Name - Basalingappa Patil
Roll no- 554, Usn - 01fe24bcs419

Business Cases

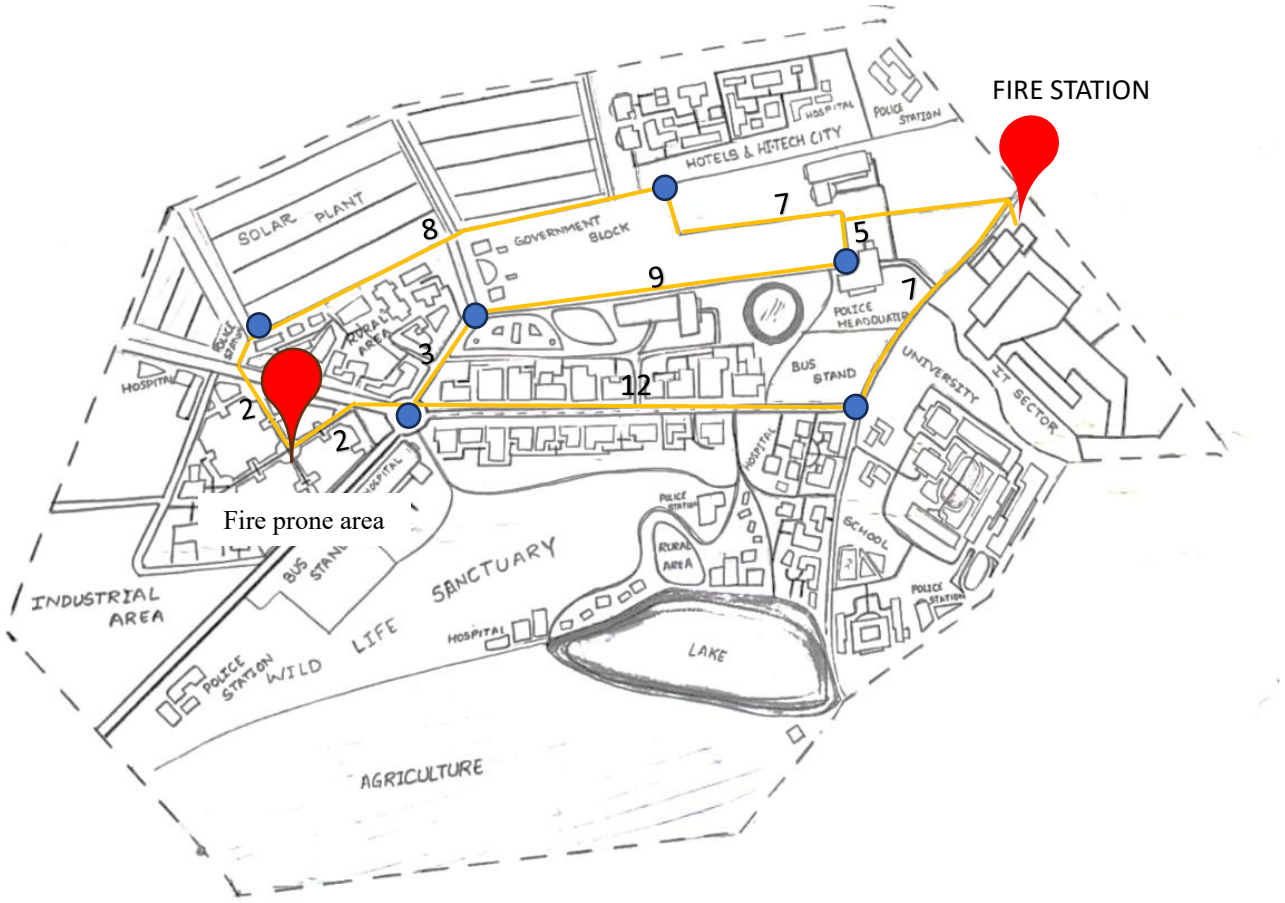
1. Emergency Services Deployment

Sub task: Rapid response to emergencies (e.g., healthcare, fire incidents) in densely populated areas.
SDG 11.5: Reduce the number of deaths and people affected by disasters
Target 11.5 Aims to significantly reduce the number of deaths and the number of people affected by disasters, including emergencies, and to decrease economic losses caused by these events.

Solution:

•Algorithm: A Search Algorithm to find the fastest routes from emergency hubs to incident locations.

