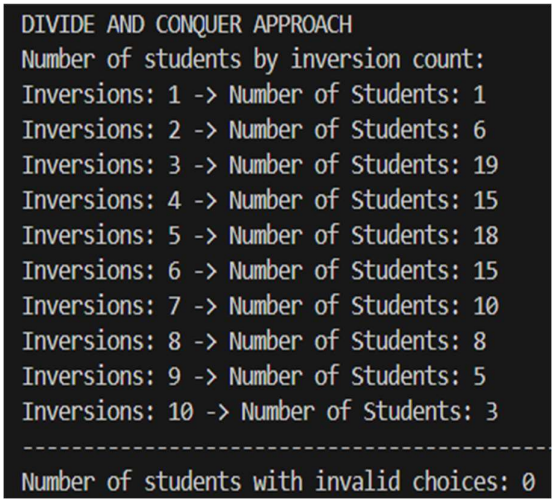
CSV FILE 8 OUTPUT:



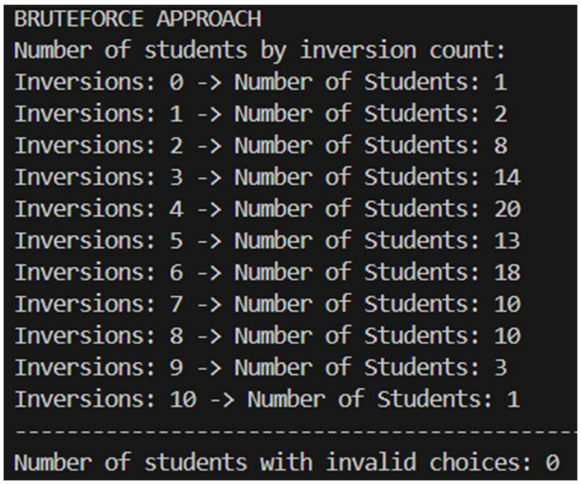


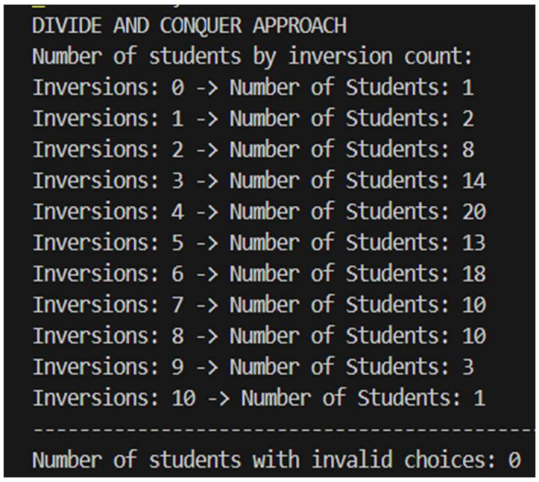
Csv file 9 output:





Csv file 10 output:





CONCLUSION:

In conclusion, we looked at two ways to count inversions in course-code choices: the brute-force method and the divide-and-conquer method. The brute-force method is simple but slow, with a time complexity of (O(n^2)), which makes it inefficient for larger groups. On the other hand, the divide-and-conquer method is faster, operating at (O(n log n)), which makes it a better choice for counting inversions as the number of students grows. Overall, the divide-and-conquer approach is the best option for this task, as it is both faster and more efficient.

