IT SECTOR:

1. **Sub-task:** Efficiently manage and search large amounts of employee data such as names, employee IDs, and departments.

**SDG Goal:** 8 (Decent Work and Economic Growth)  
**Target:** 8.3  
**Indicator:** 8.3.1

**Description:**

In large organizations, managing and searching through employee records (such as names, employee IDs, and departments) can become cumbersome as the database grows. To solve this issue, we can implement a **Trie** data structure that enables fast insertion, search, and prefix-based search operations. By organizing the employee names into a Trie, we can quickly find employees based on full name or search for all employees whose names start with a given prefix. This will improve efficiency in accessing employee information, making it suitable for real-time search scenarios such as HR management systems, payroll systems, or employee directories.

**Input Data:**

* **Employee Records**: Each employee record consists of the following fields:
  + **Employee ID** (unique identifier)
  + **Employee Name** (string)
  + **Department** (string)
  + **Location** (string)

**Example Input** (Employee Records):



* Algorithm used:
* **Trie Data Structure**: For storing and searching employee names efficiently.
* **Operations Implemented**:
  + **Insert**: Insert employee names into the Trie.
  + **Search**: Search for an exact match of an employee's name.
  + **Prefix Search**: Find all employee names that start with a specific prefix (e.g., all employees starting with "John").

**Output:**

1. **Search Results**: Check if an employee name exists in the Trie.
   * Example Output:
     + **Searching for "John Doe"**: Found
     + **Searching for "Chris Brown"**: Found
     + **Searching for "Mike Johnson"**: Not Found
2. **Prefix Search Results**: List all employees whose names start with a specific prefix.
   * Example Output:
     + **Employees with names starting with "J"**:
       - John Doe
       - Jane Smit

CODE:

#include <iostream>

#include <unordered\_map>

#include <vector>

using namespace std;

// Trie Node Structure

struct TrieNode {

unordered\_map<char, TrieNode\*> children;

bool isEndOfWord;

TrieNode() : isEndOfWord(false) {}

};

// Trie Data Structure

class Trie {

private:

TrieNode\* root;

public:

Trie() {

root = new TrieNode();

}

// Insert an employee name into the Trie

void insert(const string& name) {

TrieNode\* node = root;

for (char ch : name) {

if (node->children.find(ch) == node->children.end()) {

node->children[ch] = new TrieNode();

}

node = node->children[ch];

}

node->isEndOfWord = true;

}

// Search for an employee by name in the Trie

bool search(const string& name) {

TrieNode\* node = root;

for (char ch : name) {

if (node->children.find(ch) == node->children.end()) {

return false;

}

node = node->children[ch];

}

return node->isEndOfWord;

}

// Search for all names starting with a prefix

void prefixSearch(TrieNode\* node, const string& prefix, vector<string>& result, string current) {

if (node->isEndOfWord) {

result.push\_back(current);

}

for (auto& child : node->children) {

prefixSearch(child.second, prefix, result, current + child.first);

}

}

vector<string> findAllWithPrefix(const string& prefix) {

TrieNode\* node = root;

vector<string> result;

for (char ch : prefix) {

if (node->children.find(ch) == node->children.end()) {

return result; // No such prefix

}

node = node->children[ch];

}

prefixSearch(node, prefix, result, prefix);

return result;

}

};

// Test Trie with Employee Data

int main() {

Trie trie;

// Inserting employee names

trie.insert("John Doe");

trie.insert("Jane Smith");

trie.insert("Michael Johnson");

trie.insert("Emily Davis");

trie.insert("Chris Brown");

trie.insert("Mary Williams");

trie.insert("David Harris");

trie.insert("Sarah Martinez");

trie.insert("William Lee");

trie.insert("Lisa Walker");

// Searching for an employee

cout << "Searching for 'John Doe': " << (trie.search("John Doe") ? "Found" : "Not Found") << endl;

cout << "Searching for 'Chris Brown': " << (trie.search("Chris Brown") ? "Found" : "Not Found") << endl;

// Searching by prefix

vector<string> result = trie.findAllWithPrefix("J");

cout << "\nEmployees with names starting with 'J':\n";

for (const string& name : result) {

cout << name << endl;

}

return 0;

}

2.**Sub-task:** Efficiently validate user credentials in IT systems by searching for username-password pairs.

**SDG Goal:** 9 (Industry, Innovation, and Infrastructure)  
**Target:** 9.5  
**Indicator:** 9.5.2

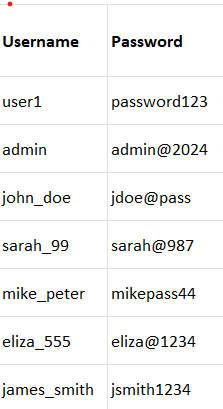
**Description:**

In modern IT systems, securely validating user credentials (i.e., usernames and passwords) is a critical task. Traditional methods such as linear search or direct comparison may be inefficient, especially with a large number of user records. The **Rabin-Karp string matching algorithm** can be applied here to efficiently search for username-password pairs within a database, where the hash value of each password is computed and used to verify its correctness. This improves the efficiency of the credential verification process in large-scale systems.

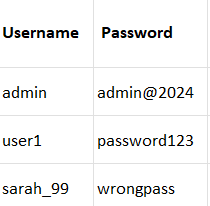
**Input Data:**

* **Username-Password Database**: A list of usernames and their corresponding passwords stored in the system.
* **User Credentials to Validate**: Username and password input by the user attempting to log in.

**Example Input** (Username-Password Database and User Credentials):

****

* **User Credentials to Validate**:

****

**Algorithm Used:**

* **Rabin-Karp String Matching Algorithm**: Used for efficiently searching the given user credentials (username-password pairs) against a database using hashing.
  + **Hash Function**: A rolling hash function is used to calculate hash values of the passwords, enabling efficient validation.

