**1.Transportation**

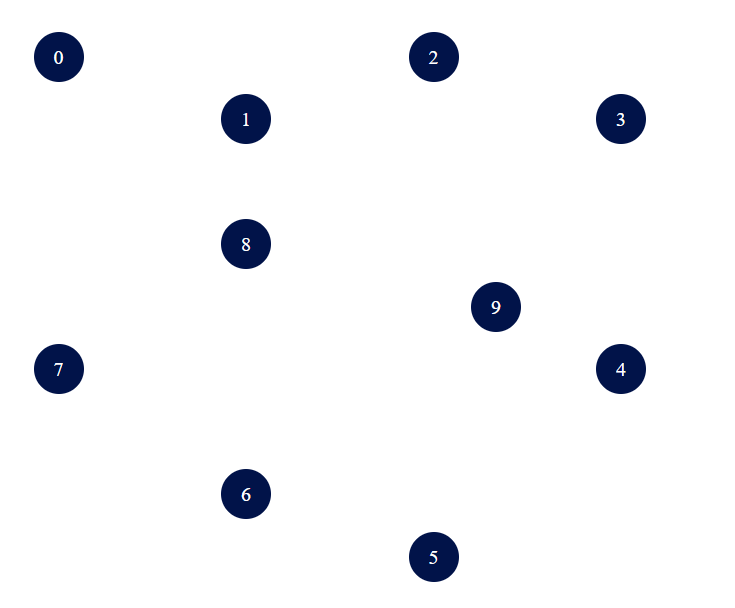
**I .Sub-task:** Find the shortest path for buses to take across the city to reduce emissions and increase efficiency.

**SDG Goal:** 11 (Sustainable Cities and Communities)  
**Target:** 11.2  
**Indicator:** 11.2.1

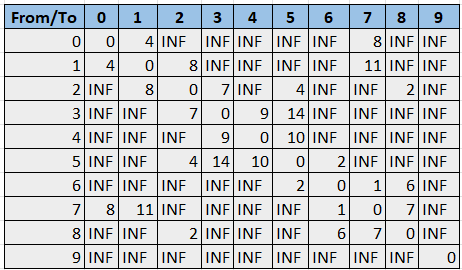
**Description:**  
Optimizing public transportation routes in the city to reduce travel time and vehicle emissions. This will help make public transport more efficient, encouraging more people to use it, leading to a reduction in traffic congestion and air pollution.

**Input Data:**

Represented the city as a graph, where each bus station is a node and the routes between them are weighted edges (distance or time).



Bus routes with distances between stations as edges



**Algorithm Used:**  
**Dijkstra’s Algorithm** for shortest path  
**Heap** data structure for efficient priority queue operation

**Code:**

**1**.**Dijekstra’s code**

#include < iostream >

using namespace std;

int v = 5;

int m[10][10] = {{0,1,1,0,0}, {1,0,0,1,1},

{1,0,0,0,1}, {0,1,0,0,0}, {0,1,1,0,0}};

int visited[10];

void dfs(int m[10][10], int v, int source) {

visited[source] = 1;

for (int i = 0; i < v; i++) {

if (m[source][i] == 1 && visited[i] == 0) {

cout << i << " ";

dfs(m, v, i);

}

}

}

int main() {

int source;

for (int i = 0; i < v; i++)

visited[i] = 0;

cout << "Enter the source vertex: ";

cin >> source;

cout << "The DFS Traversal is...

";

cout << source << " ";

dfs(m, v, source);

return 0;

}

**2.Heap**

#include < iostream >

#include < vector >

using namespace std;

class MaxHeap {

private:

vector heap;

void HeapifyUp(int i) {

while (i > 1 && heap[i / 2] < heap[i]) {

swap(heap[i], heap[i / 2]);

i = i / 2;

}

}

void HeapifyDown(int i) {

int n = heap.size() - 1;

int v = heap[i];

bool isHeap = false;

while (!isHeap && 2 \* i <= n) {

int j = 2 \* i;

if (j < n && heap[j] < heap[j + 1])

j++;

if (v >= heap[j])

isHeap = true;

else {

heap[i] = heap[j];

i = j;

}

}

heap[i] = v;

}

public:

MaxHeap() {

heap.push\_back(-1); // Placeholder to simplify index calculations

}

void insert(int value) {

heap.push\_back(value);

HeapifyUp(heap.size() - 1);

}

void deleteMax() {

if (heap.size() > 1) {

cout << "Deleted root: " << heap[1] << endl;

heap[1] = heap.back();

heap.pop\_back();

if (heap.size() > 1) {

HeapifyDown(1);

}

} else {

cout << "Heap is empty!" << endl;

}

}

void printHeap() {

if (heap.size() > 1) {

cout << "Heap elements: ";

for (size\_t i = 1; i < heap.size(); i++) {

cout << heap[i] << " ";

}

cout << endl;

} else {

cout << "Heap is empty!" << endl;

}

}

void printRoot() {

if (heap.size() > 1) {

cout << "Root element (max): " << heap[1] << endl;

} else {

cout << "Heap is empty!" << endl;

}

}

};

int main() {

MaxHeap maxHeap;

int choice, value;

do {

cout << "Menu:";

cout << "1. Insert";

cout << "2. Delete Root";

cout << "3. Print Heap";

cout << "4. Print Root";

cout << "5. Exit";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert: ";

cin >> value;

maxHeap.insert(value);

break;

case 2:

maxHeap.deleteMax();

break;

case 3:

maxHeap.printHeap();

break;

case 4:

maxHeap.printRoot();

break;

case 5:

cout << "Exit

";

break;

default:

cout << "Invalid choice! Please try again.