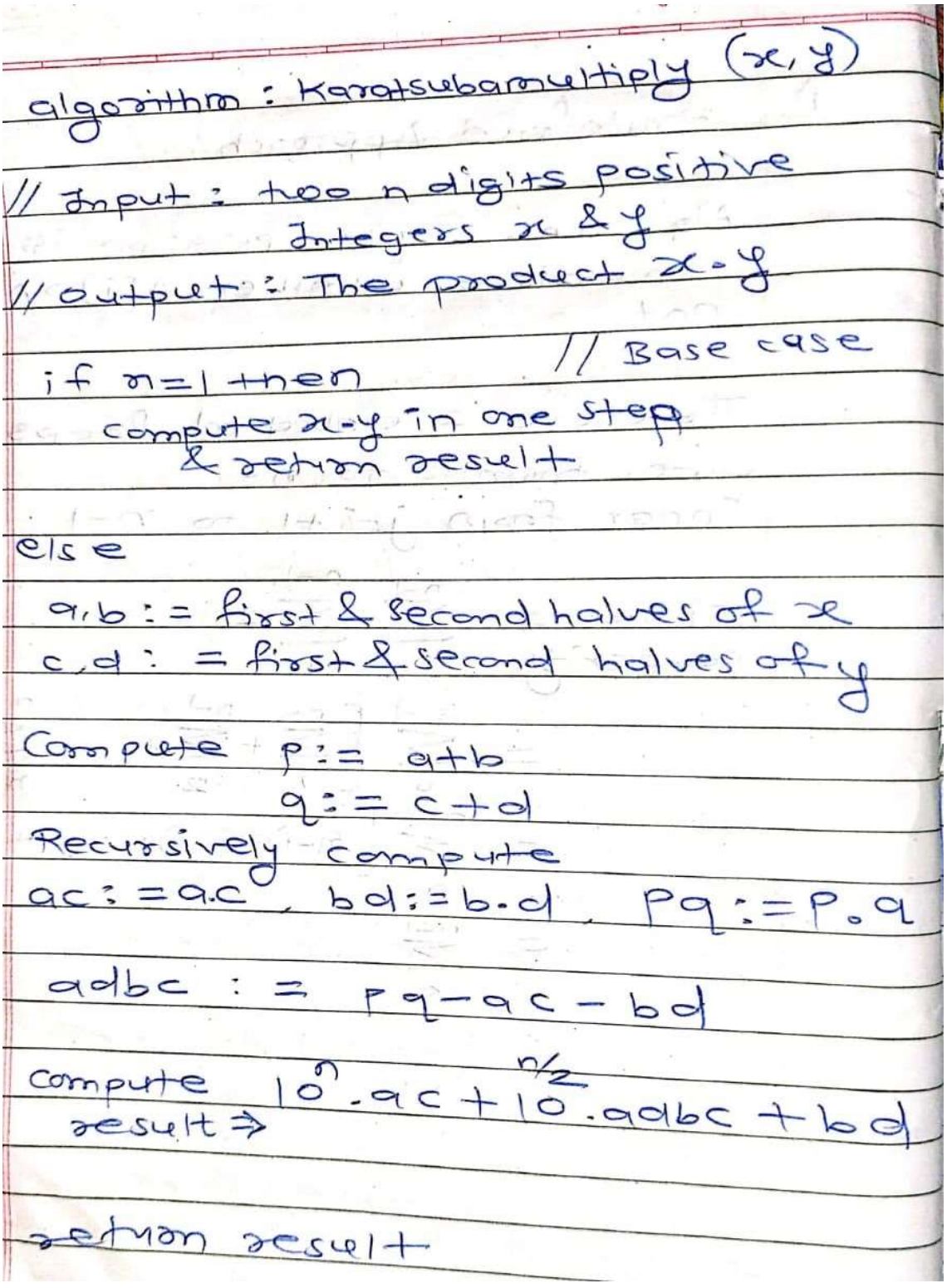
EXPERIMENTAL TASK-2:

AIM: Consider large integers of size 10, 50, 100, 500 and 1000 digits. Write integer multiplication program .Write integer multiplication program using divide and conquer technique.

THEORY:

The Karatsuba multiplication algorithm is an efficient method for multiplying large integers, significantly improving upon the traditional approach. By using a divide-and-conquer strategy, it breaks each number into two halves and computes fewer multiplications than classical methods. Instead of four multiplications, Karatsuba only requires three, which reduces the overall complexity and speeds up the process. This makes it particularly useful for handling very large integers, such as those with hundreds or thousands of digits, allowing for faster calculations in applications like cryptography and numerical analysis.

ALGORITHM:



algorithm : Karatsubamultiply (x, y)

// Input : two n digits positive Integers x & y

// Output : The product x.y

if n=1 then // Base case

compute x.y in one step & return result

else

a, b := first & second halves of x

c, d := first & second halves of y

Compute p := a+b

q := c+d

Recursively compute

ac := a.c , bd := b.d , pq := p.q

adbc := pq - ac - bd

Compute $10^{n}.ac + 10^{n/2}.adbc + bd$

result ⇒

return result

TEST CASES:

Test case	x	y	Result (x.y)
1	5	6	30
2	789	568	448152
3	123456789	987654321	121932631112635264
4	-123456789	-123456789	15241578750190520
5	985612348	-1246879512	-1228939843495414272
6	-1234	53212678	-65664444652
7	15972531597	824682465	548128971992305792
8	0	523123678	0
9	1232131134	0	0
10	0	0	0

TIME COMPLEXITY:

The algorithm split each of 2 number into 2 halves.
then recursively compute product of Three smaller numbers instead of 4 which reduces no. of recursive calls.

recursive step yield following:
1) ac 2) bd 3) (a+b).(c+d)

Time complexity $T(n) = 3T(n/2) + O(n)$

add^n & multiplic needed for final ans Construction.