**Output:**

1. **Valid Credentials:**
   * Example Output:
     + **For input username "admin" and password "admin@2024"**: Valid credentials.
     + **For input username "user1" and password "password123"**: Valid credentials.
2. **Invalid Credentials:**
   * Example Output:
     + **For input username "sarah\_99" and password "wrongpass"**: Invalid credentials.

CODE:

2.#include <iostream>

#include <unordered\_map>

#include <string>

using namespace std;

// Rabin-Karp String Matching Algorithm

int hashFunction(string str) {

int hash = 0;

int prime = 31; // A prime number used for hashing

for (char c : str) {

hash = hash \* prime + c;

}

return hash;

}

bool validateCredentials(const unordered\_map<string, string>& database, const string& username, const string& password) {

// Check if username exists in the database

if (database.find(username) != database.end()) {

// Validate password using Rabin-Karp algorithm (hash matching)

return hashFunction(database.at(username)) == hashFunction(password);

}

return false; // Username not found

}

int main() {

// Database of username-password pairs

unordered\_map<string, string> userDatabase = {

{"user1", "password123"},

{"admin", "admin@2024"},

{"john\_doe", "jdoe@pass"},

{"sarah\_99", "sarah@987"},

{"mike\_peter", "mikepass44"},

{"eliza\_555", "eliza@1234"},

{"james\_smith", "jsmith1234"}

};

// User credentials to validate

string username, password;

// Simulating a login attempt

cout << "Enter username: ";

cin >> username;

cout << "Enter password: ";

cin >> password;

if (validateCredentials(userDatabase, username, password)) {

cout << "Valid credentials!" << endl;

} else {

cout << "Invalid credentials!" << endl;

}

return 0;

}

3. **Sub-task:** Efficiently organize IT projects based on priority or deadlines using sorting algorithms.

**SDG Goal:** 9 (Industry, Innovation, and Infrastructure)  
**Target:** 9.5  
**Indicator:** 9.5.2

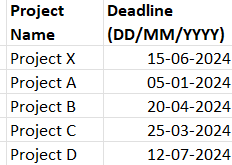
**Description:**

In modern IT companies, managing projects based on their priority or deadlines is crucial for timely delivery. A sorting algorithm can help organize the list of projects, ensuring that higher-priority or earlier deadline projects are worked on first. In this case, we'll use **Merge Sort** (a divide-and-conquer sorting algorithm) to arrange the projects based on their deadlines, which will allow project managers to prioritize tasks more efficiently and improve workflow management.

**Input Data:**

* **List of IT projects**: Each project has a name and a deadline.
* **Sorting Criteria**: The projects should be sorted by their deadlines in ascending order (earliest deadlines first).

**Example Input:**



**Algorithm Used:**

* **Merge Sort Algorithm**: A divide-and-conquer algorithm that splits the list into smaller sublists, sorts them, and then merges them back together in sorted order.

**Output:**

1. **Sorted Projects by Deadline**:
   * Example Output:
     + **Sorted Projects:**
       1. Project A - 05/01/2024
       2. Project C - 25/03/2024
       3. Project B - 20/04/2024
       4. Project X - 15/06/2024
       5. Project D - 12/07/2024

CODE:

3.#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

using namespace std;

// Helper function to compare dates in the format DD/MM/YYYY

bool compareDates(const string& date1, const string& date2) {

int day1 = stoi(date1.substr(0, 2));

int month1 = stoi(date1.substr(3, 2));

int year1 = stoi(date1.substr(6, 4));

int day2 = stoi(date2.substr(0, 2));

int month2 = stoi(date2.substr(3, 2));

int year2 = stoi(date2.substr(6, 4));

// Compare year, then month, then day

if (year1 != year2) return year1 < year2;

if (month1 != month2) return month1 < month2;

return day1 < day2;

}

// Merge function to combine two sorted sublists

void merge(vector<pair<string, string>>& projects, int left, int mid, int right) {

vector<pair<string, string>> temp;

int i = left, j = mid + 1;

while (i <= mid && j <= right) {

if (compareDates(projects[i].second, projects[j].second)) {

temp.push\_back(projects[i]);

i++;

} else {

temp.push\_back(projects[j]);

j++;

}

}

while (i <= mid) {

temp.push\_back(projects[i]);

i++;

}

while (j <= right) {

temp.push\_back(projects[j]);

j++;

}

for (int i = left; i <= right; i++) {

projects[i] = temp[i - left];

}

}

// Merge Sort function to sort projects by deadline

void mergeSort(vector<pair<string, string>>& projects, int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(projects, left, mid);

mergeSort(projects, mid + 1, right);

merge(projects, left, mid, right);

}

}

int main() {

// List of IT projects with deadlines

vector<pair<string, string>> projects = {

{"Project X", "15/06/2024"},

{"Project A", "05/01/2024"},

{"Project B", "20/04/2024"},

{"Project C", "25/03/2024"},

{"Project D", "12/07/2024"}

};

// Apply Merge Sort to the projects list based on deadlines

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

// Output sorted projects

cout << "Sorted Projects by Deadline:\n";

for (const auto& project : projects) {

cout << project.first << " - " << project.second << endl;

}

return 0;

}

AGRICULTURE:

1. **Sub-task:** Efficiently analyze crop yields over different regions of the agricultural field using segment trees.

**SDG Goal:** 2 (Zero Hunger)  
**Target:** 2.4  
**Indicator:** 2.4.1

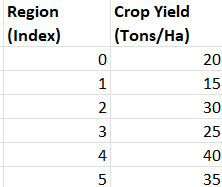
**Description:**

In agricultural fields, crop yields can vary significantly across different regions. Monitoring these yields is important for determining the overall health of the crops, identifying problem areas, and optimizing resource allocation (e.g., water, fertilizer). A **Segment Tree** can be used to efficiently store and query crop yields over a range of regions in the field. This structure allows us to answer range queries (e.g., total yield or average yield over a specific area) in logarithmic time, which is much more efficient than a simple linear search.

