PIMPRI CHINCHWAD EDUCATION TRUST's.

**PIMPRI CHINCHWAD COLLEGE OF ENGINEERING**

(An Autonomous Institute)

**Class : SY BTech** **Acad. Yr. 2025-26** **Semester : I**

**Name of the student: Om Jitendra Khalane** **PRN : 124B1B040**

**Department: Computer Engineering** **Division : A**

**Course Name : Data Structures and Laboratory** **Course Code: BCE23PC02**

**Completion Date : 30/07/2025**

# Assignment No. 03

**Problem Statement:**

A banking app needs to display a user's transaction history sorted by transaction amount to quickly identify large deposits or withdrawals. Write a program for above scenario.
Hint:
Given a list of transaction amounts (positive and negative), implement Quick Sort to sort transactions in ascending order of amount. The solution should efficiently handle thousands of transactions.

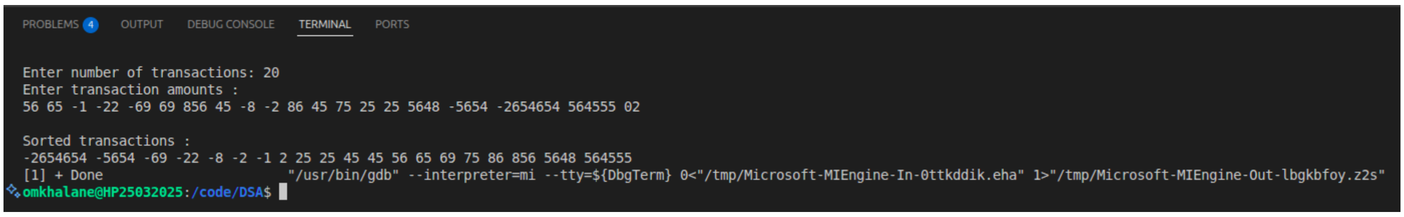
**Source Code :**

https://github.com/omkhalane/DSAL-SY-PCCOE/blob/main/lab_assignments/assignment03.cpp

```
#include <bits/stdc++.h>
 using namespace std;

 // partition
 int partition(vector<long double> &arr, int low, int high)
 {
     int pivot = arr[high];
     int i = low - 1;

     for (int j = low; j < high; j++)
     {
         if (arr[j] <= pivot)
         {
             i++;
```

```
                swap(arr[i], arr[j]);
            }
        }
        swap(arr[i + 1], arr[high]);
        return i + 1;
    }

    // quick sort
    void quickSort(vector<long double> &arr, int low, int high)
    {
        if (low < high)
        {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    int main()
    {
        vector<long double> t; // long double to handle large transactions
        int n;

        cout << "Enter number of transactions: ";
        cin >> n;

        cout << "Enter transaction amounts :\n";
        for (int i = 0; i < n; i++)
        {
            long double amount;
            cin >> amount;
            t.push_back(amount);
        }

        quickSort(t, 0, n - 1);

        cout << "\nSorted transactions :\n";
        for (int i = 0; i < t.size(); i++)
        {
            cout << t[i] << " ";
        }
        cout << endl;

        return 0;
    }
```

## Screen Shot of Output :



```
Enter number of transactions: 20
Enter transaction amounts :
56 65 -1 -22 -69 69 856 45 -8 -2 86 45 75 25 25 5648 -5654 -2654654 564555 02

Sorted transactions :
-2654654 -5654 -69 -22 -8 -2 -1 2 25 25 45 45 56 65 69 75 86 856 5648 564555
[1] + Done                    "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-0ttkddik.eha" 1>"/tmp/Microsoft-MIEngine-Out-lbgkbfoy.z2s"
omkhalane@HP25032025:/code/DSA$ 
```

## Conclusion:

The implemented program successfully sorts a user's transaction history in ascending order of transaction amounts using the **Quick Sort** algorithm.

- **Time Complexity:**
    - **Best Case:** O(nlogn)O(n \log n)O(nlogn) — Achieved when the pivot divides the list into two nearly equal halves at each partitioning step.
    - **Average Case:** O(nlogn)O(n \log n)O(nlogn) — Expected when transaction amounts are randomly distributed.
    - **Worst Case:** O(n2)O(n^2)O(n2) — Occurs when the pivot selection results in highly unbalanced partitions, such as with pre-sorted data, reverse-sorted data, or a large number of duplicate transaction amounts.
- **Space Complexity:**
    - **Auxiliary Space:** O(logn)O(\log n)O(logn) for the recursion stack in the best and average cases, and O(n)O(n)O(n) in the worst case.
    - **In-place Sorting:** The algorithm does not require additional arrays for sorting, making it space-efficient apart from the recursion overhead.
- **Observations:**
    - Large transaction values do not impact the performance as Quick Sort operates on element comparisons, not magnitude.
    - A high frequency of duplicate values can degrade efficiency toward the worst case. Techniques such as **randomized pivot selection** or **three-way partitioning** can be adopted to improve performance in such cases.

This approach ensures efficient handling of thousands of transactions, enabling the banking application to quickly display and analyze large deposits or withdrawals.