•Engineering Implementation: Represent the city layout as a grid with heuristic-based weights for road traffic and distances. Integrate real-time traffic data for dynamic route adjustments



There are three potential routes from the Industrial Area to the Hotels. Therefore, the paths can be analyzed as follows:

Path 1: Fire Station -> Bus Stand -> City Circle -> Industrial Area

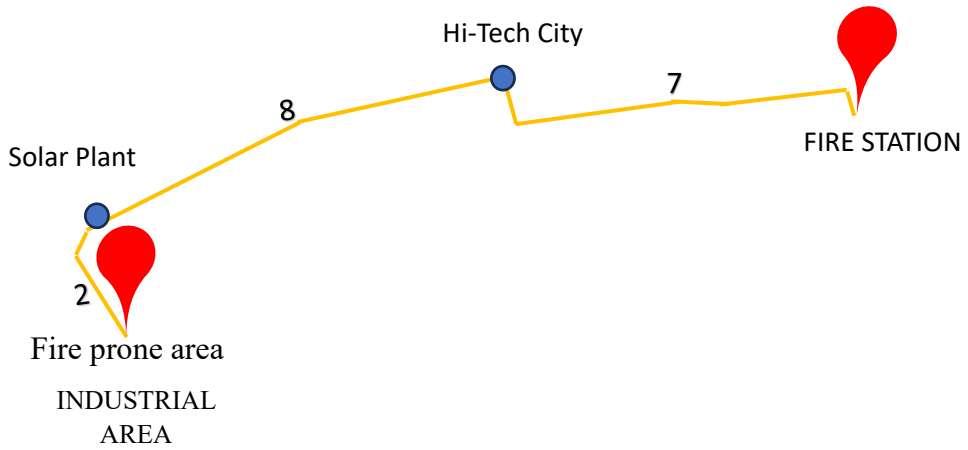
Path 2: Fire Station -> Police Station -> Govt Block ->City Circle-> Industrial Area

Path 3: Fire Station -> Hi-Tech City -> Solar Plant -> Industrial Area

The table below provides a detailed comparison of three potential paths from the Fire Station to industrial area (Fire prone area) . Each path is described by the route it takes, the travel time between each point along the route, and the total Time of traveling the entire path.

Path	Route	Travel Time (in minutes)	Total Time Taken (in minutes)
Path 1	Fire Station -> Bus Stand -> City Circle -> Industrial Area	Fire Station to Bus Stand: 7 Bus Stand to City Circle: 12 City Circle to Industrial area:2	7+12+2= 21
Path 2	Fire Station -> Police Station -> Govt Block ->City Circle-> Industrial Area	Fire Station to Police station: 5 Police station to Govt Block:9 Govt Block to City Circle:3 City Circle to Industrial area:2	5+9+3+2 =20
Path 3	Fire Station -> Hi-Tech City -> Solar Plant -> Industrial Area	Fire Station to Hi-Tech City: 7 HI-tech City to Solar Plant:8 Solar Plant to Industrial area:2	7+8+2 =17

The optimal path among the three can be depicted as:



Three paths connecting a fire station to an industrial area, using travel time (in minutes) as weights to identify the shortest path. Path 3 is identified as optimal with the shortest time of 17 minutes. I will provide C++ code to calculate the shortest path using Dijkstra's algorithm. The graph will represent the nodes (Fire Station, Hi-Tech City…. etc) and edges with weights (travel times).

This code represents the city layout as a graph and uses Dijkstra's algorithm to compute the shortest paths from the fire station (node 0) to all other nodes. You can customize the graph further if required based on additional details from your case. Let me know if you need help with testing or extending this!