**Input Data:**

* **Crop yields**: An array of crop yields representing different regions of the field.
* **Query**: Range queries that ask for the total or average crop yield over a specified segment (region) of the field.

**Example Input:**



**Algorithm Used:**

* **Segment Tree**: A data structure that is particularly efficient for answering range queries (such as sum, minimum, maximum) and performing point updates.

**Output:**

* **Range Query Result**: The result of querying the sum or average of crop yields over a specified range.
  + Example Output: For the range query from region 1 to 4, the sum of crop yields is **105 tons/ha**.

**CODE:**

**1.#include <iostream>**

**#include <vector>**

**using namespace std;**

**// Segment Tree Class**

**class SegmentTree {**

**private:**

**vector<int> tree;**

**int n;**

**// Utility function to build the segment tree**

**void build(const vector<int>& arr, int node, int start, int end) {**

**if (start == end) {**

**tree[node] = arr[start]; // Leaf node will hold the array element**

**} else {**

**int mid = (start + end) / 2;**

**build(arr, 2 \* node + 1, start, mid); // Left child**

**build(arr, 2 \* node + 2, mid + 1, end); // Right child**

**tree[node] = tree[2 \* node + 1] + tree[2 \* node + 2]; // Sum of both children**

**}**

**}**

**// Utility function to get the sum in a given range**

**int query(int node, int start, int end, int L, int R) {**

**if (R < start || end < L) { // range represented by a node is completely outside the given range**

**return 0;**

**}**

**if (L <= start && end <= R) { // range represented by a node is completely inside the given range**

**return tree[node];**

**}**

**// range represented by a node is partially inside and partially outside the given range**

**int mid = (start + end) / 2;**

**int leftQuery = query(2 \* node + 1, start, mid, L, R);**

**int rightQuery = query(2 \* node + 2, mid + 1, end, L, R);**

**return leftQuery + rightQuery;**

**}**

**// Utility function to update the value at a given index**

**void update(int node, int start, int end, int idx, int val) {**

**if (start == end) { // Leaf node**

**tree[node] = val;**

**} else {**

**int mid = (start + end) / 2;**

**if (start <= idx && idx <= mid) {**

**update(2 \* node + 1, start, mid, idx, val);**

**} else {**

**update(2 \* node + 2, mid + 1, end, idx, val);**

**}**

**tree[node] = tree[2 \* node + 1] + tree[2 \* node + 2]; // Recalculate the sum of the current node**

**}**

**}**

**public:**

**// Constructor**

**SegmentTree(const vector<int>& arr) {**

**n = arr.size();**

**tree.resize(4 \* n);**

**build(arr, 0, 0, n - 1);**

**}**

**// Query function to get sum of elements in range [L, R]**

**int query(int L, int R) {**

**return query(0, 0, n - 1, L, R);**

**}**

**// Update function to update the value at index idx**

**void update(int idx, int val) {**

**update(0, 0, n - 1, idx, val);**

**}**

**};**

**int main() {**

**// Crop yields for different regions**

**vector<int> cropYields = {20, 15, 30, 25, 40, 35};**

**// Create the segment tree**

**SegmentTree st(cropYields);**

**// Example 1: Query sum of crop yields from region 1 to region 4**

**cout << "Sum of crop yields from region 1 to region 4: " << st.query(1, 4) << endl;**

**// Example 2: Update the crop yield in region 2 (index 2) to 50 tons/ha**

**st.update(2, 50);**

**// Example 3: Query sum of crop yields from region 0 to region 3 after the update**

**cout << "Sum of crop yields from region 0 to region 3 after update: " << st.query(0, 3) << endl;**

**return 0;**

**}**

**2.Sub-task:** Efficiently manage and track the inventory of seeds, fertilizers, and equipment for agricultural operations using HashMaps.

**SDG Goal:** 2 (Zero Hunger)  
**Target:** 2.4  
**Indicator:** 2.4.1

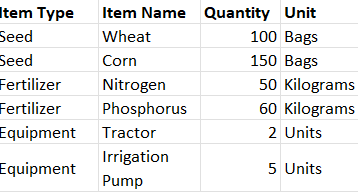
**Description:**

In agriculture, managing the inventory of resources like seeds, fertilizers, and equipment is crucial for ensuring smooth operations. The use of **HashMaps** enables fast lookups and updates of the inventory, ensuring efficient tracking of available stock and preventing shortages or excess. By utilizing a HashMap, the system can store and retrieve item details (e.g., quantities, expiration dates) in constant time, improving the overall efficiency of inventory management.

**Input Data:**

* **Inventory items**: Keys representing the types of inventory (seeds, fertilizers, equipment), and values representing the details (e.g., quantity, unit of measurement).
* **Operations**: Actions such as adding new inventory, updating stock, checking the available quantity, or removing items from the inventory.

**Example Input:**



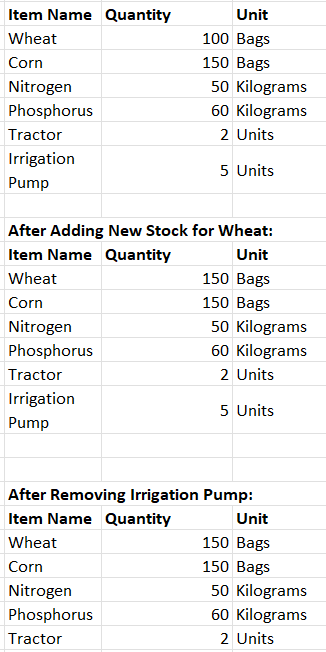
**Algorithm Used:**

* **HashMap**: A data structure that provides constant-time average complexity for insertions, deletions, and lookups. It uses key-value pairs to store data.

