



**Symbiosis Institute of Technology, Pune**

**Department of Computer Science and Engineering**

**Academic Year 2025-26**

**Design and Analysis of Algorithms– Lab**

**Batch 2023-27 - Sem V**

**Lab Assignment No:- 2**

<b>Lab Assignment No:- 2</b>	
<b>Name of Student</b>	Deepti Pal
<b>PRN No.</b>	23070122081
<b>Batch</b>	TY CSE (2023-27)
<b>Class</b>	A3
<b>Academic Year &amp; Semester</b>	2025-26, 3 <sup>rd</sup> year, 5 <sup>th</sup> Semester
<b>Date of Submission</b>	3 <sup>rd</sup> August 2025
<b>Title of Assignment:</b>	<p>WAP to perform Quick sort, Merge sort and display the partial pass-wise sorting done.</p> <p><b>Problem Given: Sorting Transaction Data</b></p> <p><b>Scenario:</b> A financial institution is analysing its customers' spending behaviour. To identify high-value transactions and spending trends, they need to <b>sort transaction records by transaction amount</b> in ascending order.</p> <p>You are required to implement and compare <b>Quick Sort</b> and <b>Merge Sort</b> algorithms to perform this task efficiently. Additionally, track how sorting progresses over multiple passes and measure the performance of each algorithm.</p> <p><b>Input Specifications:</b></p> <ul style="list-style-type: none"><li>• Generate a dataset of at least <b>50 transaction records</b>.</li></ul>

	<ul style="list-style-type: none"><li>Each transaction record must include:<ul style="list-style-type: none"><li><b>Transaction ID</b> (e.g., TXN5238, unique alphanumeric ID)</li><li><b>Customer Name</b> (e.g., randomly generated)</li><li><b>Transaction Amount</b> (a float value between ₹100.00 and ₹1,00,000.00)</li><li><b>Timestamp</b> (in YYYY-MM-DD HH:MM:SS format)</li></ul></li></ul> <p><b>Sorting Requirement:</b></p> <ul style="list-style-type: none"><li><b>Primary Key:</b> Sort based on <b>Transaction Amount</b> (lowest to highest).</li><li><b>Stability Check (for Merge Sort):</b> If two transactions have the same amount, maintain their original input order.</li></ul> <p><b>Implementation Tasks:</b></p> <ol style="list-style-type: none"><li><b>Implement Quick Sort and Merge Sort</b> on the transaction list.</li><li><b>Track the number of key comparisons</b> made during sorting.</li><li><b>Display partial pass-wise outputs:</b> For each algorithm, show the list after <b>at least the first 5 key passes/recursive stages</b>.</li></ol> <p><b>Performance Metrics (Display):</b></p> <ul style="list-style-type: none"><li><b>Number of comparisons</b> made by each algorithm.</li><li><b>Time taken</b> to sort the data (in milliseconds or microseconds).</li><li><b>Sorted list</b> by transaction amount.</li></ul> <p><b>Constraints &amp; Validation Rules:</b></p> <ul style="list-style-type: none"><li>All Transaction IDs must be unique (prefix with TXN + random digits).</li><li>Transaction amounts should be valid floating-point numbers (₹100.00 – ₹1,00,000.00).</li><li>Timestamps should be properly formatted and represent valid dates within the last 30 days.</li></ul>
	<p><b>Sample Output Format (for Pass-Wise Display):</b></p> <p>--- Quick Sort Pass 1 --- [500.25, 300.10, 700.00, 950.45, 120.00, 850.35]</p> <p>--- Quick Sort Pass 2 --- [120.00, 300.10, 500.25, 700.00, 850.35, 950.45]</p> <p>... Total Comparisons: 72 Time Taken: 0.0023 seconds</p>
<b>Theory: (Handwritten)</b>	<ol style="list-style-type: none"><li>Apply Quick Sort, Merge on 10 input items and display the partial pass-wise sorting done.</li><li>Explain time complexities of both algorithms.</li></ol>