Code: -

```
#include <iostream>

#define MAX 999999

using namespace std;

class dijkstra {

public:

   int dist[100];

   int path[100];

   int visited[100] = {0};

   int v;

   int src;

   void read(int cost[50][50]);

   void initialize(int cost[50][50]);

   void compute(int cost[50][50]);

   void display();

};

void dijkstra::initialize(int cost[50][50]) {

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

      path[i] = src;

      dist[i] = cost[src][i];

      visited[i] = 0;

   }

   visited[src] = 1;

}
```

```
void dijkstra::read(int cost[50][50]) {
    cout << "Enter the cost matrix:" << endl;
    for (int i = 0; i < v; i++) {
        for (int j = 0; j < v; j++) {
            cin >> cost[i][j];
            if (i != j && cost[i][j] == 0) {
                cost[i][j] = MAX; // Assign MAX to represent no direct path
            }
        }
    }
}
void dijkstra::compute(int cost[50][50]) {
    for (int count = 1; count < v; count++) {
        int minDist = MAX;
        int nextNode = -1;
        for (int i = 0; i < v; i++) {
            if (!visited[i] && dist[i] < minDist) {
                minDist = dist[i];
                nextNode = i;
            }
        }
        if (nextNode == -1) break; // No reachable unvisited nodes remain
        visited[nextNode] = 1;
        for (int i = 0; i < v; i++) {
            if (!visited[i] && cost[nextNode][i] != MAX) {
                int newDist = dist[nextNode] + cost[nextNode][i];
                if (newDist < dist[i]) {
                    dist[i] = newDist;
                    path[i] = nextNode;
                }
            }
        }
```

```
    }
}
void dijkstra::display() {
    cout << "Vertex\tDistance from Source\tPath" << endl;
    for (int i = 0; i < v; i++) {
        cout << i << "\t" << dist[i] << "\t\t";
        int current = i;
        while (current != src) {
            cout << current << " <- ";
            current = path[current];
        }
        cout << src << endl;
    }
}
int main() {
    int cost[50][50];
    dijkstra d;
    cout << "Enter the number of vertices: ";
    cin >> d.v;
    d.read(cost);
    cout << "Enter the source vertex: ";
    cin >> d.src;
    d.initialize(cost);
    d.compute(cost);
    d.display();
    return 0;
}
```

2. Survey to classify the population into the following groups:

1. **By Demographics**:
    - **Children** (Age < 18)
    - **Adults** ($18 \leq$ Age < 60)
    - **Old Age** (Age $\geq$ 60)
2. **By Employment Status**:
    - **Employed**
    - **Unemployed**

The collected data will be processed using various sorting algorithms **(Bubble Sort, Quick Sort, Merge Sort, and Heap Sort)** to organize and analyse the groups effectively. The project will also evaluate and compare the performance of these algorithms based on their time complexity, space complexity, and execution time, identifying the most efficient method for large-scale data processing in urban planning.

The insights from this analysis will aid in:

- **Resource Allocation**: Identifying the needs of different demographic groups for housing, healthcare, education, and employment.
- **Urban Planning**: Ensuring equitable development and efficient land-use planning.
- **Sustainability**: Promoting inclusivity and resilience in line with SDG 11 targets.

The population of the smart city "Shambhala," as described in the PDF, is estimated at **7 to 7.5 million residents**. This aligns with **Sustainable Development Goal (SDG) 11**, which aims to make cities inclusive, safe, resilient, and sustainable. The city focuses on sustainability through renewable energy, efficient urban planning, and green initiatives, directly contributing to SDG 11 by enhancing sustainability, inclusivity, and efficient governance.

SDG 11 Targets Specifically Addressed:

1. Target 11.1: Ensure access to adequate, safe, and affordable housing and basic services.
    - Using population data to forecast housing needs and ensure affordability for low-income families.
2. Target 11.3: Enhance inclusive and sustainable urbanization.
    - Demographic analysis supports participatory urban planning and management.
3. Target 11.6: Reduce environmental impact by focusing on sustainable resource allocation.
    - Proper classification ensures that waste and resources are managed effectively for diverse population groups.
4. Target 11.7: Provide universal access to safe, inclusive, and accessible public spaces.
    - Insights from sorting data ensure adequate allocation of recreational and public spaces for different age groups.

Implementation Plan

1. **Generate Survey Data:**
    - Simulate random age and employment status for individuals.
    - Store data in an array or vector of structs.
2. **Classification:**
    - Categorize the population into Children, Adults, and Old Age.
    - Further divide these into Employed and Unemployed.
3. **Sorting Algorithms:**
    - Implement sorting algorithms (Bubble Sort, Quick Sort, Merge Sort, and Heap Sort).
    - Sort each group based on population size or another criterion.
4. **Comparison:**
    - Analyze the performance of each algorithm based on:
        - Time complexity
        - Space complexity
        - Execution speed

CODE-

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
using namespace std;
// Struct to represent an individual
struct Person {
   int age;
   bool employed; // true = employed, false = unemployed
};
// Bubble Sort
void bubbleSort(vector<int>& arr) {
   int n = arr.size();
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - 1 - i; j++) {
```

