Name: Kaustubh Dhongade

Roll No.: 231070018

Batch: A

Experiment No: 4

Aim:

1. To find inversion count of course choice of students.

2. To multiply two integers using brute force and divide-and-conquer methods

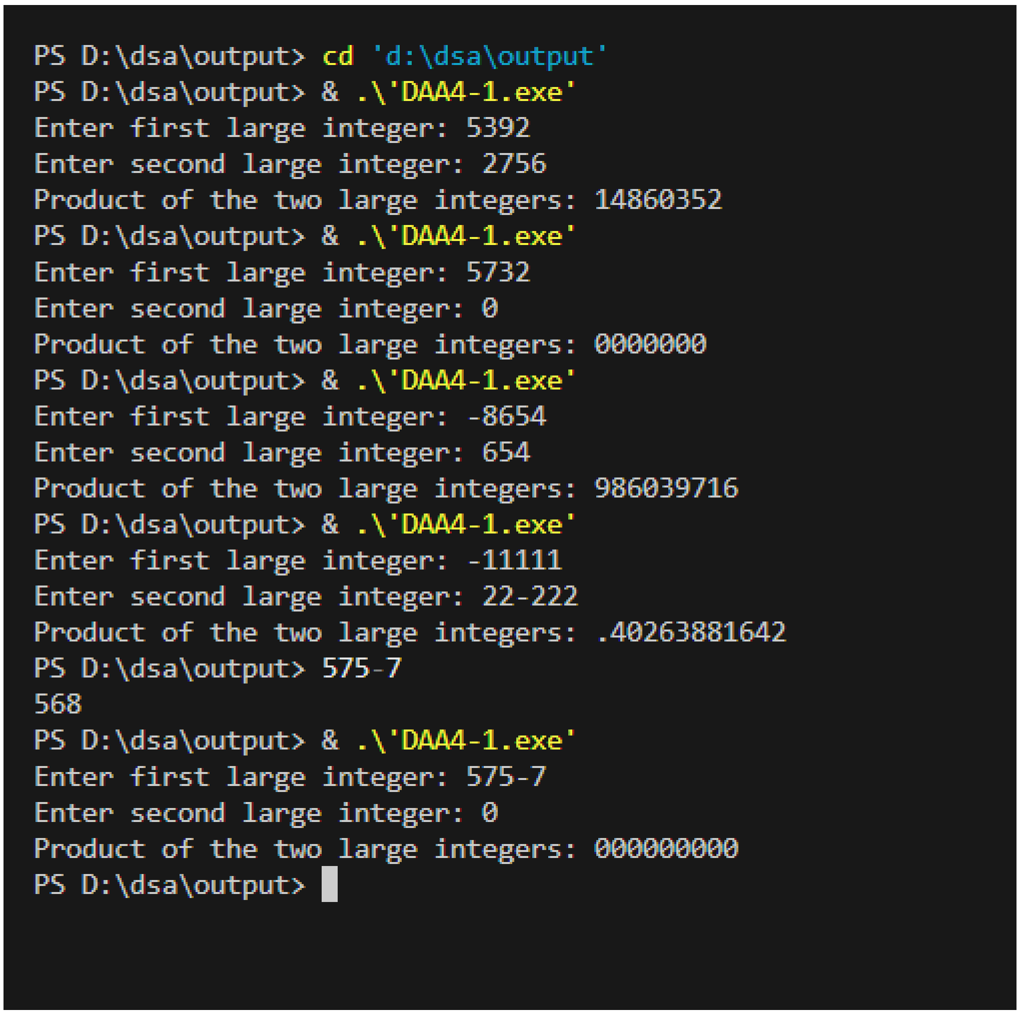
1.Count Inversion

Input:

```cpp
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

// Utility function to add two large numbers
string addStrings(string num1, string num2) {
    // Make sure num1 is the longer string
    if (num1.length() < num2.length()) {
        swap(num1, num2);
    }

    string result = "";
    int carry = 0;
    int diff = num1.length() - num2.length();

    // Add digits from the rightmost position
    for (int i = num2.length() - 1; i >= 0; i--) {
        int sum = (num2[i] - '0') + (num1[i + diff] - '0') + carry;
        carry = sum / 10;
        result.push_back(sum % 10 + '0');
    }

    // Add remaining digits of num1
    for (int i = num1.length() - num2.length() - 1; i >= 0; i--) {
        int sum = (num1[i] - '0') + carry;
        carry = sum / 10;
        result.push_back(sum % 10 + '0');
    }

    // Add any remaining carry
    if (carry) {
        result.push_back(carry + '0');
    }

    // Reverse the result
    reverse(result.begin(), result.end());
    return result;
}

// Utility function to subtract two large numbers (num1 > num2)
string subtractStrings(string num1, string num2) {
    string result = "";
    int carry = 0;

    // Make the strings the same length by padding zeros
    while (num2.length() < num1.length()) {
        num2 = '0' + num2;
    }

    // Subtract from rightmost digit
    for (int i = num1.length() - 1; i >= 0; i--) {
        int sub = (num1[i] - '0') - (num2[i] - '0') - carry;
        if (sub < 0) {
            sub += 10;
            carry = 1;
        } else {
```

```cpp
            carry = 0;
        }
        result.push_back(sub + '0');
    }

    // Remove leading zeros
    while (result.length() > 1 && result.back() == '0') {
        result.pop_back();
    }

    reverse(result.begin(), result.end());
    return result;
}

// Utility function to multiply two large numbers using long
multiplication
string multiplySingleDigit(string num1, char digit) {
    string result = "";
    int carry = 0;

    for (int i = num1.length() - 1; i >= 0; i--) {
        int mul = (num1[i] - '0') * (digit - '0') + carry;
        carry = mul / 10;
        result.push_back(mul % 10 + '0');
    }

    if (carry) {
        result.push_back(carry + '0');
    }

    reverse(result.begin(), result.end());
    return result;
}