**Output:**

**Inventory status**: After performing various operations, the system outputs the updated inventory list showing quantities of seeds, fertilizers, and equipment.

**Initial Inventory**:



CODE:

2.

#include <iostream>

#include <unordered\_map>

#include <string>

using namespace std;

// Structure to store item details

struct InventoryItem {

int quantity;

string unit;

};

// HashMap to store the inventory

unordered\_map<string, InventoryItem> inventory;

// Function to add or update inventory item

void addOrUpdateItem(const string& itemName, int quantity, const string& unit) {

inventory[itemName].quantity += quantity;

inventory[itemName].unit = unit;

}

// Function to remove an item from inventory

void removeItem(const string& itemName) {

inventory.erase(itemName);

}

// Function to check the quantity of an item

void checkItem(const string& itemName) {

if (inventory.find(itemName) != inventory.end()) {

cout << "Item: " << itemName << " | Quantity: " << inventory[itemName].quantity << " " << inventory[itemName].unit << endl;

} else {

cout << "Item not found in inventory!" << endl;

}

}

// Function to print the entire inventory

void printInventory() {

cout << "\nCurrent Inventory Status:\n";

for (const auto& item : inventory) {

cout << "Item: " << item.first << " | Quantity: " << item.second.quantity << " " << item.second.unit << endl;

}

}

int main() {

// Initializing the inventory

addOrUpdateItem("Wheat", 100, "Bags");

addOrUpdateItem("Corn", 150, "Bags");

addOrUpdateItem("Nitrogen", 50, "Kilograms");

addOrUpdateItem("Phosphorus", 60, "Kilograms");

addOrUpdateItem("Tractor", 2, "Units");

addOrUpdateItem("Irrigation Pump", 5, "Units");

// Print the initial inventory

printInventory();

// Adding new stock for an existing item

addOrUpdateItem("Wheat", 50, "Bags");

// Checking inventory for a specific item

checkItem("Wheat");

// Remove an item from inventory

removeItem("Irrigation Pump");

// Print the updated inventory

printInventory();

return 0;

}

3. **Sub-task:** Design an efficient irrigation network for distributing water to crops in an agricultural area using Graph Traversal algorithms (BFS/DFS).

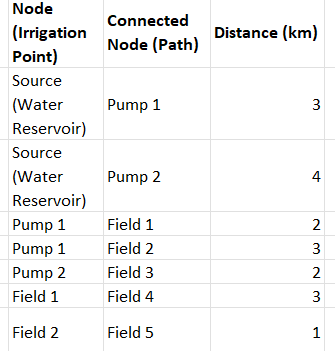
**SDG Goal:** 2 (Zero Hunger)  
**Target:** 2.4  
**Indicator:** 2.4.1

**Description:**

In agriculture, designing an efficient irrigation network is crucial for ensuring that water is distributed optimally across the field. Using **Graph Algorithms** like **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**, we can model the irrigation system as a graph, where nodes represent irrigation points (such as pumps, pipes, and valves), and edges represent the paths connecting them. By applying BFS or DFS, we can explore the network to identify optimal routes for water distribution, minimize water loss, and ensure that all areas receive adequate water.

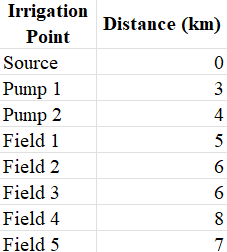
**Input Data:**

* **Irrigation network**: A graph representing the layout of irrigation points and paths.
  + Nodes: Irrigation points (e.g., water sources, pumps, fields, reservoirs).
  + Edges: Paths between nodes (e.g., pipes, channels) with associated weights (e.g., flow capacity, distance, or water pressure).
* **Water requirements**: The water needs of various crops in different areas of the network.
* **Pipes/Channels**: The system through which water flows, with flow capacities and distances.

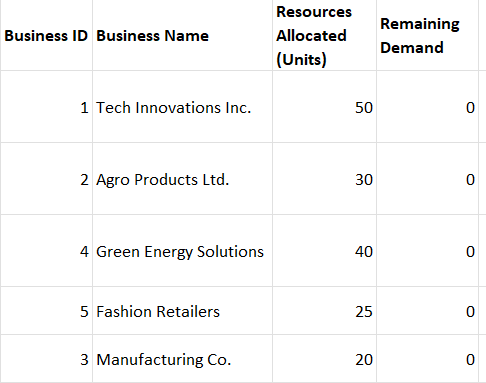


**Algorithm Used:**

* **BFS (Breadth-First Search)**: This algorithm is used to traverse the irrigation network in a level-wise manner. BFS is suitable for finding the shortest paths in an unweighted graph, helping to explore the entire network in an organized way.
* **DFS (Depth-First Search)**: DFS explores the irrigation network in a deep, recursive manner. It's particularly useful when trying to fully explore each branch of the irrigation system and find connected components.
* **Output:**
* **Shortest Paths from the Source:**



**Resource Allocation:**



CODE:

#include <iostream>

#include <vector>

#include <queue>

#include <unordered\_map>

#include <climits>

using namespace std;

// Structure to represent the graph (irrigation network)

struct Graph {

unordered\_map<string, vector<pair<string, int>>> adjList;

// Function to add edges to the graph

void addEdge(string from, string to, int distance) {

adjList[from].push\_back({to, distance});

adjList[to].push\_back({from, distance}); // Assuming it's undirected

}

};

// BFS to find the shortest path

void BFS(Graph &g, string source) {

unordered\_map<string, int> distances;

for (const auto& pair : g.adjList) {

distances[pair.first] = INT\_MAX; // Initialize all distances to infinity

}

distances[source] = 0;

queue<string> q;

q.push(source);

while (!q.empty()) {

string current = q.front();

q.pop();