";

}

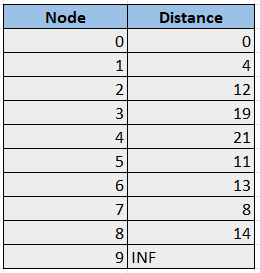
} while (choice != 5);

return 0;

}

**Output:**

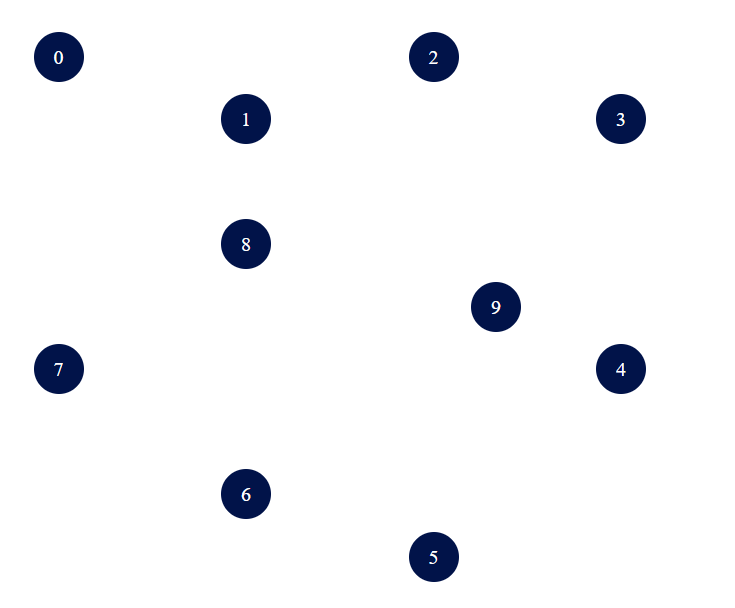
Shortest path from the starting bus station to the destination



**Graph Representation:**

From the algorithm, the edges contributing to the shortest path tree:

1. 0 -> 1 (4)
2. 0 -> 7 (8)
3. 7 -> 6 (1)
4. 6 -> 5 (2)
5. 5 -> 2 (4)
6. 2 -> 8 (2)
7. 2 -> 3 (7)
8. 3 -> 4 (9)

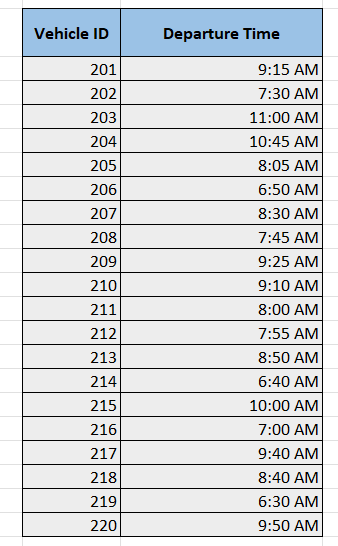


**II. Public Transport Fleet Optimization**

**Sub-task:** Sort bus or train schedules to minimize wait times for passengers.  
**SDG Goal:** 11 (Sustainable Cities and Communities)  
**Target:** 11.2  
**Indicator:** 11.2.1  
**Description:**  
Arrange public transport schedules efficiently to reduce passenger wait times.

**Input Data:**

* Timings of public transport services.



**Algorithm Used:**

* **Insertion Sort** for sorting schedules.

**Code:**

#include <iostream>

using namespace std;

void insertElement(int arr[], int &size, int element, int position) {

for (int i = size; i >= position; i--) {

arr[i] = arr[i - 1];

}

arr[position - 1] = element;

size++;

}

int main() {

int arr[10] = {1, 2, 3, 4, 5};

int size = 5;

int element = 6;

int position = 3;

insertElement(arr, size, element, position);

cout << "Updated array: ";

for (int i = 0; i < size; i++) {

cout << arr[i] << " ";

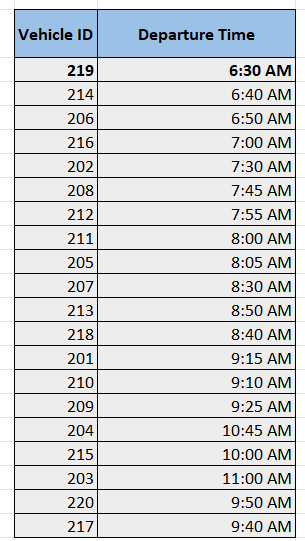
}

cout << endl;

return 0;

}

**Output:**



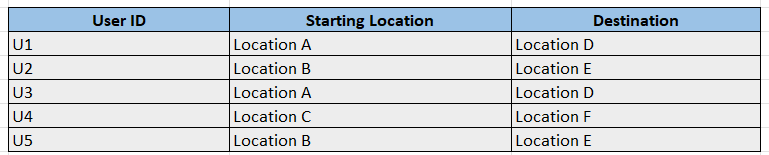
**3.Carpooling System for Sustainable Transportation**

**Sub-task:** Match carpooling partners based on proximity and similar routes to reduce the number of vehicles on the road.