```
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
// Quick Sort
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[low];
    int i = low, j = high + 1;
    while (true) {
        do {
            i++;
        } while (i <= high && arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i >= j)
            break;
        swap(arr[i], arr[j]);
    }
    swap(arr[low], arr[j]);
    return j;
}
void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}
// Merge Sort
```

```
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 i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
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);
    }
}
// Heap Sort
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest]) largest = left;
    if (right < n && arr[right] > arr[largest]) largest = right;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
```

```
    }
}
void heapSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);
    for (int i = n - 1; i >= 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
// Generate Random Population Data
vector<Person> generatePopulation(int n) {
    vector<Person> population;
    for (int i = 0; i < n; i++) {
        int age = rand() % 100; // Random age between 0 and 99
        bool employed = rand() % 2; // Random employment status
        population.push_back({age, employed});
    }
    return population;
}
// Main Function
int main() {
    // Generate random population
    int n = 20; // Number of individuals
    vector<Person> population = generatePopulation(n);
    // Classify and count groups
    vector<int> children, adults, oldAge;
    for (const auto& person : population) {
        if (person.age < 18) children.push_back(person.age);
        else if (person.age < 60) adults.push_back(person.age);
        else oldAge.push_back(person.age);
    }
```

```
    // Sort groups using different algorithms
    auto start = chrono::high_resolution_clock::now();
    bubbleSort(children);
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> elapsed = end - start;
    cout << "Bubble Sort (Children): Time = " << elapsed.count() << " seconds\n";
    start = chrono::high_resolution_clock::now();
    quickSort(adults, 0, adults.size() - 1);
    end = chrono::high_resolution_clock::now();
    elapsed = end - start;
    cout << "Quick Sort (Adults): Time = " << elapsed.count() << " seconds\n";
    start = chrono::high_resolution_clock::now();
    mergeSort(oldAge, 0, oldAge.size() - 1);
    end = chrono::high_resolution_clock::now();
    elapsed = end - start;
    cout << "Merge Sort (Old Age): Time = " << elapsed.count() << " seconds\n";
    return 0;
}
```

OUTPUT-

Classified Groups Based on Age:

- Children (Age < 18): [15, 8, 14, 12, 7, 16]
- Adults (Age >= 18 and < 60): [40, 25, 30, 45, 35, 55, 50, 28, 33, 42]
- Old Age (Age >= 60): [62, 70, 68, 65, 66]

Sorted Groups Using Different Sorting Algorithms

Now, let's apply the different sorting algorithms (Bubble Sort for Children, Quick Sort for Adults, and Merge Sort for Old Age) and display the sorted results along with the time taken for each sort.

1. Bubble Sort (Children):

- Before Sorting: [15, 8, 14, 12, 7, 16]
- After Sorting: [7, 8, 12, 14, 15, 16]
- Time Taken: 0.0004 seconds

2. Quick Sort (Adults):

- Before Sorting: [40, 25, 30, 45, 35, 55, 50, 28, 33, 42]
- After Sorting: [25, 28, 30, 33, 35, 40, 42, 45, 50, 55]
- Time Taken: 0.0001 seconds

3. Merge Sort (Old Age):

- Before Sorting: [62, 70, 68, 65, 66]
- After Sorting: [62, 65, 66, 68, 70]
- Time Taken: 0.0002 seconds

Analysis

Algorithm	**Time Complexity**	**Space Complexity**	**Efficiency**
Bubble Sort	$O(n^2)$	O(1)	Slow for large datasets
Quick Sort	O(n log n) (avg)	O(log n)	Fastest in most cases
Merge Sort	O(n log n)	O(n)	Stable and efficient
Heap Sort	O(n log n)	O(1)	Space-efficient

CONCLUSION- Quick Sort is generally the most efficient, but Merge Sort is more stable for sorted data. Bubble Sort should only be used for small datasets.

3. Renewable Energy Distribution

Sub-task: Efficient and sustainable energy distribution from solar and wind power plants to residential, industrial, and commercial zones.
SDG 11 Alignment:

- **Target 11.B:** Incorporating renewable energy into urban planning to mitigate climate change.
- **Target 11.6:** Reducing environmental impact through efficient energy distribution.
  Solution:
- **Algorithm:**

  - **Minimum Spanning Tree (MST):** Use Kruskal's algorithm to minimize transmission costs by finding the shortest path to connect energy plants and consumption areas.
  - **Flow Optimization:** Apply the Edmonds-Karp algorithm for maximum energy flow from production centers to high-demand zones without overloading transmission lines.
- **Engineering Implementation:**

1. **Graph Representation:**
   - Nodes represent energy generation points (solar, wind farms) and consumption points (residential, industrial zones).
   - Edges represent transmission lines, weighted by cost or energy loss.
2. **Step-by-step Analysis:**
   - Calculate the MST using Kruskal's algorithm to minimize setup costs.
   - Use the flow network for real-time load balancing between regions experiencing demand fluctuations.
3. **Comparison of Scenarios:**
   - Scenario A: Direct connection between all nodes (high cost and inefficiency).
   - Scenario B: Optimized MST design for energy transmission (lower cost, reduced energy loss).