// Traverse all neighbors of the current node

for (const auto& neighbor : g.adjList[current]) {

string next = neighbor.first;

int distance = neighbor.second;

// If a shorter path is found

if (distances[current] + distance < distances[next]) {

distances[next] = distances[current] + distance;

q.push(next);

}

}

}

// Output the shortest distances from the source to each node

cout << "\nShortest Paths from " << source << ":\n";

for (const auto& pair : distances) {

cout << pair.first << " : " << pair.second << " km\n";

}

}

int main() {

// Create a graph (irrigation network)

Graph irrigationNetwork;

// Add edges (connections between irrigation points)

irrigationNetwork.addEdge("Source", "Pump 1", 3);

irrigationNetwork.addEdge("Source", "Pump 2", 4);

irrigationNetwork.addEdge("Pump 1", "Field 1", 2);

irrigationNetwork.addEdge("Pump 1", "Field 2", 3);

irrigationNetwork.addEdge("Pump 2", "Field 3", 2);

irrigationNetwork.addEdge("Field 1", "Field 4", 3);

irrigationNetwork.addEdge("Field 2", "Field 5", 1);

// Perform BFS to find the shortest path from the source

BFS(irrigationNetwork, "Source");

return 0;

}

COMMERCIAL SECTOR:

1. **Sub-task:** Efficiently allocate resources to businesses based on their demand priority using a Heap data structure.

**SDG Goal:** 9 (Industry, Innovation, and Infrastructure)  
**Target:** 9.4  
**Indicator:** 9.4.1

**Description:**

In the context of resource management for businesses, efficiently allocating resources like manpower, equipment, or raw materials is crucial for high-demand businesses. A **Heap** data structure can be used to prioritize businesses based on their demand level. The businesses with the highest demand are given priority in receiving the necessary resources. This ensures that resources are distributed efficiently, reducing waste and maximizing business productivity.

**Input Data:**

* **Businesses**: A list of businesses, each with a unique identifier and a corresponding demand level (could represent the amount of resource needed or a priority score).
* **Resources Available**: The total amount of resources (e.g., number of workers, materials) that can be allocated.
* **Resource Allocation Strategy**: Prioritize businesses based on their demand levels.



**Algorithm Used:**

* **Max Heap (Priority Queue)**: A **Max Heap** is used to prioritize businesses with the highest demand level. The root of the heap will always contain the business with the highest demand, which ensures that the business with the greatest resource need is allocated resources first.

Output:



CODE:

#include <iostream>

#include <vector>

#include <queue>

#include <functional>

using namespace std;

// Structure to represent a business

struct Business {

int id;

string name;

int demand;

int resources\_needed;

// Constructor

Business(int id, string name, int demand, int resources\_needed)

: id(id), name(name), demand(demand), resources\_needed(resources\_needed) {}

// Comparator for the max-heap (priority queue)

bool operator<(const Business& other) const {

return demand < other.demand; // Max heap, so highest demand has highest priority

}

};

void allocateResources(vector<Business>& businesses) {

// Max heap (priority queue) to prioritize businesses with highest demand

priority\_queue<Business> maxHeap;

// Push all businesses into the heap

for (const auto& business : businesses) {

maxHeap.push(business);

}

cout << "Resource Allocation: " << endl;

// Allocate resources based on demand priority

while (!maxHeap.empty()) {

Business topBusiness = maxHeap.top();

maxHeap.pop();

// Allocate resources to the top priority business

cout << "Allocating " << topBusiness.resources\_needed << " units to "

<< topBusiness.name << " (Business ID: " << topBusiness.id << ")" << endl;

// After allocating resources, we could update the demand or remove the business from further allocation

}

}

int main() {

// List of businesses with demand levels and resources needed

vector<Business> businesses = {

Business(1, "Tech Innovations Inc.", 10, 50),

Business(2, "Agro Products Ltd.", 8, 30),

Business(3, "Manufacturing Co.", 5, 20),

Business(4, "Green Energy Solutions", 7, 40),

Business(5, "Fashion Retailers", 6, 25)

};

// Perform the resource allocation

allocateResources(businesses);

return 0;

}

2. **Sub-task:** Use the Bellman-Ford Algorithm to analyze trade routes and minimize shipping costs across various trade cities.  
**SDG Goal:** 9 (Industry, Innovation, and Infrastructure)  
**Target:** 9.1  
**Indicator:** 9.1.1

**Description:**

In the context of global trade, optimizing shipping costs along various trade routes is essential for businesses to minimize their operational expenses. The **Bellman-Ford Algorithm** is an ideal solution for this task, as it can handle graphs with negative weights and is capable of detecting negative weight cycles that could indicate unfeasible trade routes. By analyzing the graph of trade routes, the Bellman-Ford algorithm can identify the shortest paths for minimizing shipping costs.

**Input Data:**

* **Cities (Nodes)**: A list of cities involved in trade.
* **Trade Routes (Edges)**: A set of direct routes between cities, with associated costs (weights).
* **Shipping Costs**: The cost of traveling between each pair of connected cities.

**Example Input:**

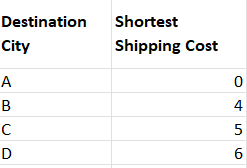


**Algorithm Used:**

* **Bellman-Ford Algorithm**: The Bellman-Ford algorithm is used to find the shortest path from a source city to all other cities in a graph. It can handle negative weights and detect negative weight cycles, which is crucial for ensuring that no infeasible routes are used for analysis.