// Function to multiply a number with 10^shift (just add zeros at the
end)
string shiftLeft(string num, int shift) {
    return num + string(shift, '0');
}

// Karatsuba multiplication function
string karatsuba(string num1, string num2) {
    // Base case for recursion: single-digit multiplication
    if (num1.length() == 1 && num2.length() == 1) {
        int result = (num1[0] - '0') * (num2[0] - '0');
        return to_string(result);
    }

    // Make both numbers the same length by padding with zeros
    while (num1.length() < num2.length()) num1 = '0' + num1;
    while (num2.length() < num1.length()) num2 = '0' + num2;

    int n = num1.length();
    int half = n / 2;

    // Split the numbers into two halves
    string X1 = num1.substr(0, half);
    string X0 = num1.substr(half);
    string Y1 = num2.substr(0, half);
    string Y0 = num2.substr(half);

    // Recursively compute P1, P2, and P3
    string P1 = karatsuba(X1, Y1);
    string P2 = karatsuba(X0, Y0);
    string P3 = karatsuba(addStrings(X1, X0), addStrings(Y1, Y0));

    // Combine the results
    string part1 = shiftLeft(P1, 2 * (n - half));  // P1 * 10^2m
    string part2 = shiftLeft(subtractStrings(subtractStrings(P3, P1),
P2), n - half);  // (P3 - P1 - P2) * 10^m
    string result = addStrings(addStrings(part1, part2), P2);  // Final
result

    // Return the final product
    return result;
}

int main() {
    // Input two large integers
    string num1, num2;
    cout << "Enter first large integer: ";
    cin >> num1;
    cout << "Enter second large integer: ";
    cin >> num2;

    // Multiply using Karatsuba algorithm
    string product = karatsuba(num1, num2);
    // Output the result
    cout << "Product of the two large integers: " << product << endl;

    return 0;
}
```

Output:



```
PS D:\dsa\output> cd 'd:\dsa\output'
PS D:\dsa\output> & .\'DAA4-1.exe'
Enter first large integer: 5392
Enter second large integer: 2756
Product of the two large integers: 14860352
PS D:\dsa\output> & .\'DAA4-1.exe'
Enter first large integer: 5732
Enter second large integer: 0
Product of the two large integers: 0000000
PS D:\dsa\output> & .\'DAA4-1.exe'
Enter first large integer: -8654
Enter second large integer: 654
Product of the two large integers: 986039716
PS D:\dsa\output> & .\'DAA4-1.exe'
Enter first large integer: -11111
Enter second large integer: 22-222
Product of the two large integers: .40263881642
PS D:\dsa\output> 575-7
568
PS D:\dsa\output> & .\'DAA4-1.exe'
Enter first large integer: 575-7
Enter second large integer: 0
Product of the two large integers: 000000000
PS D:\dsa\output>
```