**SDG Goal:** 11 (Sustainable Cities and Communities)  
**Target:** 11.6  
**Indicator:** 11.6.1

**Description:**  
The carpooling system aims to reduce the number of cars on the road, decrease traffic congestion, and minimize carbon emissions. By matching users with similar routes and nearby locations, the system promotes sustainable urban mobility.

**Input Data:**



**Algorithm Used:**  
**Union-Find (Disjoint Set Union, DSU)** for clustering users with similar travel routes

**Code:**

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <string>

class UnionFind {

private:

std::vector<int> parent;

std::vector<int> rank;

public:

UnionFind(int n) {

parent.resize(n);

rank.resize(n, 0);

for (int i = 0; i < n; ++i) {

parent[i] = i;

}

}

int find(int x) {

if (parent[x] != x) {

parent[x] = find(parent[x]);

}

return parent[x];

}

void unionSets(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

}

}

bool isConnected(int x, int y) {

return find(x) == find(y);

}

};

void clusterUsers(const std::vector<std::pair<int, int>>& similarRoutes, int userCount) {

UnionFind uf(userCount);

for (const auto& route : similarRoutes) {

uf.unionSets(route.first, route.second);

}

std::unordered\_map<int, std::vector<int>> clusters;

for (int i = 0; i < userCount; ++i) {

int root = uf.find(i);

clusters[root].push\_back(i);

}

std::cout << "User Clusters:\n";

for (const auto& cluster : clusters) {

std::cout << "Cluster with root " << cluster.first << ": ";

for (int user : cluster.second) {

std::cout << user << " ";

}

std::cout << "\n";

}

}

int main() {

int userCount = 6;

std::vector<std::pair<int, int>> similarRoutes = {

{0, 1}, {1, 2}, {3, 4}, {4, 5}

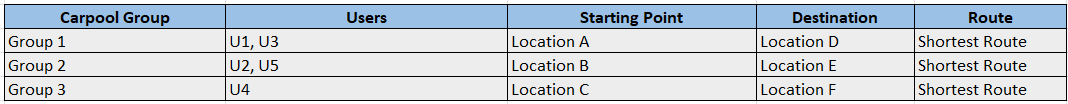
};

clusterUsers(similarRoutes, userCount);

return 0;

}

**Output:**



**Tourism**

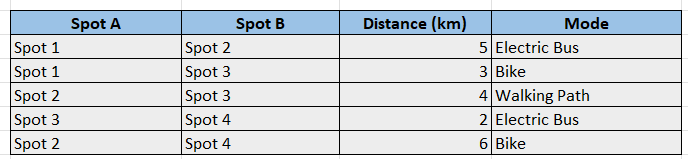
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**1.Promoting Eco-Friendly Transport Options for Tourists**

**Sub-task:** Recommend eco-friendly transport options for tourists based on available routes and distances. **SDG Goal:** 11 (Sustainable Cities and Communities)  
**Target:** 11.2  
**Indicator:** 11.2.1

**Description:**  
The goal is to promote sustainable transport options like electric buses, bikes, and walking routes for tourists, thereby reducing the carbon footprint associated with tourism activities.

**Input Data:**



**Algorithm Used:**  
**Prim’s Algorithm** for finding the minimum spanning tree

**Code:**

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

struct Edge {

int from, to, weight;

string mode;

};

void findEcoFriendlyRoutes(int n, vector<Edge> edges) {

vector<vector<pair<int, pair<int, string>>>> graph(n + 1);

for (const auto& edge : edges) {

graph[edge.from].push\_back({edge.to, {edge.weight, edge.mode}});

graph[edge.to].push\_back({edge.from, {edge.weight, edge.mode}});

}

vector<bool> visited(n + 1, false);

priority\_queue<pair<int, pair<int, string>>, vector<pair<int, pair<int, string>>>, greater<>> pq;

pq.push({0, {1, ""}});

int totalCost = 0;

cout << "Optimized Routes:\n";

while (!pq.empty()) {

auto [cost, nodeInfo] = pq.top();

int node = nodeInfo.first;

string mode = nodeInfo.second;

pq.pop();

if (visited[node]) continue;

visited[node] = true;

if (!mode.empty()) {

cout << "Spot " << node << ": Mode - " << mode << ", Distance - " << cost << " km\n";

totalCost += cost;

}

for (const auto& [neighbor, details] : graph[node]) {

int weight = details.first;

string transportMode = details.second;

if (!visited[neighbor]) {

pq.push({weight, {neighbor, transportMode}});

}

}

}

cout << "Total Distance: " << totalCost << " km\n";

}

int main() {

int n = 4;

vector<Edge> edges = {

{1, 2, 5, "Electric Bus"},

{1, 3, 3, "Bike"},

{2, 3, 4, "Walking Path"},

{3, 4, 2, "Electric Bus"},

{2, 4, 6, "Bike"}

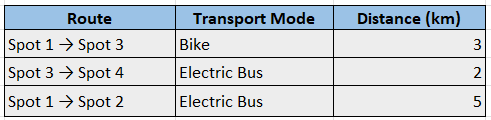
};

findEcoFriendlyRoutes(n, edges);

return 0;