**Source code  
(Implementation  
Screenshot)**

```
#include<iostream>
#include<fstream>
#include<sstream>
#include<string>
#include<ctime>
#include<vector>
#include<chrono>
#include<iomanip>
using namespace std;

time_t parseTimestamp(const string &ts) {
    struct tm tmStruct = {};
    strptime(ts.c_str(), "%Y-%m-%d %H:%M:%S", &tmStruct);
    return mktime(&tmStruct);
}

string formatTimeStamp(time_t t) {
    char time_r[20];
    strftime(time_r, sizeof(time_r), "%Y-%m-%d %H:%M:%S",
    localtime(&t));
    return string(time_r);
}

class Customer {
protected:
    int customerID;
    string customerName;
    double balance;

public:
    Customer() {}
    Customer (int id, string name, double balance) {
        customerID = id;
        customerName = name;
        this->balance = balance;
    }
};

class Transaction : public Customer {
private:
    int transactionID;
    double transactionAMt;
    time_t timeStamp;

public:
    Transaction() {}
    Transaction(int tid, int id, string name, double bal, double amt,
    time_t ts ) {
```

```

        transactionID = tid;
        customerID = id;
        customerName = name;
        balance = bal;
        transactionAMt = amt;
        timeStamp = ts;
    }

    int getTransactionID() const {
        return transactionID;
    }
    int getCustomerID() const {
        return customerID;
    }
    string getCustomerName() const {
        return customerName;
    }
    double getBalance() const {
        return balance;
    }
    double getTransactionAmt() const {
        return transactionAMt;
    }
    time_t getTimeStamp() const {
        return timeStamp;
    }
};

void swap(Transaction &a, Transaction &b) {
    Transaction temp = a;
    a = b;
    b = temp;
}

int partitions(vector<Transaction> &arr, int low, int high) {
    double pivot = arr[high].getTransactionAmt();
    int i = low - 1;
    for(int j = low; j < high; j++) {
        if(arr[j].getTransactionAmt() <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i+1], arr[high]);
    return i+1;
}

void quickSort(vector<Transaction> &arr, int low, int high, int &pass) {
    if(low < high) {
        int p = partitions(arr, low, high);

```

```

        cout << "Quick Sort Pass " << pass++ << ": ";
        for (auto &tx : arr) cout << tx.getTransactionID() << " ";
        cout << "\n";

        quickSort(arr, low, p-1, pass);
        quickSort(arr, p+1, high, pass);
    }
}

void merge(vector<Transaction> &arr, int left, int mid, int right, int &pass)
{
    int n1 = mid-left+1;
    int n2 = right-mid;

    vector<Transaction> L(n1), R(n2);
    for(int i=0; i<n1; i++) L[i] = arr[left+i];
    for(int j=0; j<n2; j++) R[j] = arr[mid+1+j];

    int i=0, j=0, k=left;
    while(i<n1 && j<n2) {
        if(L[i].getTransactionAmt() <= R[j].getTransactionAmt())
            arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while(i<n1) arr[k++] = L[i++];
    while(j<n2) arr[k++] = R[j++];

    cout << "Merge Sort Pass " << pass++ << ": ";
    for (auto &tx : arr) cout << tx.getTransactionID() << " ";
    cout << "\n";
}

void mergeSort(vector<Transaction> &arr, int left, int right, int &pass) {
    if(left < right) {
        int mid = (left+right)/2;
        mergeSort(arr, left, mid, pass);
        mergeSort(arr, mid+1, right, pass);
        merge(arr, left, mid, right, pass);
    }
}

void writeFile(const string &f, const vector<Transaction> &arr) {
    ofstream file(f);
    if(!file.is_open()) {
        cerr << "File is not opening!" << endl;
        return ;
    }
}

```

```

        file <<
"TransactionID,CustomerID,CustomerName,Balance,TransactionAmount,T
imeStamp\n";
        for(auto &tx : arr) {
            file << tx.getTransactionID() << ","
                << tx.getCustomerID() << ","
                << tx.getCustomerName() << ","
                << tx.getBalance() << ","
                << tx.getTransactionAmt() << ","
                << formatTimeStamp(tx.getTimeStamp()) << "\n";
        }
        file.close();
    }

int main()
{
    string inputFile;
    cout << "Enter the inputFile: ";
    cin >> inputFile;
    ifstream file(inputFile);
    if(!file.is_open()) {
        cerr << "File is not opening!" << endl;
        return 1;
    }

    vector<Transaction> arr;

    string line;
    getline(file, line); // skip header

    while(getline(file, line)) {
        if(line.empty()) continue;
        stringstream ss(line);
        string tID, cID, cName, balstr, amtstr, tsstr;

        getline(ss, tID, ',');
        getline(ss, cID, ',');
        getline(ss, cName, ',');
        getline(ss, balstr, ',');
        getline(ss, amtstr, ',');
        getline(ss, tsstr);

        if (balstr.empty() || amtstr.empty()) continue;

        Transaction tx(stoi(tID), stoi(cID), cName, stod(balstr),
stod(amtstr), parseTimeStamp(tsstr));
        arr.emplace_back(tx);
    }
}

