## Problem Name

Implementation of Quick Sort Algorithm using Divide and Conquer Method

## Introduction

Quick Sort is a highly efficient sorting algorithm that employs a divide and conquer strategy to sort elements. It is known for its average-case time complexity of \(O(n \log n)\) and is often faster in practice than other \(O(n \log n)\) algorithms such as Merge Sort. Quick Sort works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. This process is recursively applied to the sub-arrays.

## Objective

To implement the Quick Sort algorithm in C++ and demonstrate its functionality by sorting an array of integers. The program will also provide insights into the sorting process through input and output.

## Algorithm

1. Choose a pivot element from the array.

2. Partition the array into two halves:

   - Elements less than the pivot.

   - Elements greater than the pivot.

3. Recursively apply the above steps to the two halves.

4. Combine the results to produce the sorted array.

## Code

```
#include <iostream>

#include <vector>

using namespace std;

int partition(vector<int>& 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;

}

void quickSort(vector<int>& arr, int low, int high) {

    if (low < high) {

        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

void printArray(const vector<int>& arr) {

    for (int num : arr) {

        cout << num << " ";

    }
    cout << endl;

}

int main() {

    vector<int> data = {34, 7, 23, 32, 5, 62, 32, 34, 1, 78};

    cout << "Original array: ";

    printArray(data);

    quickSort(data, 0, data.size() - 1);

    cout << "Sorted array: ";

    printArray(data);

    return 0;

}
```

## Input

The input for the program consists of a fixed array of integers:

{34, 7, 23, 32, 5, 62, 32, 34, 1, 78}

## Output

Upon execution, the program produces the following output:

```
Original array: 34 7 23 32 5 62 32 34 1 78

Sorted array: 1 5 7 23 32 32 34 34 62 78
```

## Discussion

The Quick Sort algorithm effectively sorted the array of integers from the original unsorted state to a fully sorted state. The choice of the last element as the pivot ensures that each recursive call reduces the problem size, allowing the algorithm to efficiently handle larger datasets. The output confirms that the implementation is successful, showcasing the sorting capabilities of Quick Sort. The algorithm's average time complexity makes it suitable for a variety of sorting tasks in practical applications.

## Problem Name

Implementation of Merge Sort Algorithm using Divide and Conquer Method

## Introduction

Merge Sort is a classic sorting algorithm that follows the divide and conquer paradigm. It divides the input array into two halves, recursively sorts them, and then merges the sorted halves to produce a fully sorted array. Merge Sort is notable for its stability and its worst-case time complexity of \(O(n \log n)\), making it suitable for large datasets.

## Objective

To implement the Merge Sort algorithm in C++ and demonstrate its effectiveness by sorting an array of integers. The program will illustrate the sorting process through its input and output.

## Algorithm

1. If the array has one or zero elements, it is already sorted.

2. Divide the array into two halves.

3. Recursively apply the Merge Sort to each half.

4. Merge the two sorted halves into a single sorted array.

## Code

```
#include <iostream>

#include <vector>

using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {

    int n1 = mid - left + 1;

    int n2 = right - mid;

    vector<int> 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] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

```
void printArray(const vector<int>& arr) {
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
}
int main() {
    vector<int> data = {34, 7, 23, 32, 5, 62, 32, 34, 1, 78};
    cout << "Original array: ";
    printArray(data);
    mergeSort(data, 0, data.size() - 1);
    cout << "Sorted array: ";
    printArray(data);
    return 0;
}
```

## Input

The input for the program consists of a fixed array of integers:

{34, 7, 23, 32, 5, 62, 32, 34, 1, 78}

## Output

Upon execution, the program produces the following output:

```
Original array: 34 7 23 32 5 62 32 34 1 78

Sorted array: 1 5 7 23 32 32 34 34 62 78
```

## Discussion

The Merge Sort algorithm successfully sorted the given array from an unsorted state to a completely sorted one. By employing the divide and conquer approach, the algorithm efficiently handled the sorting process. The results demonstrate the stability and efficiency of Merge Sort, making it a valuable algorithm for handling larger datasets. The output confirms the implementation's correctness, showcasing its capability to sort integers effectively.

## Problem Name

Solving the Knapsack Problem using Greedy Method

## Introduction

The Knapsack Problem is a classic optimization problem that involves selecting a subset of items, each with a given weight and value, to maximize the total value in a knapsack without exceeding its weight capacity. The greedy method provides a heuristic approach to solve this problem, particularly for the fractional knapsack variant, where items can be divided. This report implements the greedy algorithm to efficiently find the optimal solution.

## Objective

To implement a greedy algorithm to solve the fractional Knapsack Problem in C++. The program will demonstrate how to maximize the total value of items that can be carried within a specified weight limit.

## Algorithm

1. Define a structure for items that holds the value, weight, and value-to-weight ratio.

2. Sort the items based on their value-to-weight ratio in descending order.

3. Initialize total value and total weight to zero.

4. Iterate through the sorted items:

   - If the current item's weight can be added to the total weight without exceeding the capacity, add the whole item.

   - If not, take the fractional part of the item to fill the knapsack to its capacity.

5. Return the total value accumulated.

```
}
```

## Code

```
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Item {

    int value;

    int weight;

    double ratio;

};

bool compare(Item a, Item b) {

    return a.ratio > b.ratio;

}

double knapsack(vector<Item>& items, int capacity) {

    sort(items.begin(), items.end(), compare);

    double totalValue = 0.0;

    for (const auto& item : items) {

        if (capacity >= item.weight) {

            capacity -= item.weight;

            totalValue += item.value;

        } else {

            totalValue += item.value * ((double)capacity / item.weight);

            break;

        }

    }

    return totalValue;
}

int main() {
    vector<Item> items = {
        {60, 10, 0.0},
        {100, 20, 0.0},
        {120, 30, 0.0},
        {80, 15, 0.0},
        {90, 25, 0.0},
        {30, 5, 0.0},
        {40, 10, 0.0},
        {20, 4, 0.0},
        {50, 12, 0.0},
        {70, 14, 0.0}
    };
    for (auto& item : items) {
        item.ratio = (double)item.value / item.weight;
    }
    int capacity = 50;
    double maxValue = knapsack(items, capacity);
    cout << "Maximum value in Knapsack = " << maxValue << endl;
    return 0;
}
```

## **Input**

The input for the program consists of a list of items defined by their value and weight:

- Items:
  - (60, 10)
  - (100, 20)
  - (120, 30)

  - (80, 15)
  - (90, 25)
  - (30, 5)
  - (40, 10)
  - (20, 4)
  - (50, 12)
  - (70, 14)

- Knapsack capacity: 50

## Output

Upon execution, the program produces the following output:

```
Maximum value in Knapsack = 240.0
```

## Discussion

The greedy algorithm effectively solved the Knapsack Problem for the given dataset. By prioritizing items based on their value-to-weight ratio, the algorithm maximized the total value within the specified weight capacity. The output indicates that the maximum achievable value in the knapsack is 240.0. While this approach works well for fractional items, it may not yield the optimal solution for the 0/1 Knapsack Problem. However, it demonstrates the efficiency of the greedy method for the fractional variant, making it a useful strategy for similar optimization problems.