**Assignemnt -1**

#include <bits/stdc++.h>

using namespace std;

struct Order {

    long long timestamp;

    string customerName;

    string orderDetails;

};

// Merge function

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

    int n1 = mid - left + 1;

    int n2 = right - mid;

    vector<Order> 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].timestamp <= R[j].timestamp)

            arr[k++] = L[i++];

        else

            arr[k++] = R[j++];

    }

    while (i < n1) arr[k++] = L[i++];

    while (j < n2) arr[k++] = R[j++];

}

// Recursive Merge Sort

void mergeSort(vector<Order>& 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);

    }

}

int main() {

    ios::sync\_with\_stdio(false);

    cin.tie(nullptr);

    const int N = 1'000'000; // 1 million records

    vector<Order> orders(N);

    // Random data generator

    mt19937\_64 rng(chrono::steady\_clock::now().time\_since\_epoch().count());

    uniform\_int\_distribution<long long> tsDist(1600000000LL, 1700000000LL); // random epoch timestamps

    uniform\_int\_distribution<int> nameDist(1, 1000000);

    for (int i = 0; i < N; i++) {

        orders[i].timestamp = tsDist(rng);

        orders[i].customerName = "Customer" + to\_string(nameDist(rng));

        orders[i].orderDetails = "Order" + to\_string(i);

    }

    cout << "Sorting " << N << " records...\n";

    auto start = chrono::high\_resolution\_clock::now();

    mergeSort(orders, 0, N - 1);

    auto end = chrono::high\_resolution\_clock::now();

    chrono::duration<double> elapsed = end - start;

    cout << "Sorting completed in " << elapsed.count() << " seconds\n";

    // Print first 10 sorted records to verify

    cout << "\nFirst 10 sorted records:\n";

    for (int i = 0; i < 10; i++) {

        cout << orders[i].timestamp << " "

             << orders[i].customerName << " "

             << orders[i].orderDetails << "\n";

    }

    return 0;

}

**No 10000 thing:**

#include <bits/stdc++.h>

using namespace std;

struct Order {

long long timestamp;

string customerName;

string orderDetails;

};

// Merge function

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

int n1 = mid - left + 1;

int n2 = right - mid;

vector<Order> 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].timestamp <= R[j].timestamp)

arr[k++] = L[i++];

else

arr[k++] = R[j++];

}

while (i < n1) arr[k++] = L[i++];

while (j < n2) arr[k++] = R[j++];

}

// Recursive Merge Sort

void mergeSort(vector<Order>& 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);

}

}

int main() {

ios::sync\_with\_stdio(false);

cin.tie(nullptr);

// Example dataset

vector<Order> orders = {

{1700000005, "Alice", "Order1"},

{1600000002, "Bob", "Order2"},

{1650000003, "Charlie", "Order3"},

{1620000001, "Dave", "Order4"},

{1680000004, "Eve", "Order5"}

};

cout << "Before sorting:\n";

for (auto &o : orders) {

cout << o.timestamp << " " << o.customerName << " " << o.orderDetails << "\n";

}

mergeSort(orders, 0, orders.size() - 1);

cout << "\nAfter sorting by timestamp:\n";

for (auto &o : orders) {

cout << o.timestamp << " " << o.customerName << " " << o.orderDetails << "\n";

}

return 0;

}

**Assingment -2**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

struct Movie {

string title;

float rating;

int release\_year;

int popularity;

void display() const {

cout << title << " | Rating: " << rating

<< " | Year: " << release\_year

<< " | Popularity: " << popularity << endl;

}

};

// QuickSort

int partition(vector<Movie> &movies, int low, int high, bool (\*compare)(const Movie &, const Movie &)) {

Movie pivot = movies[high];

int i = low - 1;

for (int j = low; j < high; j++) {

if (compare(movies[j], pivot)) {

i++;

swap(movies[i], movies[j]);

}

}

swap(movies[i + 1], movies[high]);

return i + 1;

}

void quickSort(vector<Movie> &movies, int low, int high, bool (\*compare)(const Movie &, const Movie &)) {

if (low < high) {

int pi = partition(movies, low, high, compare);

quickSort(movies, low, pi - 1, compare);

quickSort(movies, pi + 1, high, compare);

}

}

// Comparators

bool compareByRating(const Movie &a, const Movie &b) {

return a.rating < b.rating;

}

bool compareByYear(const Movie &a, const Movie &b) {

return a.release\_year < b.release\_year;

}

bool compareByPopularity(const Movie &a, const Movie &b) {

return a.popularity < b.popularity;