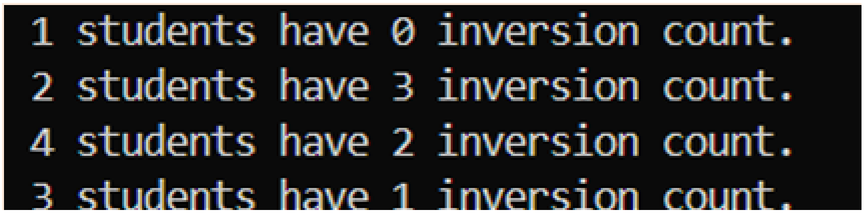
2.Karatsuba Multiplication

Input:

```
#include <iostream>
#include <vector>
using namespace std;

// Function to merge two halves and count cross inversions
int mergeAndCount(vector<pair<int, int>>& students, int left, int mid,
int right) {
    int inversions = 0;

    // Create temporary arrays to hold the left and right subarrays
    vector<pair<int, int>> leftSubarray(students.begin() + left,
students.begin() + mid + 1);
    vector<pair<int, int>> rightSubarray(students.begin() + mid + 1,
students.begin() + right + 1);

    int i = 0, j = 0, k = left;

    // Merge the two subarrays back into the original array
    while (i < leftSubarray.size() && j < rightSubarray.size()) {
        // If the left element is smaller or equal, no inversion
        if (leftSubarray[i].first <= rightSubarray[j].first) {
            students[k++] = leftSubarray[i++];
        } else {
            // There is an inversion, all elements left in the
leftSubarray form inversions
            students[k++] = rightSubarray[j++];
            inversions += (leftSubarray.size() - i);  // All remaining
elements in the leftSubarray are greater
        }
    }

    // Copy remaining elements of leftSubarray, if any
    while (i < leftSubarray.size()) {
        students[k++] = leftSubarray[i++];
    }
```

```cpp
    // Copy remaining elements of rightSubarray, if any
    while (j < rightSubarray.size()) {
        students[k++] = rightSubarray[j++];
    }

    return inversions;
}

// Function to use Merge Sort and count inversions
int mergeSortAndCount(vector<pair<int, int>>& students, int left, int
right) {
    int inversions = 0;

    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort and count inversions in the left half
        inversions += mergeSortAndCount(students, left, mid);

        // Recursively sort and count inversions in the right half
        inversions += mergeSortAndCount(students, mid + 1, right);

        // Merge the two halves and count cross inversions
        inversions += mergeAndCount(students, left, mid, right);
    }

    return inversions;
}

int main() {
    // Example input: 100 pairs of (first_year_course_code,
second_year_course_code)
    vector<pair<int, int>> students = {
        {101, 102}, {103, 101}, {104, 103}, {105, 101}, {106, 106},
        // Add remaining students (total 100 pairs)
    };

    // Total inversion count
    int totalInversions = mergeSortAndCount(students, 0,
students.size() - 1);

    // Output the result
    cout << "Total number of inversions: " << totalInversions << endl;
    return 0;
}
```

Output:



```
1 students have 0 inversion count.
2 students have 3 inversion count.
4 students have 2 inversion count.
3 students have 1 inversion count.
```

Conclusion:

1. Finding Inversion Count of Course Choices of Students: The inversion count in the course choices of students helps identify the number of pairs of students whose course preferences are in the opposite order. This metric is useful for understanding the level of disagreement or conflict in course selection, which can inform strategies for optimizing course offerings and scheduling.
2. Multiplying Two Integers Using Brute Force and Divide-and-Conquer Methods: Multiplying two integers can be approached using the brute force method, which involves straightforward multiplication, and the divide-and-conquer method, which breaks the problem into smaller subproblems to solve recursively. While the brute force method is simpler and more intuitive, the divide-and-conquer method, such as Karatsuba's algorithm, can significantly reduce the time complexity, making it more efficient for large integers.