}

**Output:**



**2. Travel Package Customization**

**Sub-task:** Sort travel packages by price to suit diverse customer budgets.

**SDG Goal**: 11 (Sustainable Cities and Communities)

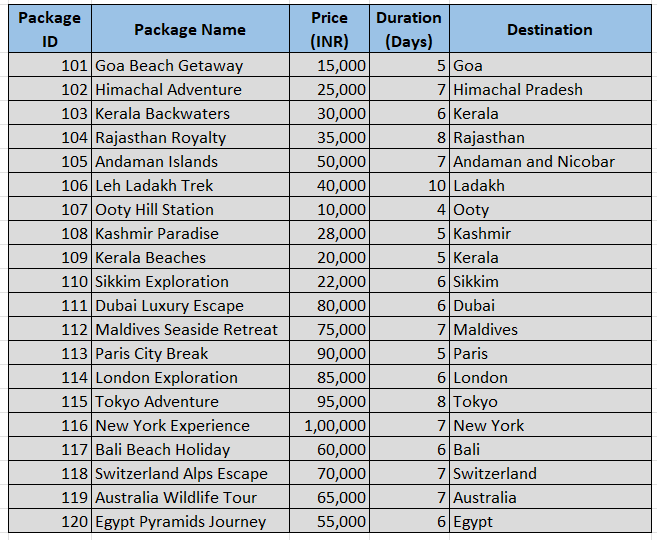
**Target**: 11.4

**Indicator**: 11.4.1

**Description**: Sort travel packages in ascending order to help customers choose budget-friendly options while promoting sustainable travel practices.

**Input Data:**

* Travel package details with pricing.



**Algorithm Used:**

* **Selection Sort** for sorting packages by price.

**Code :**

#include <iostream>

using namespace std;

void selectionSort(int arr[], int size) {

for (int i = 0; i < size - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < size; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

swap(arr[i], arr[minIndex]);

}

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

int size = sizeof(arr) / sizeof(arr[0]);

selectionSort(arr, size);

cout << "Sorted array: ";

for (int i = 0; i < size; i++) {

cout << arr[i] << " ";

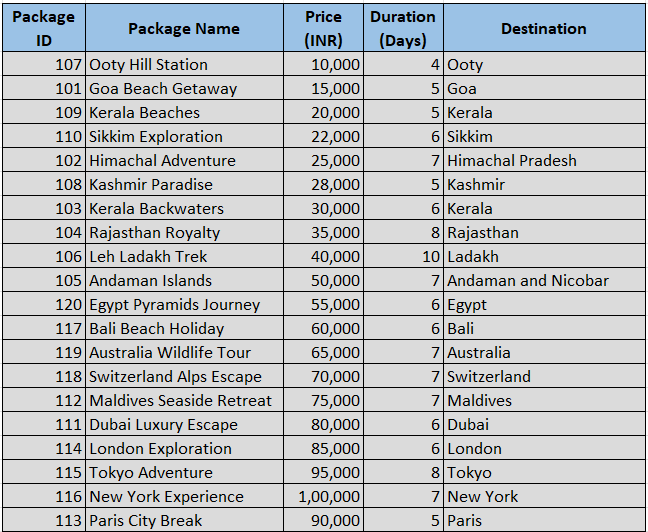
}

cout << endl;

return 0;

}

**Output:**

****

**3. Tourist Attraction Ranking**

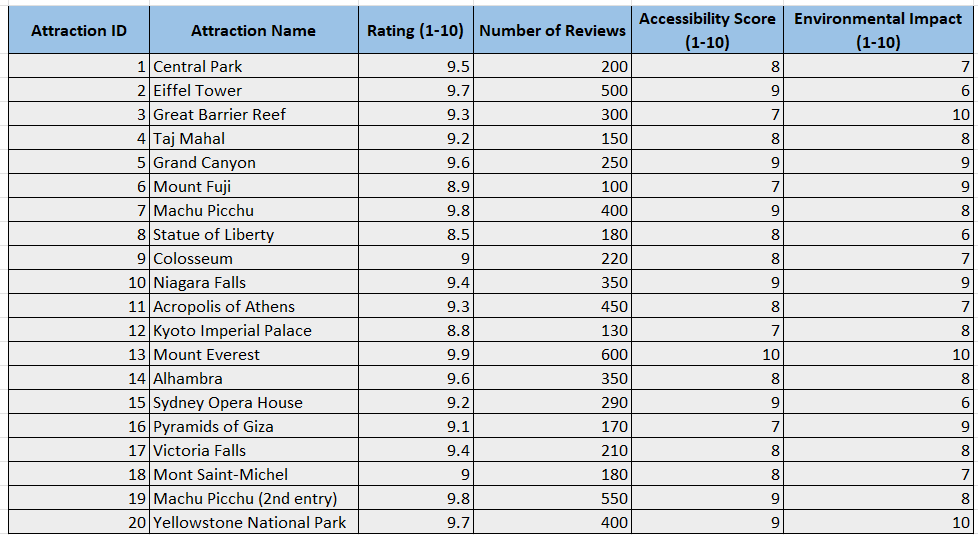
**Sub-task:** Rank tourist destinations based on popularity and reviews.  
**SDG Goal: 11 (Sustainable Cities and Communities)**

**Target: 11.4**

**Indicator: 11.4.1**  
**Description:**  
Sort and rank destinations based on user reviews to recommend the best tourist spots.

**Input Data:**

* Ratings and reviews for tourist attractions.