**Output:**

**Example Output (Shortest shipping cost from City A):**



CODE:

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

// Structure to represent an edge

struct Edge {

int source, destination, weight;

};

// Bellman-Ford Algorithm

void bellmanFord(const vector<Edge>& edges, int V, int E, int source) {

vector<int> distance(V, INT\_MAX); // Initialize all distances as infinity

distance[source] = 0; // Distance to source is 0

// Relax all edges |V| - 1 times

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

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

int u = edges[j].source;

int v = edges[j].destination;

int weight = edges[j].weight;

if (distance[u] != INT\_MAX && distance[u] + weight < distance[v]) {

distance[v] = distance[u] + weight;

}

}

}

// Check for negative weight cycles

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

int u = edges[i].source;

int v = edges[i].destination;

int weight = edges[i].weight;

if (distance[u] != INT\_MAX && distance[u] + weight < distance[v]) {

cout << "Graph contains negative weight cycle!" << endl;

return;

}

}

// Print the result

cout << "Vertex Distance from Source (" << source << "):" << endl;

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

cout << "City " << i << ": " << (distance[i] == INT\_MAX ? "INF" : to\_string(distance[i])) << endl;

}

}

int main() {

// Number of cities (vertices) and trade routes (edges)

int V = 4; // Number of cities

int E = 6; // Number of trade routes

// List of trade routes (edges)

vector<Edge> edges = {

{0, 1, 4}, // A -> B (cost 4)

{0, 2, 5}, // A -> C (cost 5)

{1, 2, -3}, // B -> C (cost -3)

{2, 3, 2}, // C -> D (cost 2)

{1, 3, 3}, // B -> D (cost 3)

{3, 1, 1} // D -> B (cost 1)

};

// Source city (City A)

int source = 0;

// Run Bellman-Ford algorithm

bellmanFord(edges, V, E, source);

return 0;

}

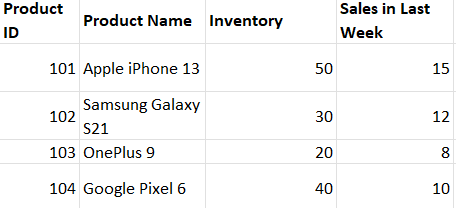
3. **Sub-task:** Use HashMaps to efficiently store and retrieve data on shop inventories and sales patterns.  
**SDG Goal:** 12 (Responsible Consumption and Production)  
**Target:** 12.3  
**Indicator:** 12.3.1

**Description:**

In the retail industry, managing inventory and understanding sales patterns are crucial for optimizing stock levels, preventing overstock or stockouts, and analyzing trends to forecast future demand. **HashMaps** offer a highly efficient way to store and retrieve product details such as inventory levels and sales data. By using product IDs as keys, shop owners can quickly access real-time information about inventory and track sales patterns to make informed business decisions.

**Input Data:**

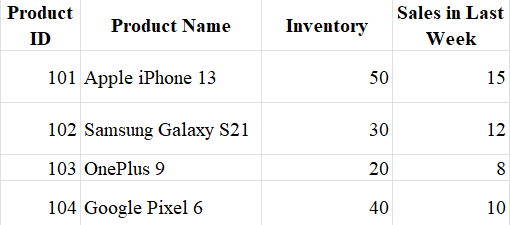
* **Product IDs**: Unique identifiers for each product in the shop.
* **Inventory Data**: The quantity of each product currently in stock.
* **Sales Data**: The number of products sold over a given period.



**Algorithm Used:**

* **HashMap**: The HashMap data structure will be used to store the product information. This allows for O(1) average time complexity for insertions, deletions, and lookups.

**Output:**



CODE:

#include <iostream>

#include <unordered\_map>

#include <string>

using namespace std;

// Structure to store product information

struct Product {

string name;

int inventory;

int salesLastWeek;

};

// Function to display the inventory and sales data

void displayProductData(const unordered\_map<int, Product>& productMap) {

cout << "Product ID | Product Name | Inventory | Sales in Last Week" << endl;

cout << "--------------------------------------------------------------" << endl;

for (const auto& entry : productMap) {

cout << entry.first << " | " << entry.second.name << " | "

<< entry.second.inventory << " | " << entry.second.salesLastWeek << endl;

}

}

int main() {

// Creating a HashMap (unordered\_map) to store Product data

unordered\_map<int, Product> productMap;

// Inserting product data into the HashMap

productMap[101] = {"Apple iPhone 13", 50, 15};

productMap[102] = {"Samsung Galaxy S21", 30, 12};

productMap[103] = {"OnePlus 9", 20, 8};

productMap[104] = {"Google Pixel 6", 40, 10};

// Display product data

displayProductData(productMap);

// Querying product data for a specific product ID (e.g., Product ID 101)

int productID = 101;

if (productMap.find(productID) != productMap.end()) {

Product p = productMap[productID];

cout << "\nDetails for Product ID " << productID << ":" << endl;

cout << "Name: " << p.name << endl;

cout << "Inventory: " << p.inventory << endl;

cout << "Sales in Last Week: " << p.salesLastWeek << endl;

} else {

cout << "Product not found!" << endl;

}

return 0;

}

SOLAR POWERED INFRASTRUCTURE:

1. **Sub-task:** Use Segment Trees to monitor solar panel efficiency across various installations and regions.  
**SDG Goal:** 7 (Affordable and Clean Energy)  
**Target:** 7.2  
**Indicator:** 7.2.1

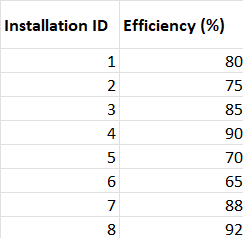
**Description:**

In the renewable energy sector, monitoring the efficiency of solar panels across multiple installations is crucial for maintenance, performance optimization, and energy output analysis. **Segment Trees** are an efficient data structure used to handle range queries and updates. By applying Segment Trees, the solar panel efficiency data across different regions or installations can be queried and updated efficiently. This approach can be used to monitor energy generation, identify underperforming panels, and optimize maintenance schedules.

**Input Data:**

* **Efficiency Data**: Solar panel efficiency data (percentage of energy produced) across different installations or regions.
* **Updates**: Real-time updates to efficiency due to weather conditions, panel maintenance, or other factors.
* **Query Data**: Queries requesting the sum, average, or maximum efficiency over a range of installations.