```

```

vector<Transaction> quickArr = arr;
vector<Transaction> mergeArr = arr;

int qPass = 1, mPass = 1;

// Quick Sort timing
auto start = chrono::high_resolution_clock::now();
quickSort(quickArr, 0, quickArr.size()-1, qPass);
auto end = chrono::high_resolution_clock::now();
double quickTime = chrono::duration<double>(end-start).count();

// Merge Sort timing
start = chrono::high_resolution_clock::now();
mergeSort(mergeArr, 0, mergeArr.size()-1, mPass);
end = chrono::high_resolution_clock::now();
double mergeTime = chrono::duration<double>(end-start).count();

string quickfile, mergefile;
cout << "Enter output file for quick sort: ";
cin >> quickfile;
cout << "Enter output file for mergesort: ";
cin >> mergefile;

writeFile(quickfile, quickArr);
writeFile(mergefile, mergeArr);

cout << "\nQuick Sort Time: " << quickTime << " secs\n";
cout << "Merge Sort Time: " << mergeTime << " secs\n";

file.close();
return 0;
}

```

Input file:

```
Draft version
Run Debug Stop Save {} Beautify submit Language
main.cpp ipSample_a2.txt mergeFile.txt quickFile.txt
1 TransactionID, CustomerID, CustomerName, Balance, TransactionAmt, Timestamp
2 1, 101, Alice, 5000, 200, 2025-08-01 10:23:45
3 2, 102, Bob, 3000, 150, 2025-07-31 18:45:10
4 3, 103, Charlie, 4500, 500, 2025-08-01 08:30:00
5 4, 104, Diana, 6000, 250, 2025-07-30 14:05:50
6 5, 105, Edward, 7000, 100, 2025-08-02 09:10:30
7
8
```