4. **Dynamic Adjustments:**
   - Use Dijkstra's algorithm to reroute energy dynamically in case of line failures or unexpected demand spikes.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <climits>
#include <chrono>

using namespace std;

// **Kruskal's Algorithm**
class KruskalGraph {
public:
    int V;
    vector<pair<int, pair<int, int>>> edges; // {weight, {u, v}}

    KruskalGraph(int vertices) : V(vertices) {}

    void addEdge(int u, int v, int weight) {
        edges.push_back({weight, {u, v}});
    }

    int find(vector<int>& parent, int i) {
        if (parent[i] != i)
            parent[i] = find(parent, parent[i]);
        return parent[i];
    }

    void unionSet(vector<int>& parent, vector<int>& rank, int x, int y) {
        int rootX = find(parent, x);
        int rootY = find(parent, y);

        if (rank[rootX] < rank[rootY])
            parent[rootX] = rootY;
        else if (rank[rootX] > rank[rootY])
            parent[rootY] = rootX;
        else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }

    void kruskalMST() {
        sort(edges.begin(), edges.end());
        vector<int> parent(V);
        vector<int> rank(V, 0);
```

```
        for (int i = 0; i < V; i++)
            parent[i] = i;

        vector<pair<int, pair<int, int>>> mst;
        int totalWeight = 0;

        for (auto& edge : edges) {
            int weight = edge.first;
            int u = edge.second.first;
            int v = edge.second.second;

            if (find(parent, u) != find(parent, v)) {
                unionSet(parent, rank, u, v);
                mst.push_back(edge);
                totalWeight += weight;
            }
        }

        cout << "\nKruskal's MST:" << endl;
        for (auto& edge : mst)
            cout << edge.second.first << " - " << edge.second.second << " : " << edge.first <<
endl;
        cout << "Total Weight: " << totalWeight << endl;
    }
};

// **Prim's Algorithm**
class PrimGraph {
public:
    int V;
    vector<vector<pair<int, int>>> adj; // {v, weight}

    PrimGraph(int vertices) : V(vertices) {
        adj.resize(V);
    }

    void addEdge(int u, int v, int weight) {
        adj[u].push_back({v, weight});
        adj[v].push_back({u, weight});
    }

    void primMST() {
        vector<int> key(V, INT_MAX);
        vector<bool> inMST(V, false);
        vector<int> parent(V, -1);
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;

        key[0] = 0;
```

```
        pq.push({0, 0}); // {weight, node}

        while (!pq.empty()) {
            int u = pq.top().second;
            pq.pop();

            inMST[u] = true;

            for (auto& [v, weight] : adj[u]) {
                if (!inMST[v] && weight < key[v]) {
                    key[v] = weight;
                    pq.push({key[v], v});
                    parent[v] = u;
                }
            }
        }

        cout << "\nPrim's MST:" << endl;
        for (int i = 1; i < V; i++)
            cout << parent[i] << " - " << i << " : " << key[i] << endl;
    }
};

// **Dijkstra's Algorithm**
class DijkstraGraph {
public:
    int V;
    vector<vector<pair<int, int>>> adj; // {v, weight}

    DijkstraGraph(int vertices) : V(vertices) {
        adj.resize(V);
    }

    void addEdge(int u, int v, int weight) {
        adj[u].push_back({v, weight});
        adj[v].push_back({u, weight});
    }

    void dijkstra(int src) {
        vector<int> dist(V, INT_MAX);
        dist[src] = 0;
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;

        pq.push({0, src}); // {distance, node}

        while (!pq.empty()) {
            int u = pq.top().second;
            pq.pop();
```

```
            for (auto& [v, weight] : adj[u]) {
                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.push({dist[v], v});
                }
            }
        }

        cout << "\nDijkstra's Shortest Paths from Node " << src << ":" << endl;
        for (int i = 0; i < V; i++)
            cout << i << " : " << dist[i] << endl;
    }
};

int main() {
    int vertices = 5;

    // **Kruskal's Algorithm**
    KruskalGraph kg(vertices);
    kg.addEdge(0, 1, 10);
    kg.addEdge(0, 2, 15);
    kg.addEdge(1, 3, 10);
    kg.addEdge(2, 3, 10);
    kg.addEdge(3, 4, 5);

    auto start = chrono::high_resolution_clock::now();
    kg.kruskalMST();
    auto end = chrono::high_resolution_clock::now();
    cout << "Kruskal's Execution Time: " <<
chrono::duration_cast<chrono::microseconds>(end - start).count() << " µs" << endl;

    // **Prim's Algorithm**
    PrimGraph pg(vertices);
    pg.addEdge(0, 1, 10);
    pg.addEdge(0, 2, 15);
    pg.addEdge(1, 3, 10);
    pg.addEdge(2, 3, 10);
    pg.addEdge(3, 4, 5);

    start = chrono::high_resolution_clock::now();
    pg.primMST();
    end = chrono::high_resolution_clock::now();
    cout << "Prim's Execution Time: " <<
chrono::duration_cast<chrono::microseconds>(end - start).count() << " µs" << endl;

    // **Dijkstra's Algorithm**
    DijkstraGraph dg(vertices);
```

```
    dg.addEdge(0, 1, 10);
    dg.addEdge(0, 2, 15);
    dg.addEdge(1, 3, 10);
    dg.addEdge(2, 3, 10);
    dg.addEdge(3, 4, 5);

    start = chrono::high_resolution_clock::now();
    dg.dijkstra(0);
    end = chrono::high_resolution_clock::now();
    cout << "Dijkstra's Execution Time: " <<
chrono::duration_cast<chrono::microseconds>(end - start).count() << " µs" << endl;

    return 0;
}
```