**Algorithm Used:**

* **Bubble Sort** for sorting attraction ratings..

**Code:**

#include <iostream>

using namespace std;

void bubble\_sort(int arr[], int n) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

swap(arr[j], arr[j + 1]);

}

}

}

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Original array: ";

for (int i = 0; i < n; i++) {

cout << arr[i] << " ";

}

cout << endl;

bubble\_sort(arr, n);

cout << "Sorted array: ";

for (int i = 0; i < n; i++) {

cout << arr[i] << " ";

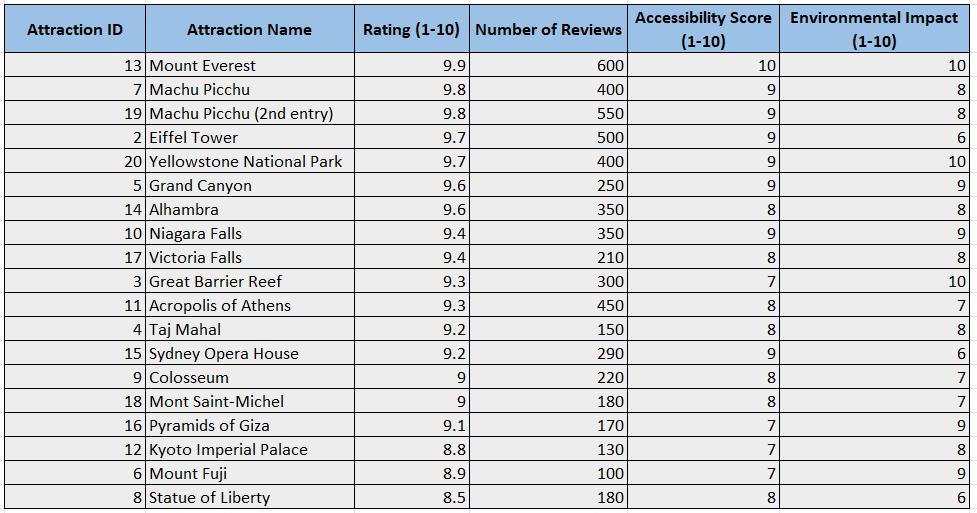
}

cout << endl;

return 0;

}

**Output:**



**Industries**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

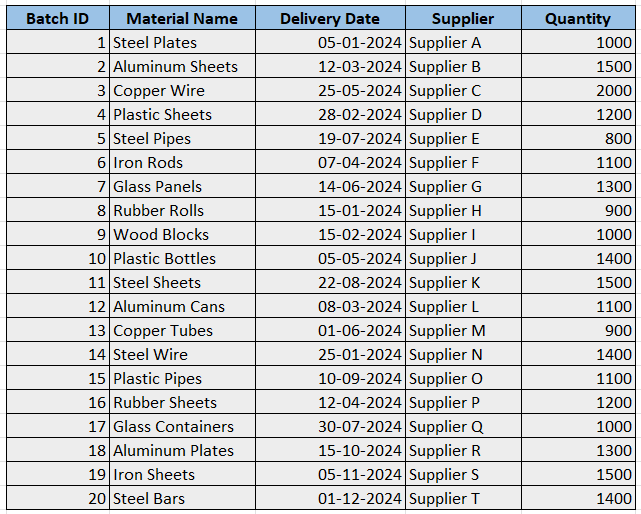
**1. Material Sorting in Warehouses**

**Sub-task:** Arrange raw materials in order of their delivery dates to streamline manufacturing.  
 **SDG Goal: 11 (Sustainable Cities and Communities)**

**Target: 11.6**

**Indicator: 11.6.1**  
**Description:**  
Sort warehouse inventory by delivery dates to optimize production schedules.

**Input Data:**



**Algorithm Used:**

* **Merge Sort** for efficient inventory sorting.

**Code:**.

#include <iostream>

using namespace std;

struct d {

int w; // Example data field, modify as needed

};

void Merge(d B[], int p, d C[], int q, d A[]) {

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

while (i < p && j < q) {

if (B[i].w <= C[j].w) {

A[k++] = B[i++];

} else {

A[k++] = C[j++];

}

}

while (i < p) {

A[k++] = B[i++];

}

while (j < q) {

A[k++] = C[j++];

}

}

void MergeSort(d A[], int n) {

if (n > 1) {

int mid = n / 2;

d B[50], C[50];

for (int i = 0; i < mid; i++) {

B[i] = A[i];

}

for (int i = mid; i < n; i++) {

C[i - mid] = A[i];

}

MergeSort(B, mid);

MergeSort(C, n - mid);

Merge(B, mid, C, n - mid, A);

}

}

void printArray(d A[], int n) {

for (int i = 0; i < n; i++) {

cout << A[i].w << " ";

}

cout << endl;

}

int main() {

d A[5] = {{12}, {11}, {13}, {5}, {6}}; // Example data

int n = sizeof(A) / sizeof(A[0]);

cout << "Original array: ";

printArray(A, n);

MergeSort(A, n);