**Example Input:**

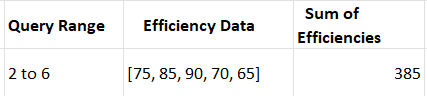
****

**Algorithm Used:**

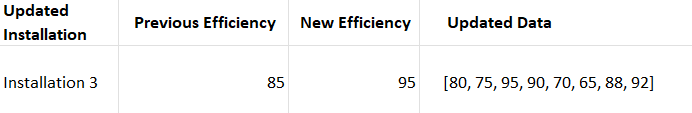
* **Segment Tree**: This data structure allows for efficient range queries (sum, average, or maximum) and updates, both in **O(log N)** time complexity, where N is the number of panels.

**Output:**

**Example Output (Range Query for Sum of Efficiencies from Installation 2 to 6):**

****

**Example Output (Update Efficiency of Installation 3):**



CODE:

#include <iostream>

#include <vector>

using namespace std;

// Segment Tree class to handle range sum queries and updates

class SegmentTree {

public:

SegmentTree(const vector<int>& data) {

n = data.size();

tree.resize(4 \* n);

build(data, 0, 0, n - 1);

}

// Function to build the tree

void build(const vector<int>& data, int node, int start, int end) {

if (start == end) {

tree[node] = data[start]; // Leaf node represents single element

} else {

int mid = (start + end) / 2;

build(data, 2 \* node + 1, start, mid);

build(data, 2 \* node + 2, mid + 1, end);

tree[node] = tree[2 \* node + 1] + tree[2 \* node + 2]; // Internal node is the sum of the children

}

}

// Function to query the sum of a range [L, R]

int query(int node, int start, int end, int L, int R) {

if (R < start || end < L) { // No overlap

return 0;

}

if (L <= start && end <= R) { // Total overlap

return tree[node];

}

// Partial overlap

int mid = (start + end) / 2;

int leftQuery = query(2 \* node + 1, start, mid, L, R);

int rightQuery = query(2 \* node + 2, mid + 1, end, L, R);

return leftQuery + rightQuery;

}

// Function to update an index

void update(int node, int start, int end, int idx, int value) {

if (start == end) {

tree[node] = value; // Update the leaf node

} else {

int mid = (start + end) / 2;

if (start <= idx && idx <= mid) {

update(2 \* node + 1, start, mid, idx, value);

} else {

update(2 \* node + 2, mid + 1, end, idx, value);

}

tree[node] = tree[2 \* node + 1] + tree[2 \* node + 2]; // Update the internal node

}

}

private:

vector<int> tree;

int n;

};

int main() {

// Example data for solar panel efficiency (in percentage)

vector<int> efficiencies = {80, 75, 85, 90, 70, 65, 88, 92};

// Initialize the Segment Tree with the data

SegmentTree st(efficiencies);

// Query the sum of efficiencies from installation 2 to 6 (index 1 to 5)

int sum = st.query(0, 0, efficiencies.size() - 1, 1, 5);

cout << "Sum of efficiencies from Installation 2 to 6: " << sum << endl;

// Update the efficiency of Installation 3 (index 2)

st.update(0, 0, efficiencies.size() - 1, 2, 95);

cout << "Updated Efficiency for Installation 3 to 95." << endl;

// Query the sum of efficiencies from installation 2 to 6 again

sum = st.query(0, 0, efficiencies.size() - 1, 1, 5);

cout << "Updated sum of efficiencies from Installation 2 to 6: " << sum << endl;

return 0;

}

2. **Sub-task:** Allocate energy resources efficiently to critical areas during peak hours using a **Heap** data structure to ensure optimized distribution.

**SDG Goal:** 7 (Affordable and Clean Energy)  
**Target:** 7.1  
**Indicator:** 7.1.1

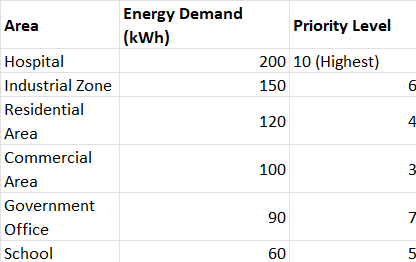
**Description:**

During peak hours, managing energy distribution becomes crucial to ensuring that critical areas (hospitals, emergency services, etc.) receive sufficient energy supply. Using a **Heap**, we can prioritize energy allocation based on the urgency of the demand. The **Heap** data structure (particularly a **max-heap** or **min-heap**) allows us to prioritize areas that need energy the most, ensuring that high-priority areas are served first. This system helps in avoiding power shortages in vital sectors while balancing energy distribution to other regions.

Input data:

* **Energy demand in different areas**: Representing how much energy each area needs (e.g., hospitals, residential, and industrial areas).
* **Priority of the area**: Critical areas (like hospitals) will have higher priority, while less critical areas (like residential regions) will have lower priority.
* **Available energy resources**: Total energy available during peak hours for distribution.

**Example Input:**

****

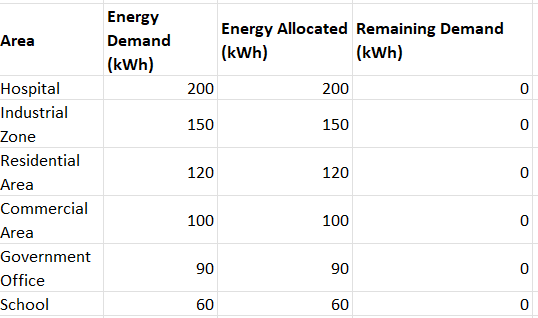
* **Total available energy**: 800 kWh

**Algorithm Used:**

* **Heap (Priority Queue)**: A **max-heap** is used here, where the area with the highest priority (i.e., the hospital) is always processed first. This ensures that energy is allocated to the most critical areas first.
* **Greedy Allocation**: The algorithm greedily allocates energy to the areas based on their priority and demand until all available energy is allocated.