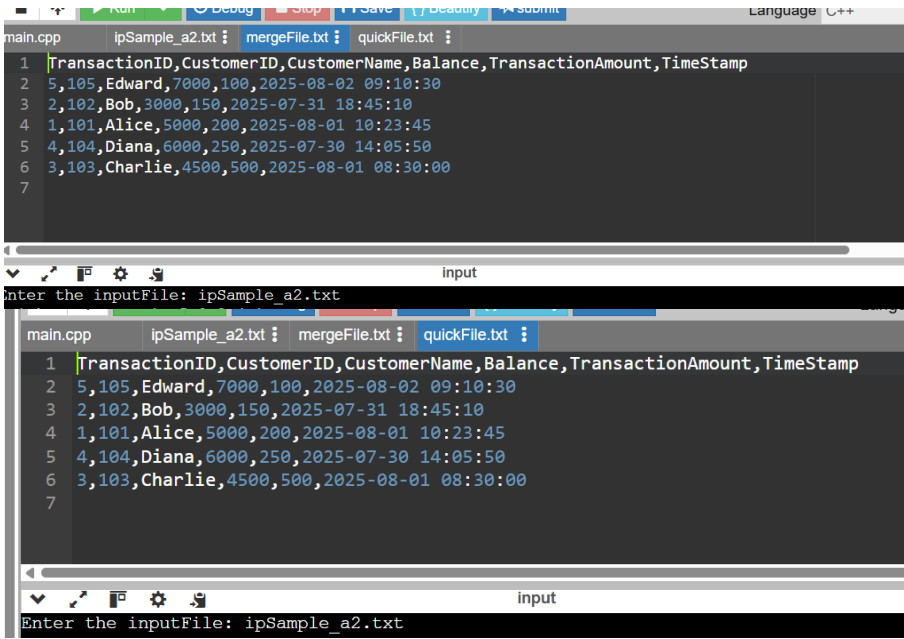
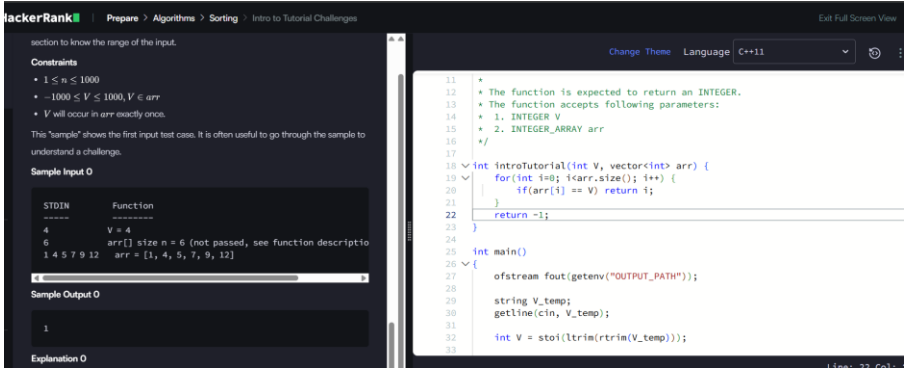
## Output Screenshots

```
Draft version
Run Debug Stop Save {} Beautify submit Language
main.cpp ipSample_a2.txt mergeFile.txt quickFile.txt
94
95 void quickSort(vector<Transaction> &arr, int low, int high, int &pass) {
96     if(low < high) {
97         int p = partitions(arr, low, high);
98
99         cout << "Quick Sort Pass " << pass++ << ": ";
100         for (auto &tx : arr) cout << tx.getTransactionID() << " ";
101         cout << "\n";
102
input
Enter the inputFile: ipSample_a2.txt
Quick Sort Pass 1: 5 2 3 4 1
Quick Sort Pass 2: 5 2 1 4 3
Quick Sort Pass 3: 5 2 1 4 3
Merge Sort Pass 1: 2 1 3 4 5
Merge Sort Pass 2: 2 1 3 4 5
Merge Sort Pass 3: 2 1 3 5 4
Merge Sort Pass 4: 5 2 1 4 3
Enter output file for quick sort: quickFile.txt
Enter output file for mergesort: mergeFile.txt

Quick Sort Time: 3.17e-05 secs
Merge Sort Time: 3.596e-05 secs

...Program finished with exit code 0
Press ENTER to exit console.
```



	 <pre> main.cpp   ipSample_a2.txt   mergeFile.txt   quickFile.txt   1 TransactionID, CustomerID, CustomerName, Balance, TransactionAmount, TimeStamp 2 5, 105, Edward, 7000, 100, 2025-08-02 09:10:30 3 2, 102, Bob, 3000, 150, 2025-07-31 18:45:10 4 1, 101, Alice, 5000, 200, 2025-08-01 10:23:45 5 4, 104, Diana, 6000, 250, 2025-07-30 14:05:50 6 3, 103, Charlie, 4500, 500, 2025-08-01 08:30:00 7 </pre> <p>input</p> <p>Enter the inputFile: ipSample_a2.txt</p>								
<p><b>Problems Solved from Hacker Rank (Minimum 4)</b></p>	<ol style="list-style-type: none"> <li>1. <a href="https://www.hackerrank.com/challenges/tutorial-intro/problem?isFullScreen=true">https://www.hackerrank.com/challenges/tutorial-intro/problem?isFullScreen=true</a></li> </ol> <p><b>Solution Screenshot:</b></p>  <p>section to know the range of the input.</p> <p><b>Constraints</b></p> <ul style="list-style-type: none"> <li>• <math>1 \leq n \leq 1000</math></li> <li>• <math>-1000 \leq V \leq 1000, V \in \text{arr}</math></li> <li>• <math>V</math> will occur in <math>\text{arr}</math> exactly once.</li> </ul> <p>This "sample" shows the first input test case. It is often useful to go through the sample to understand a challenge.</p> <p><b>Sample Input 0</b></p> <table border="1"> <thead> <tr> <th>STDIN</th> <th>Function</th> </tr> </thead> <tbody> <tr> <td>4</td> <td>V = 4</td> </tr> <tr> <td>6</td> <td>arr[] size n = 6 (not passed, see function description)</td> </tr> <tr> <td>1 4 5 7 9 12</td> <td>arr = [1, 4, 5, 7, 9, 12]</td> </tr> </tbody> </table> <p><b>Sample Output 0</b></p> <pre>1</pre> <p><b>Explanation 0</b></p> <pre> 11 * 12 * The function is expected to return an INTEGER. 13 * The function accepts following parameters: 14 * 1. INTEGER V 15 * 2. INTEGER_ARRAY arr 16 */ 17 18 int introTutorial(int V, vector&lt;int&gt; arr) { 19     for(int i=0; i&lt;arr.size(); i++) { 20         if(arr[i] == V) return i; 21     } 22     return -1; 23 } 24 25 int main() 26 { 27     ofstream fout(getenv("OUTPUT_PATH")); 28 29     string V_temp; 30     getline(cin, V_temp); 31 32     int V = stoi(trim(trim(V_temp))); 33 </pre>	STDIN	Function	4	V = 4	6	arr[] size n = 6 (not passed, see function description)	1 4 5 7 9 12	arr = [1, 4, 5, 7, 9, 12]
STDIN	Function								
4	V = 4								
6	arr[] size n = 6 (not passed, see function description)								
1 4 5 7 9 12	arr = [1, 4, 5, 7, 9, 12]								