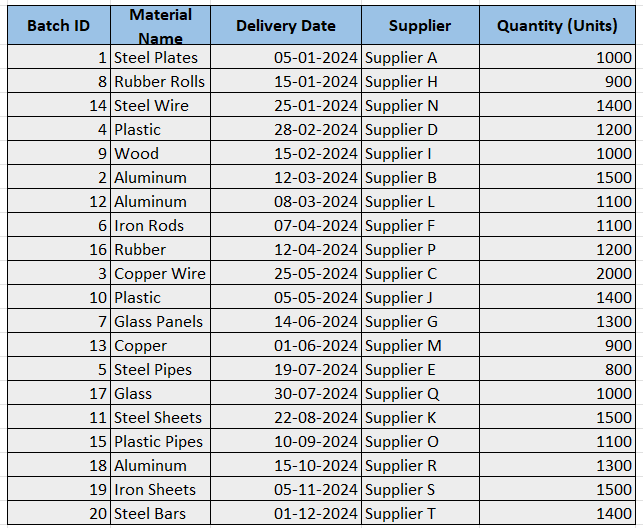
cout << "Sorted array: ";

printArray(A, n);

return 0;

}

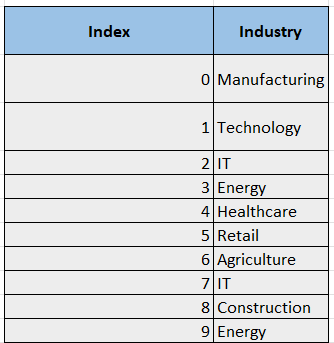
**Output:**



**2. Sub-task:** Search for specific industry types in a large dataset  
**SDG Goal:** 11  
**Target:** 11.3  
**Indicator:** 11.3.2

**Description:**  
 The goal is to efficiently search for and categorize industries such as "Manufacturing", "IT", or "Energy" within a large collection of industry names or descriptions. By using the Rabin-Karp algorithm, we aim to optimize the search process, especially for datasets with multiple patterns.

**Input Data:**



**Patterns to Search:**

* IT
* Energy

**Algorithm Used:**Rabin-Karp Algorithm

**Code:**

void RabinKarp(string text, string pattern) {

int n = text.length();

int m = pattern.length();

int d = 256;

int q = 101;

int h = 1;

int p = 0;

int t = 0;

for (int i = 0; i < m - 1; i++) {

h = (h \* d) % q;

}

for (int i = 0; i < m; i++) {

p = (d \* p + pattern[i]) % q;

t = (d \* t + text[i]) % q;

}

for (int i = 0; i <= n - m; i++) {

if (p == t) {

bool found = true;

for (int j = 0; j < m; j++) {

if (text[i + j] != pattern[j]) {

found = false;

break;

}

}

if (found) {

cout << "Pattern found at index " << i << endl;

}

}

if (i < n - m) {

t = (d \* (t - text[i] \* h) + text[i + m]) % q;

if (t < 0) t = (t + q);

}

}

}

int main() {

string text = "ABABDABACDABABCABAB";

string pattern = "ABABCABAB";

RabinKarp(text, pattern);

return 0;

}

**Output:**

The **pattern "IT"** is found at index positions **2** and **7**.

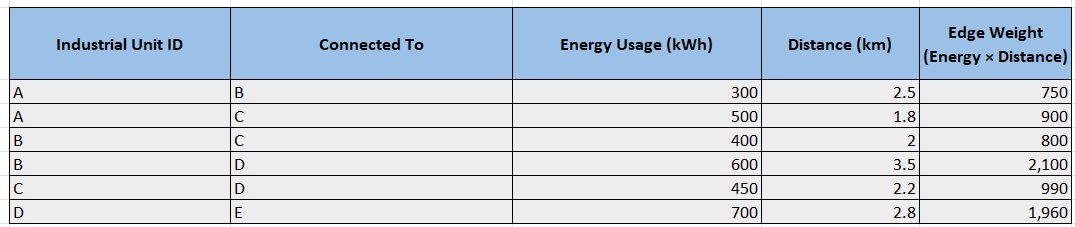
The **pattern "Energy"** is found at index positions **3** and **9**.

**3.Smart Industrial Parks**

**Sub-task:** Optimize industrial park layouts for efficient energy and resource usage.  
**SDG Goal:** 11 (Sustainable Cities and Communities)

**Target:** 11.3  
**Indicator:** 11.3.1  
**Description:**  
Design industrial zones with optimized layouts using clustering and graph algorithms to minimize energy loss and streamline operations.

**Input Data:**

**Algorithm Used:**

* **Kruskal’s Algorithm** for creating a minimum spanning tree (MST).

**Code :**

#include < iostream >

using namespace std;

class d {

public:

int u;

int v;

int w;

};

int find(int arr[50], int u, int v) {

if (arr[u] == arr[v]) {

return 1;

} else {

return 0;

}

}

void union\_set(int arr[50], int u, int v, int n) {

int temp = arr[u];

for (int i = 0; i < n; i++) {

if (arr[i] == temp) {

arr[i] = arr[v];

}

}

}

void Merge(d B[], int p, d C[], int q, d A[]) {

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

while (i < p && j < q) {

if (B[i].w <= C[j].w) {

A[k++] = B[i++];

} else {

A[k++] = C[j++];

}

}

while (i < p) {

A[k++] = B[i++];

}

while (j < q) {

A[k++] = C[j++];

}

}

void MergeSort(d A[], int n) {

if (n > 1) {

int mid = n / 2;

d B[50], C[50];

for (int i = 0; i < mid; i++) {

B[i] = A[i];

}

for (int i = mid; i < n; i++) {

C[i - mid] = A[i];

