Minimum Cost to connect all houses in a city :-

Minimum cost to connect all houses in a city

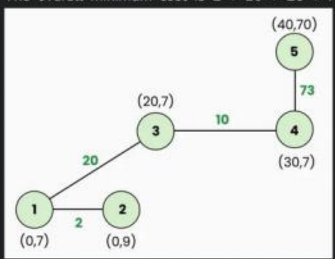
Difficulty: **Medium** Accuracy: **64.58%** Submissions: **14K+** Points: **4**

Given a 2D array **houses[][]**, consisting of **n** 2D coordinates **{x, y}** where each coordinate represents the **location of each house**, the task is to find the **minimum cost to connect** all the houses of the city.

The **cost of connecting** two houses is the **Manhattan Distance** between the two points (x_i, y_i) and (x_j, y_j) i.e., $|x_i - x_j| + |y_i - y_j|$, where $|p|$ denotes the absolute value of p.

Examples :

Input: n = 5 houses[][] = [[0, 7], [0, 9], [20, 7], [30, 7], [40, 70]]
Output: 105
Explanation:
Connect house 1 (0, 7) and house 2 (0, 9) with cost = 2
Connect house 1 (0, 7) and house 3 (20, 7) with cost = 20
Connect house 3 (20, 7) with house 4 (30, 7) with cost = 10
At last, connect house 4 (30, 7) with house 5 (40, 70) with cost 73.
All the houses are connected now.
The overall minimum cost is 2 + 10 + 20 + 73 = 105.



Input: n = 4 houses[][] = [[0, 0], [1, 1], [1, 3], [3, 0]]
Output: 7
Explanation:
Connect house 1 (0, 0) with house 2 (1, 1) with cost = 2
Connect house 2 (1, 1) with house 3 (1, 3) with cost = 2
Connect house 1 (0, 0) with house 4 (3, 0) with cost = 3
The overall minimum cost is 3 + 2 + 2 = 7.

Constraint:
$1 \le n \le 10^3$
$0 \le houses[i][j] \le 10^3$

[Approach 1] Using Prim's Algorithm – Time O(n^2 *log(n)) & Space O(n^2)

We can think of each node as a node in a graph, & the Manhattan Distance between any two houses as the weight of the edge connecting those two nodes. With this interpretation, the problem of connecting all houses with the minimum total connecting all houses with the minimum total cost becomes equivalent to finding a <u>Minimum Spanning Tree (MST)</u> of a Complete Graph.

<u>Step by step Implementation :-</u>

- Start with any house (we start with house 0).
- Push all distances from this house to other houses into a min-heap (priority queue).
- At every step: Pick the house with the smallest connection cost that hasn't been visited.
- Add that cost to the total cost & mark the house as visited.
- Push distances from this new house to all unvisited houses into the heap.
- Repeat until all houses are visited & return the total cost.

```cpp
class Solution {
  public:
  // Calculates Manhattan Distance between two houses
  int manhattanDistance(vector<int> &a, vector<int> &b){
      return abs(a[0]-b[0])+abs(a[1]-b[1]);
  }

    // Returns the minimum cost to connect
    // all houses using Prim's algorithm
    int minCost(vector<vector<int>>& houses) {
        // code here
        int n= houses.size();

        // Min-heap to store (cost, house_index)
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> minHeap;

        // Marks whether a house is already connected
        vector<bool> visited(n, false);

        // Start with the first house (index 0)
        minHeap.push({0,0});

        int totalCost=0;

        while(!minHeap.empty()){
            pair<int, int> p=minHeap.top(); minHeap.pop();
            int cost=p.first;
            int u=p.second;

            // Skip if already connected
            if(visited[u])continue;

            // Mark the house as connected and add the cost
            visited[u]=true;
            totalCost+=cost;

            // Calculate distance to all unvisited houses and add to heap
            for(int v=0;v<n;v++){
                if(!visited[v]){
                    int dist=manhattanDistance(houses[u], houses[v]);
                    minHeap.push({dist, v});
                }
            }
        }
        return totalCost;
    }
};
```

[Approach 2] Using Kruskal's Algorithm - Time O(n^2*log(n)) and Space O(n^2)

To solve the problem, we model it as a **weighted graph**, where each house is a **node**, and the **edge weight** between any two houses is the **Manhattan distance** (i.e., the cost to connect them).

> We generate all possible edges between houses and store their corresponding weights. Then, we use **Kruskal's algorithm** to find the **Minimum Spanning Tree (MST)** of this graph. To efficiently detect and avoid cycles while building the MST, we use a **Disjoint Set Union (DSU)** data structure with **path compression and union by rank**.

```cpp
class DSU{
    vector<int> parent, rank;
    public:
    DSU(int n){
        parent.resize(n, -1);
        rank.resize(n, 1);
    }
    int find(int i){
        if(parent[i]==-1)return i;
        return parent[i]=find(parent[i]);
    }
    void unite(int x, int y){
         int s1 = find(x);
    int s2 = find(y);

    if(s1 != s2) {
        if(rank[s1] < rank[s2]) swap(s1, s2);
        parent[s2] = s1;
        if(rank[s1] == rank[s2]) rank[s1]++;
    }
    }
};

class Graph{
    vector<vector<int>> edgeList;
    int V;

    public:
    Graph(int V){this->V=V;}

    // Function to add edge in a graph
    void addEdge(int x, int y, int w){
        edgeList.push_back({w, x, y});
    }

    int kruskalMST(){
        // sort all edges
        sort(edgeList.begin(), edgeList.end());
        // Initialize the DSU
        DSU s(V);

        //stores the final  answer
        int ans=0;
        int count=0; // no of edges in MST

        for(auto edge: edgeList){
            int w=edge[0];
            int x=edge[1];
            int y=edge[2];

            // Take this edge in MST if it does not form a cycle
            if(s.find(x)!=s.find(y)){
                s.unite(x,y);
                ans+=w;
                count++;
            }
            if(count==V-1)break;
        }
    return ans;
    }
};

class Solution {
  public:
    int minCost(vector<vector<int>>& houses) {
        // code here
        int n=houses.size();

        // Create graph with n nodes
        Graph g(n);

        // Add all possible edges
        for(int i=0;i<n;i++){
            for(int j=i+1;j<n;j++){
                int cost = abs(houses[i][0]-houses[j][0])+abs(houses[i][1]-houses[j][1]);
                g.addEdge(i, j, cost);
            }
        }
        return g.kruskalMST();
    }
};
```