## Congratulations

You solved this challenge. Would you like to challenge your friends? [f](#) [t](#) [in](#)

[Next Challenge](#)

✓ Test case 0

Compiler Message

Success

✓ Test case 1 [🔒](#)

✓ Test case 2 [🔒](#)

Input (stdin)

[Download](#)

```
1 4
2 6
3 1 4 5 7 9 12
```

Expected Output

[Download](#)

```
1 1
```

- <https://www.hackerrank.com/challenges/correctness-invariant/problem?isFullScreen=true>

### Solution Screenshot:

**HackerRank** | Prepare > Algorithms > Sorting > Correctness and the Loop Invariant

**Problem**

In the previous challenge, you wrote code to perform an Insertion Sort on an unsorted array. But how would you prove that the code is correct? I.e. how do you show that for any input your code will provide the right output?

**Loop Invariant**

In computer science, you could prove it formally with a loop invariant, where you state that a desired property is maintained in your loop. Such a proof is broken down into the following parts:

- Initialization: It is true (in a limited sense) before the loop runs.
- Maintenance: If it's true before an iteration of a loop, it remains true before the next iteration.
- Termination: It will terminate in a useful way once it is finished.

**Insertion Sort's Invariant**

Say, you have some InsertionSort code, where the outer loop goes through the whole array *A*:

```
for (let i = 1; i < A.length; i++){
  //insertion sort code
}
```

You could then state the following loop invariant:

**Submissions**

**Leaderboard**

**Code Editor**

```
1 #include <math.h>
2 #include <stdlib.h>
3 #include <assert.h>
4
5 using namespace std;
6
7 void insertionSort(int N, int arr[]) {
8   for (int i = 1; i < N; i++) {
9     int key = arr[i];
10    int j = i - 1;
11
12    // Move elements greater than key to the right
13    while (j >= 0 && arr[j] > key) {
14      arr[j + 1] = arr[j];
15      j--;
16    }
17
18    // Insert the key at its correct position
19    arr[j + 1] = key;
20
21    // Loop Invariant: arr[0..i] is sorted after each iteration
22  }
23 }
24
25 int main(void) {
```

Line: 36 Col: 26

Correctness and the Loop Invariant

Exit Full Screen View

## Congratulations

You solved this challenge. Would you like to challenge your friends? [f](#) [t](#) [in](#) [Next Challenge](#)

Test case 0 Success

Test case 1 Success

Test case 2 Success

Test case 3 Success

Compiler Message

Success

Input (stdin)

```
6
4 1 3 5 6 2
```

Download

Expected Output

```
1 1 2 3 4 5 6
```

Download

3. [https://www.hackerrank.com/challenges/countingsort2/problem?isFull](https://www.hackerrank.com/challenges/countingsort2/problem?isFullScreen=true)

### Solution Screenshot:

HackerRank | Prepare > Algorithms > Sorting > Counting Sort 2

Change Theme Language C++11

Exit Full Screen View

Problem

Often, when a list is sorted, the elements being sorted are just keys to other values. For example, if you are sorting files by their size, the sizes need to stay connected to their respective files. You cannot just take the size numbers and output them in order, you need to output all the required file information.