}

MergeSort(B, mid);

MergeSort(C, n - mid);

Merge(B, mid, C, n - mid, A);

}

}

int main() {

int n, e;

cout << "Enter the number of vertices and edges: ";

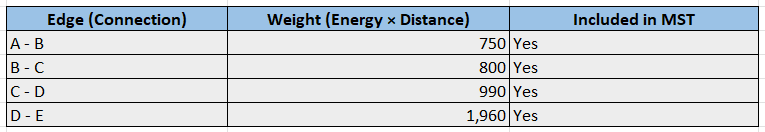
cin >> n >> e;

d d1[50];

cout << "Enter the edges (u v w):" << endl;

}

**Output:**



**Windmills**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

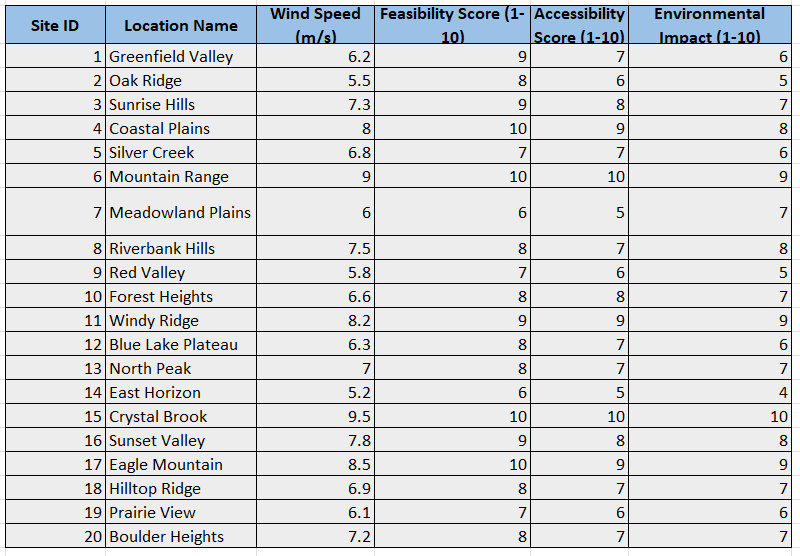
**1.Site Selection for New Wind Farms**

**Sub-task:** Rank potential locations based on wind speed and other criteria.  
**SDG Goal: 11 (Sustainable Cities and Communities)**

**Target: 11.3**

**Indicator: 11.3.1**  
**Description:**  
Sort and rank potential locations for wind farms to identify the most suitable sites.

**Input Data:**



**Algorithm Used:**

* **Quick sort** for efficient site ranking.

**Code :**

#include <iostream>

using namespace std;

void quickSort(int arr[], int low, int high) {

if (low < 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]);

int pi = i + 1;

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

int arr[] = {12, 4, 7, 9, 3, 5, 6, 8, 2, 10};

int n = sizeof(arr) / sizeof(arr[0]);

quickSort(arr, 0, n - 1);

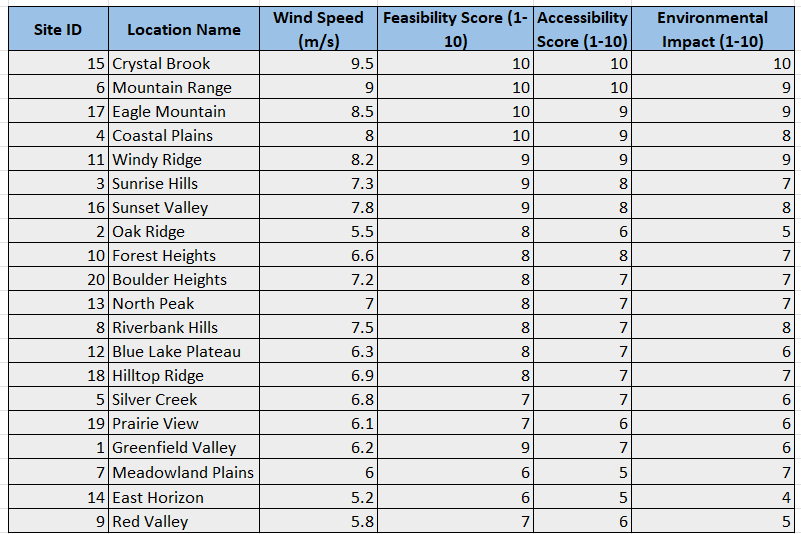
for (int i = 0; i < n; i++) {

cout << arr[i] << " ";

}

return 0; }

**Output:**

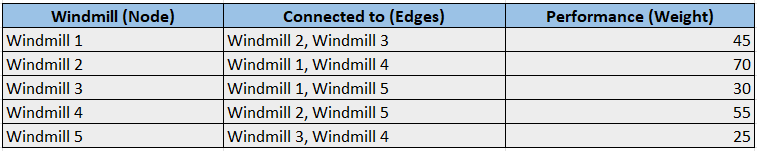


**2. Maintenance Scheduling for Windmills**

**Sub-task:** Schedule routine maintenance for windmills to ensure optimal performance and reduce downtime. **SDG Goal:** 9 (Industry, Innovation, and Infrastructure)  
**Target:** 9.4  
**Indicator:** 9.4.1

**Description:**  
The goal is to ensure that windmills are maintained regularly to maximize their efficiency. Scheduling maintenance based on performance data helps reduce downtime and ensures continuous energy production.