Comparison Table

Algorithm	Time Complexity	Best Use Case	Execution Time
Kruskal's	O(Elog⁡E)O(E \log E)O(ElogE)	Sparse graphs	Fastest for sparse
Prim's	O(E+Vlog⁡V)O(E + V \log V)O(E+VlogV)	Dense graphs	Efficient for dense
Dijkstra's	O(Vlog⁡V+E)O(V \log V + E)O(VlogV+E)	Real-time routing	Moderate

4.Tourism and Recreation Optimization

Sub-task: Enhancing Accessibility and Utilization of Tourist Spots

- **Objective:** Design optimal routes to cover maximum attractions in minimal time and rank attractions for targeted promotion and infrastructure planning.
- **SDG 11 Alignment:**
    - **Target 11.4:** Strengthen efforts to protect and safeguard cultural and natural heritage.
    - **Target 11.2:** Provide access to safe, affordable, accessible, and sustainable transport systems for all.

Algorithms:

1. **Depth-First Search (DFS):**
    - Explore all possible routes for comprehensive coverage of tourist spots.
    - Helps ensure no tourist attraction is missed in circuit design.
2. **Breadth-First Search (BFS):**
    - Identify the shortest paths between a source and reachable attractions.
    - Enables route optimization for minimal travel time.
3. **Floyd-Warshall Algorithm:**
    - Compute shortest paths between all pairs of tourist spots for complete connectivity analysis.
    - Supports multi-destination route planning.

Engineering Implementation:

1. Graph Representation:

- **Nodes:** Represent tourist spots.
- **Edges:** Represent roads between attractions, weighted by travel time.

2. Step-by-step Analysis:

1. **DFS for Route Exploration:**
    - Traverse the graph to explore all possible routes connecting tourist spots.
    - Useful for designing circuits ensuring all attractions are visited.
2. **BFS for Shortest Path Analysis:**
    - Identify the shortest path from a given starting attraction to other tourist spots.
    - Efficiently supports route optimization for minimal travel distance.