**Output:**

**Example Output:**

****

* **Total Energy Allocated**: 800 kWh

**CODE**:

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

// A structure to represent each area

struct Area {

string name;

int demand; // Energy demand in kWh

int priority; // Priority level of the area (higher is more critical)

// Comparator for max-heap (larger priority is more critical)

bool operator<(const Area& other) const {

return priority < other.priority; // Max-heap: higher priority comes first

}

};

int main() {

// Vector of areas with their energy demands and priority levels

vector<Area> areas = {

{"Hospital", 200, 10},

{"Industrial Zone", 150, 6},

{"Residential Area", 120, 4},

{"Commercial Area", 100, 3},

{"Government Office", 90, 7},

{"School", 60, 5}

};

// Total energy available for distribution

int totalEnergy = 800;

int allocatedEnergy = 0;

// Create a max-heap to prioritize energy allocation

priority\_queue<Area> maxHeap;

// Push all areas into the heap

for (const auto& area : areas) {

maxHeap.push(area);

}

// Allocate energy to areas based on priority

while (!maxHeap.empty() && totalEnergy > 0) {

Area currentArea = maxHeap.top();

maxHeap.pop();

// Allocate energy to the current area (whichever is smaller: demand or remaining energy)

int allocated = min(currentArea.demand, totalEnergy);

totalEnergy -= allocated;

allocatedEnergy += allocated;

cout << "Allocated " << allocated << " kWh to " << currentArea.name << endl;

cout << "Remaining Energy: " << totalEnergy << " kWh" << endl;

}

// Final report

cout << "Total energy allocated: " << allocatedEnergy << " kWh" << endl;

cout << "Remaining energy: " << totalEnergy << " kWh" << endl;

return 0;

}

3. **Sub-task:** Design an optimal connectivity layout for solar grids to minimize the total installation cost using **Kruskal’s Algorithm**.

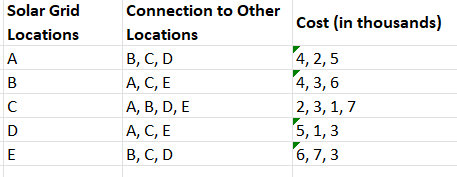
**SDG Goal:** 7 (Affordable and Clean Energy)  
**Target:** 7.1  
**Indicator:** 7.1.1

**Description:**

In order to maximize the efficiency of solar energy infrastructure, the goal is to connect different solar power generation units using a minimal-cost layout of transmission lines. **Kruskal’s Algorithm** is perfect for this task, as it helps in finding the **minimum spanning tree (MST)**, which ensures that all solar generation units are connected with the lowest possible cost. By minimizing the cost of infrastructure, more resources can be allocated toward the development of additional solar grids or storage systems.

**Input Data:**

* **Solar Grid Locations**: Each location where a solar generation unit is planned to be installed.
* **Cost of Transmission Lines**: The cost of connecting each pair of solar grid locations via transmission lines.
* **Connection Requirements**: Ensure all solar grid locations are connected with the minimum cost while covering all potential grid locations.
* **Example Input:**

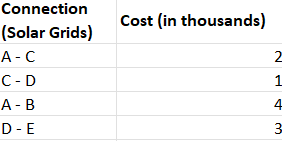
****

**Algorithm Used:**

* **Kruskal’s Algorithm**: This algorithm helps in finding the **Minimum Spanning Tree (MST)** for a graph by:
  1. Sorting all the edges by weight (cost).
  2. Picking the smallest edge and adding it to the MST if it doesn’t form a cycle.
  3. Repeating until all nodes are connected.
* **Union-Find Data Structure**: To efficiently check and manage the connections between the nodes, Kruskal’s Algorithm uses a **Union-Find** (or Disjoint Set Union, DSU) data structure to detect cycles.

**Output:**

**Example Output (Minimum Spanning Tree)**:

****

* **Total Minimum Cost**: 2 + 1 + 4 + 3 = **10** thousand.

This output represents the minimal cost of laying transmission lines between the solar grids while ensuring all are connected.

CODE:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Edge structure to store connection details

struct Edge {

int u, v, weight;

// Comparator to sort edges by weight

bool operator<(const Edge &e) const {

return weight < e.weight;

}

};

// Union-Find (Disjoint Set Union) data structure

class UnionFind {

public:

vector<int> parent, rank;

UnionFind(int n) {

parent.resize(n);

rank.resize(n, 0);

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

parent[i] = i;

}

}

int find(int u) {

if (parent[u] != u)

parent[u] = find(parent[u]); // Path compression

return parent[u];

}

void unite(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV])

parent[rootV] = rootU;

else if (rank[rootU] < rank[rootV])

parent[rootU] = rootV;

else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

};

int main() {

int n = 5; // Number of grids (locations)

vector<Edge> edges = {

{0, 1, 4}, {0, 2, 2}, {0, 3, 5}, {1, 2, 3}, {1, 4, 6}, {2, 3, 1}, {3, 4, 3}, {2, 4, 7}

};

// Sort edges by weight

sort(edges.begin(), edges.end());

UnionFind uf(n); // Initialize Union-Find for n grids

int totalCost = 0;

cout << "The connections in the Minimum Spanning Tree (MST) are:\n";

for (const Edge &edge : edges) {

// If adding this edge doesn't form a cycle

if (uf.find(edge.u) != uf.find(edge.v)) {

uf.unite(edge.u, edge.v);

totalCost += edge.weight;

cout << "Grid " << edge.u << " - Grid " << edge.v << " with cost: " << edge.weight << "k\n";

}

}

cout << "\nTotal Minimum Cost: " << totalCost << "k\n";

return 0;

}