}

int main() {

vector<Movie> movies = {

{"Inception", 8.8, 2010, 900000},

{"The Matrix", 8.7, 1999, 850000},

{"Interstellar", 8.6, 2014, 870000},

{"The Godfather", 9.2, 1972, 1200000},

{"Avengers: Endgame", 8.4, 2019, 1100000}

};

cout << "Sort movies by (rating/year/popularity): ";

string sort\_by;

cin >> sort\_by;

bool (\*compare)(const Movie &, const Movie &);

if (sort\_by == "rating")

compare = compareByRating;

else if (sort\_by == "year")

compare = compareByYear;

else if (sort\_by == "popularity")

compare = compareByPopularity;

else {

cout << "Invalid choice. Defaulting to rating.\n";

compare = compareByRating;

}

quickSort(movies, 0, movies.size() - 1, compare);

cout << "\nSorted movies:\n";

for (const auto &m : movies) {

m.display();

}

return 0;

}

**Assingment-3**

#include <bits/stdc++.h>

using namespace std;

struct Item {

    double weight, value;

    bool divisible; // true for food/water, false for medicine kits

};

// Comparator based on value-to-weight ratio

bool cmp(const Item &a, const Item &b) {

    double r1 = a.value / a.weight;

    double r2 = b.value / b.weight;

    return r1 > r2;

}

double fractionalKnapsack(double W, vector<Item> &items) {

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

    double totalValue = 0.0;

    double currentWeight = 0.0;

    cout << "Items selected for transport:\n";

    for (auto &item : items) {

        if (currentWeight + item.weight <= W) {

            // Take the whole item

            currentWeight += item.weight;

            totalValue += item.value;

            cout << "  Took full item | Weight: " << item.weight

                 << " | Value: " << item.value

                 << " | Divisible: " << (item.divisible ? "Yes" : "No") << "\n";

        } else {

            double remain = W - currentWeight;

            if (remain <= 0) break;

            if (item.divisible) {

                // Take fraction

                double fraction = remain / item.weight;

                totalValue += item.value \* fraction;

                cout << "  Took fraction " << fraction \* 100 << "% of item | Weight: " << remain

                     << " | Value: " << item.value \* fraction

                     << " | Divisible: Yes\n";

                currentWeight += remain;

                break; // boat is full

            } else {

                // Cannot take indivisible item partially

                continue;

            }

        }

    }

    cout << "Total weight on boat: " << currentWeight << "/" << W << " kg\n";

    cout << "Maximum utility value: " << totalValue << "\n";

    return totalValue;

}

int main() {

    ios::sync\_with\_stdio(false);

    cin.tie(nullptr);

    double W = 50; // max weight capacity of boat

    // Example items: {weight, value, divisible}

    vector<Item> items = {

        {10, 60, false},  // Medicine kit

        {20, 100, true},  // Food

        {30, 120, true},  // Water

        {25, 180, false}, // Another medical kit

        {15, 90, true}    // More food

    };

    fractionalKnapsack(W, items);

    return 0;

}

**Assingment 4**

#include <bits/stdc++.h>

using namespace std;

struct Edge {

    int to;

    double weight; // travel time

};

using Graph = vector<vector<Edge>>;

vector<double> dijkstra(const Graph &graph, int source) {

    int n = graph.size();

    vector<double> dist(n, 1e18); // large number = infinity

    dist[source] = 0.0;

    typedef pair<double, int> P; // {distance, node}

    priority\_queue<P, vector<P>, greater<P>> pq;

    pq.push({0.0, source});

    while (!pq.empty()) {

        P top = pq.top();

        pq.pop();

        double d = top.first;

        int u = top.second;

        if (d > dist[u]) continue; // stale entry

        for (auto &edge : graph[u]) {

            int v = edge.to;

            double w = edge.weight;

            if (dist[u] + w < dist[v]) {

                dist[v] = dist[u] + w;

                pq.push({dist[v], v});

            }

        }

    }

    return dist;

}

int main() {

    ios::sync\_with\_stdio(false);

    cin.tie(nullptr);

    int n = 6; // intersections

    Graph graph(n);

    // Roads (u -> v, time in minutes)

    graph[0].push\_back({1, 4});

    graph[0].push\_back({2, 2});

    graph[1].push\_back({2, 5});

    graph[1].push\_back({3, 10});

    graph[2].push\_back({3, 3});

    graph[3].push\_back({4, 4});

    graph[4].push\_back({5, 11});

    int source = 0; // ambulance starting point

    vector<int> hospitals = {3, 5}; // possible destinations

    auto dist = dijkstra(graph, source);

    cout << "Shortest travel times from Source " << source << ":\n";

    for (int h : hospitals) {

        cout << "To Hospital at " << h << " = " << dist[h] << " minutes\n";

    }

    return 0;

}