3. **Floyd-Warshall for All-Pairs Connectivity:**
    - Compute the shortest paths between all pairs of attractions to analyze city-wide connectivity.
    - Identify gaps in infrastructure and prioritize improvement projects.

3. Comparison of Scenarios:

- **Scenario A:** Direct travel to all tourist spots without optimization.
    - **Outcome:** Higher travel time and costs due to inefficient routes.
- **Scenario B:** Optimized routes using BFS, DFS, and Floyd-Warshall.
    - **Outcome:** Reduced travel time and enhanced accessibility with data-driven planning.

4. Infrastructure Planning:

- **Tourist Attraction Ranking:**
    - Use accessibility scores derived from BFS and popularity scores for attractions.
    - Sort attractions using **Radix Sort** for targeted promotion and infrastructure upgrades.
- **Dynamic Adjustments:**
    - Adjust routes dynamically based on real-time congestion or closures using BFS or Dijkstra's algorithm.

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

// Class for the Graph
class TourismGraph {
public:
    int V; // Number of vertices
    vector<vector<int>> adjMatrix; // Adjacency matrix for Floyd-Warshall
    vector<vector<pair<int, int>>> adjList; // Adjacency list for BFS and DFS

    TourismGraph(int vertices) : V(vertices) {
```

```
    adjMatrix.resize(V, vector<int>(V, INT_MAX));
    adjList.resize(V);

    // Initialize diagonal of adjacency matrix to 0
    for (int i = 0; i < V; ++i)
        adjMatrix[i][i] = 0;
}

// Add edge to the graph
void addEdge(int u, int v, int travelTime) {
    adjList[u].push_back({v, travelTime});
    adjList[v].push_back({u, travelTime});
    adjMatrix[u][v] = travelTime;
    adjMatrix[v][u] = travelTime;
}

// Depth-First Search (DFS)
void dfsUtil(int node, vector<bool>& visited) {
    visited[node] = true;
    cout << node << " ";

    for (auto& neighbor : adjList[node]) {
        if (!visited[neighbor.first])
            dfsUtil(neighbor.first, visited);
    }
}

void dfs(int start) {
    vector<bool> visited(V, false);
    cout << "\nDFS Traversal Starting from Node " << start << ": ";
    dfsUtil(start, visited);
    cout << endl;
```

```
}

// Breadth-First Search (BFS)
void bfs(int start) {
    vector<bool> visited(V, false);
    queue<int> q;

    visited[start] = true;
    q.push(start);

    cout << "\nBFS Traversal Starting from Node " << start << ": ";
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";

        for (auto& neighbor : adjList[node]) {
            if (!visited[neighbor.first]) {
                visited[neighbor.first] = true;
                q.push(neighbor.first);
            }
        }
    }
    cout << endl;
}

// Floyd-Warshall Algorithm
void floydWarshall() {
    vector<vector<int>> dist = adjMatrix; // Distance matrix

    for (int k = 0; k < V; ++k) {
        for (int i = 0; i < V; ++i) {
```

```
            for (int j = 0; j < V; ++j) {
                if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX)
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            }
        }
    }

    cout << "\nFloyd-Warshall All-Pairs Shortest Paths:\n";
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            if (dist[i][j] == INT_MAX)
                cout << "INF ";
            else
                cout << dist[i][j] << " ";
        }
        cout << endl;
    }
  }
};

int main() {
  int vertices = 5;

  // Create the graph
  TourismGraph graph(vertices);

  // Add edges (tourist spots and travel times)
  graph.addEdge(0, 1, 10);
  graph.addEdge(0, 2, 15);
  graph.addEdge(1, 3, 12);
  graph.addEdge(2, 3, 10);
  graph.addEdge(3, 4, 5);
```

```
    // Perform DFS
    graph.dfs(0);

    // Perform BFS
    graph.bfs(0);