By masters th^m,

$a=3, b=2, d=1$

$\therefore a > b^d$

$T(n) = O(n^{\log_b a})$

$= O(n^{\log_2 3})$

$= O(n^{1.585})$

In this way, karatsuba multiplication reduce complexity from n^2 to $n^{1.585}$

PROGRAM:

```cpp
#include <iostream>
#include <cmath>
#include <string>
#include <algorithm>
#include <vector>
#include <utility>

// Function to extract the first or second half of the digits of a number
long long getHalfDigits(long long number, int totalDigits, bool isSecondHalf) {
    long long divisor = pow(10, totalDigits / 2);
    return isSecondHalf ? number % divisor : number / divisor;
}

// Recursive function to perform Karatsuba multiplication
long long karatsubaMultiply(long long x, long long y, int totalDigits) {
    if (totalDigits == 1) {
        return x * y;
    }

    if (totalDigits % 2 != 0) {
        totalDigits++;
    }

    long long a = getHalfDigits(x, totalDigits, false);
    long long b = getHalfDigits(x, totalDigits, true);
    long long c = getHalfDigits(y, totalDigits, false);
    long long d = getHalfDigits(y, totalDigits, true);

    long long prod1 = karatsubaMultiply(a, c, totalDigits / 2);
    long long prod2 = karatsubaMultiply(b, d, totalDigits / 2);
    long long middleProduct = karatsubaMultiply(a + b, c + d, totalDigits / 2) - prod1 - prod2;

    return prod1 * pow(10, totalDigits) + middleProduct * pow(10, totalDigits / 2) + prod2;
}

// Function to start the multiplication process and handle sign
long long multiply(long long x, long long y) {
    int size1 = std::to_string(x).length();
    int size2 = std::to_string(y).length();
    int totalDigits = std::max(size1, size2);

    if (totalDigits % 2 != 0) {
        totalDigits++;
    }

    bool isNegativeResult = (x < 0) ^ (y < 0);
    long long result = karatsubaMultiply(abs(x), abs(y), totalDigits);

    return isNegativeResult ? -1*result : result;
}

int main() {
    int numTestCases;
    std::cout << "Enter the number of test cases: ";
    std::cin >> numTestCases;

    for (int i = 0; i < numTestCases; ++i) {
        long long x, y;
        std::cout << "Enter two numbers (x and y) for test case " << i + 1 << ": "<<std::endl;
        std::cin >> x >> y;
        long long result = multiply(x, y);
        std::cout << "Result of multiplication is "<< result << std::endl;
    }

    return 0;
}
```

OUTPUT:

```
Enter the number of test cases: 10
Enter two numbers (x and y) for test case 1:
5 6
Result of multiplication is 30
Enter two numbers (x and y) for test case 2:
789 568
Result of multiplication is 448152
Enter two numbers (x and y) for test case 3:
123456789 987654321
Result of multiplication is 121932631112635264
Enter two numbers (x and y) for test case 4:
-123456789 -123456789
Result of multiplication is 15241578750190520
Enter two numbers (x and y) for test case 5:
985612348 -1246879512
Result of multiplication is -1228939843495414272
Enter two numbers (x and y) for test case 6:
-1234 53212678
Result of multiplication is -65664444652
Enter two numbers (x and y) for test case 7:
1597531597 8246824651
Result of multiplication is 548128971992305792
```

```
Result of multiplication is 548128971992305792
Enter two numbers (x and y) for test case 8:
0 5231231678
Result of multiplication is 0
Enter two numbers (x and y) for test case 9:
1232131134 0
Result of multiplication is 0
Enter two numbers (x and y) for test case 10:
0 0
0
```

CONCLUSION:

In conclusion, the integer multiplication program developed using the divide and conquer technique, specifically the Karatsuba algorithm, effectively demonstrates improved efficiency in multiplying large numbers. By reducing the time complexity from O(n^2) to approximately O(n^(1.585)), this method simplifies the multiplication process through recursive decomposition. This implementation not only accommodates large integers beyond standard data type limits but also serves practical applications in areas like cryptography and scientific computing. Future enhancements could further optimize the algorithm and integrate it with libraries for arbitrary precision arithmetic.

THE END