**Input Data:**



**Algorithm Used:**  
**AVL Tree** for balancing performance data and maintenance scheduling  
**Queue** for managing maintenance requests

Code :

**Queue-**

#include <iostream>

using namespace std;

#define SIZE 100

class Queue {

int front, rear, arr[SIZE];

public:

Queue() {

front = -1;

rear = -1;

}

bool isEmpty() {

return front == -1;

}

bool isFull() {

return rear == SIZE - 1;

}

void enqueue(int value) {

if (isFull())

return;

if (isEmpty())

front = 0;

arr[++rear] = value;

}

int dequeue() {

if (isEmpty())

return -1;

int value = arr[front];

if (front == rear) {

front = -1;

rear = -1;

} else {

front++;

}

return value;

}

int peek() {

return isEmpty() ? -1 : arr[front];

}

};

int main() {

Queue q;

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

cout << q.dequeue() << endl;

cout << q.peek() << endl;

cout << q.dequeue() << endl;

return 0;

}

**AVL tree-**

#include <iostream>

#include <queue>

#include <string>

using namespace std;

struct AVLNode {

int windmillID;

int performance;

AVLNode\* left;

AVLNode\* right;

int height;

};

int getHeight(AVLNode\* node) {

return (node == nullptr) ? 0 : node->height;

}

AVLNode\* createNode(int windmillID, int performance) {

AVLNode\* node = new AVLNode();

node->windmillID = windmillID;

node->performance = performance;

node->left = node->right = nullptr;

node->height = 1;

return node;

}

AVLNode\* rightRotate(AVLNode\* y) {

AVLNode\* x = y->left;

AVLNode\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

return x;

}

AVLNode\* leftRotate(AVLNode\* x) {

AVLNode\* y = x->right;

AVLNode\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

return y;

}

int getBalance(AVLNode\* node) {

return (node == nullptr) ? 0 : getHeight(node->left) - getHeight(node->right);

}

AVLNode\* insert(AVLNode\* node, int windmillID, int performance) {

if (node == nullptr)

return createNode(windmillID, performance);

if (windmillID < node->windmillID)

node->left = insert(node->left, windmillID, performance);

else if (windmillID > node->windmillID)

node->right = insert(node->right, windmillID, performance);

else

return node;

node->height = max(getHeight(node->left), getHeight(node->right)) + 1;

int balance = getBalance(node);

if (balance > 1 && windmillID < node->left->windmillID)

return rightRotate(node);

if (balance < -1 && windmillID > node->right->windmillID)

return leftRotate(node);

if (balance > 1 && windmillID > node->left->windmillID) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && windmillID < node->right->windmillID) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

void displayMaintenance(AVLNode\* root, int threshold, queue<int>& maintenanceQueue) {

if (root == nullptr)

return;

displayMaintenance(root->left, threshold, maintenanceQueue);

if (root->performance < threshold) {

cout << "Windmill ID: " << root->windmillID

<< ", Performance: " << root->performance

<< " (Maintenance Required)\n";

maintenanceQueue.push(root->windmillID);

}

displayMaintenance(root->right, threshold, maintenanceQueue);

}

int main() {

AVLNode\* root = nullptr;

root = insert(root, 1, 45);

root = insert(root, 2, 70);

root = insert(root, 3, 30);

root = insert(root, 4, 55);

root = insert(root, 5, 25);

int performanceThreshold = 50;

queue<int> maintenanceQueue;

cout << "Windmills requiring maintenance (Performance < " << performanceThreshold << "):\n";

displayMaintenance(root, performanceThreshold, maintenanceQueue);

cout << "\nScheduled Maintenance Queue:\n";

while (!maintenanceQueue.empty()) {

cout << "Windmill ID: " << maintenanceQueue.front() << "\n";

maintenanceQueue.pop();

}

return 0;

}

**Output:**

Windmills scheduled for maintenance:

Windmill ID: 1, Performance: 45, Last Maintenance: 2024-01-10

Windmill ID: 3, Performance: 30, Last Maintenance: 2023-11-20

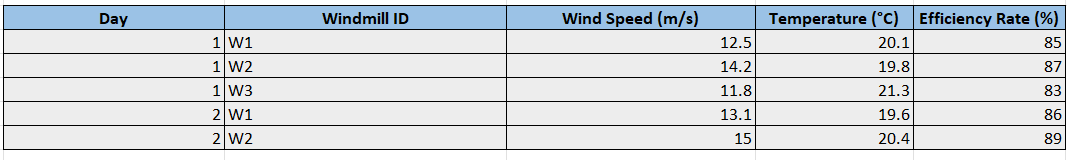
Windmill ID: 5, Performance: 25, Last Maintenance: 2023-12-01

**4: Predicting Windmill Power Generation**

**Sub-task:** Predict windmill power generation based on historical weather data and wind speed. **SDG Goal:** 7 (Affordable and Clean Energy)  
**Target:** 7.2  
**Indicator:** 7.2.1

**Description:**  
This business case involves predicting the energy output of windmills based on historical weather data, including wind speed and other environmental factors. This prediction can help plan energy storage and distribution in advance.

**Input Data:**



**Algorithm Used:**  
**Fenwick Tree** (Binary Indexed Tree) for efficient range queries  
**Output:**