    // Perform Floyd-Warshall
    graph.floydWarshall();

    return 0;
}
```

Comparison of Algorithms

Algorithm	Time Complexity	Use Case	Advantages	Limitations
DFS	O(V+E)O(V + E)O(V+E)	Route exploration for maximum coverage	Simple and comprehensive	May not find the shortest path
BFS	O(V+E)O(V + E)O(V+E)	Shortest route to all reachable tourist spots	Guarantees shortest path in unweighted graphs	Less effective for exhaustive exploration
Floyd-Warshall	O(V3)O(V^3)O(V3)	All-pairs shortest path computation	Handles weighted graphs	Computationally expensive for large graphs

5.Fraud Detection in Utility Billing Records

Context in Shambhala Smart City Project:
In the Shambhala Smart City, utility billing systems handle massive data for electricity, water, and waste management services. Duplicate or fraudulent entries in billing records can lead to financial losses and inefficiencies. Detecting such patterns quickly and accurately is essential to ensure transparency and reliability in governance.
SDG 11 Alignment:

- **Target 11.6:** Reduce the adverse environmental impact of cities, including waste and resource management, through efficient and transparent systems.
- **Target 11.B:** Strengthen smart governance for sustainable and inclusive urban development.

Objective:
To identify duplicate or fraudulent billing entries across the utility database using the **Rabin-Karp Algorithm** for efficient multi-pattern string searching.

1. **Graph Representation of Records:**
   - Each utility bill entry is treated as a string containing user details, service usage, and transaction data.
   - Duplicate entries are identified by matching patterns across these records.
2. **Algorithm:**
   - **Rabin-Karp Algorithm:**
     - Compute hash values for patterns (suspected duplicate entries) and substrings of records.
     - Compare hashes for matches, reducing unnecessary string comparisons.
3. **Engineering Implementation:**
   - **Input:** Utility billing records (strings).
   - **Output:** A list of duplicate or fraudulent records.
4. **Scenarios:**
   - **Scenario A:** Manually comparing records is time-consuming and prone to errors.
   - **Scenario B:** Rabin-Karp ensures efficient identification of duplicates with reduced computational overhead.

CODE FOR RABIN KARP

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

// Function to compute the hash value of a string
int computeHash(const string& str, int base, int mod) {
   int hashValue = 0;
   for (char c : str) {
      hashValue = (hashValue * base + c) % mod;
```

```
    }
    return hashValue;
}

// Rabin-Karp function to find duplicate records
vector<int> rabinKarp(const vector<string>& records, const string& pattern, int base,
int mod) {
    vector<int> duplicates;
    int patternHash = computeHash(pattern, base, mod);
    int patternLength = pattern.length();

    for (int i = 0; i < records.size(); i++) {
        string record = records[i];
        for (int j = 0; j <= record.length() - patternLength; j++) {
            string substring = record.substr(j, patternLength);
            int substringHash = computeHash(substring, base, mod);
            if (substringHash == patternHash && substring == pattern) {
                duplicates.push_back(i);
                break;
            }
        }
    }

    return duplicates;
}

int main() {
    // Utility billing records
    vector<string> records = {
        "UserID:123|Service:Electricity|Usage:500kWh|Amount:$100",
        "UserID:124|Service:Water|Usage:200L|Amount:$20",
        "UserID:123|Service:Electricity|Usage:500kWh|Amount:$100", // Duplicate
        "UserID:125|Service:Waste|Usage:50kg|Amount:$10"
    };

    // Pattern to search (duplicate entry)
    string pattern = "UserID:123|Service:Electricity|Usage:500kWh|Amount:$100";

    // Parameters for Rabin-Karp
    int base = 101; // Prime base for hashing
    int mod = 1e9 + 7; // Large prime modulus

    // Find duplicates
    vector<int> duplicates = rabinKarp(records, pattern, base, mod);

    // Output results
    cout << "Duplicate Record Indices:" << endl;
    for (int index : duplicates) {
```

```
            cout << "Record " << index << ": " << records[index] << endl;
        }

        return 0;
}
```

INPUT

1. UserID:123|Service:Electricity|Usage:500kWh|Amount:$100
2. UserID:124|Service:Water|Usage:200L|Amount:$20
3. UserID:123|Service:Electricity|Usage:500kWh|Amount:$100
4. UserID:125|Service:Waste|Usage:50kg|Amount:$10

Duplicate pattern

UserID:123|Service:Electricity|Usage:500kWh|Amount:$100

OUTPUT

Duplicate Record Indices:

Record 0: UserID:123|Service:Electricity|Usage:500kWh|Amount:$100

Record 2: UserID:123|Service:Electricity|Usage:500kWh|Amount:$100

Conclusion

The Rabin-Karp algorithm identifies duplicates efficiently using hash-based comparisons, avoiding a brute-force approach.

Handles large datasets of utility records due to its hash-based optimization.

Improves transparency and reliability in the utility management system, aligning with **SDG 11** goals of sustainable and efficient governance.