The counting sort is used if you just need to sort a list of integers. Rather than using a comparison, you create an integer array whose index range covers the entire range of values in your array to sort. Each time a value occurs in the original array, you increment the counter at that index. At the end, run through your counting array, printing the value of each non-zero valued index that number of times.

For example, consider an array `arr = [1, 1, 3, 2, 1]`. All of the values are in the range `[0...3]`, so create an array of zeroes, `result = [0, 0, 0, 0]`. The results of each iteration follow:

i	arr[i]	result
0	1	[0, 1, 0, 0]
1	1	[0, 2, 0, 0]
2	3	[0, 2, 0, 1]
3	2	[0, 2, 1, 1]
4	1	[0, 3, 1, 1]

Now we can print the sorted array: `sorted = [1, 1, 1, 2, 3]`.

Challenge

Counting Sort 2

Exit Full Screen View

## Congratulations

You solved this challenge. Would you like to challenge your friends? [f](#) [t](#) [in](#) [Next Challenge](#)

Test case 0 Success

Test case 1 Success

Test case 2 Success

Test case 3 Success

Test case 4 Success

Compiler Message

Success

Input (stdin)

```
100
63 25 73 1 98 73 56 84 86 57 16 83 8 25 81 56 9 53 98 67 99 12
83 89 80 91 39 86 76 85 74 39 25 90 59 10 94 32 44 3 89 30 27
79 46 96 27 32 18 21 92 69 81 40 40 34 68 78 24 87 42 69 23 41
78 22 6 90 99 89 50 30 20 1 43 3 70 95 33 46 44 9 69 48 33 60
65 16 82 67 61 32 21 79 75 75 13 87 70 33
```

Download

Expected Output

```
1 1 1 3 3 6 8 9 9 10 12 13 16 16 18 20 21 21 22 23 24 25 25 25 27
27 30 30 32 32 32 33 33 33 34 39 39 40 40 41 42 43 44 44 46 46
48 50 53 56 56 57 59 60 61 63 65 67 67 68 69 69 70 70 73 73
74 75 75 76 78 78 79 79 80 81 81 82 83 83 84 85 86 86 87 87 89
89 89 90 90 91 92 94 95 96 98 98 99 99
```

Download

#### 4. <https://www.hackerrank.com/challenges/runningtime/problem?isFullScreen=true>

### Solution Screenshot:

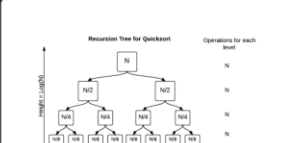
Problem

are. If you are unlucky and select the greatest or the smallest element as the pivot, then each partition will separate only one element at a time, so the running time will be similar to Insertion Sort.

However, Quicksort will usually pick a pivot that is mid-range, and it will partition the array into two parts. Let's assume Partition is lucky and it always picks the median element as the pivot. What will be the running time in such a case?

Running Time of Recursive Methods

Quicksort is a recursive method, so we will have to use a technique to calculate the total running time of all the method calls. We can use a version of the "Recursion Tree Method" to estimate the running time for a given array of  $N$  elements.



35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

Line: 62

Change Theme

Language

C++14

Exit Full Screen

Upload Code as File

Test against custom input

Run Code

Submit

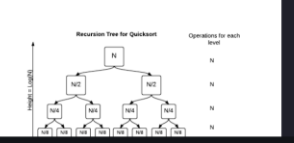
Problem

are. If you are unlucky and select the greatest or the smallest element as the pivot, then each partition will separate only one element at a time, so the running time will be similar to Insertion Sort.

However, Quicksort will usually pick a pivot that is mid-range, and it will partition the array into two parts. Let's assume Partition is lucky and it always picks the median element as the pivot. What will be the running time in such a case?

Running Time of Recursive Methods

Quicksort is a recursive method, so we will have to use a technique to calculate the total running time of all the method calls. We can use a version of the "Recursion Tree Method" to estimate the running time for a given array of  $N$  elements.



Line: 62 Col: 1

Upload Code as File

Test against custom input

Run Code

Submit Code

Test case 0

Compiler Message

Success

Input (stdin)

Download

Expected Output

Download