# Greedy techniques

## Prim’s algorithm

Q2.
N - 1
Q5.
Click submit
Q7.

```
#include <bits/stdc++.h>
using namespace std;

const int N = 2e5 + 10;
vector<bool> vis(N, false);
vector<pair<int,int>> adj[N];

long long prims(int source, int n) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, source}); // {weight, vertex}
    long long mstWeight = 0;

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

        // Skip if the vertex is already visited
        if (vis[u]) continue;
        vis[u] = true;
        mstWeight += w;

        // Traverse all adjacent vertices of u
        for (auto edge : adj[u]) {
            int v = edge.first;
            int weight = edge.second;
            if (!vis[v]) {
                pq.push({weight, v});
            }
        }
    }

    return mstWeight;
}

int main() {
    int n, m;
    cin >> n >> m;
```

```
    for(int i = 0; i < m; i++) {
        int x, y, w;
        cin >> x >> y >> w;
        adj[x].push_back({y, w});
        adj[y].push_back({x, w});
    }

    long long mstWeight = prims(1, n);
    cout << mstWeight << endl;

    return 0;
}
```

Q8.(**Change the language to python)**

```
import heapq

class UnionFind:
    def __init__(self, size):
        self.parent = [i for i in range(size)]

    def find(self, u):
        if u != self.parent[u]:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        u = self.find(u)
        v = self.find(v)
        if u != v:
            self.parent[u] = v

def main():
    t = int(input())
    for _ in range(t):
        graph = {}
        edgeList = []
        n, m = map(int, input().split())
        for i in range(n):
            graph[i] = []
        for _ in range(m):
            u, v, w = map(int, input().split())
            graph[u].append((v, w))
            graph[v].append((u, w))
            edgeList.append((u, v, w))
        edgeList.sort(key=lambda x: x[2])

        uf = UnionFind(n)
```

```
        mst = 0
        cnt = 0

        for edge in edgeList:
            u, v, w = edge
            if uf.find(u) != uf.find(v):
                uf.union(u, v)
                mst += w
                cnt += 1
                if cnt == n - 1:  # Check if we have added n-1 edges
                    break

        if cnt != n - 1:
            print("NO")
            continue

        dist = [float('inf')] * n
        edgeChosen = [0] * n
        dist[0] = 0
        pq = [(0, 0)]

        while pq:
            d, u = heapq.heappop(pq)
            if dist[u] < d:
                continue
            for v, w in graph[u]:
                if dist[v] > w + d:
                    dist[v] = w + d
                    heapq.heappush(pq, (w + d, v))
                    edgeChosen[v] = w
                elif dist[v] == w + d and w < edgeChosen[v]:
                    edgeChosen[v] = w

        mdt = sum(edgeChosen)
        if mdt == mst:
            print("YES")
        else:
            print("NO")

if __name__ == "__main__":
    main()
```

## Kruskal's Algorithm

Q3.
Click submit

**Change the language to python**

Q5.

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1

def kruskal_mst(n, edges):
    edges.sort(key=lambda edge: edge[2]) # Sort edges by weight
    uf = UnionFind(n)
    mst_weight = 0
    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst_weight += weight
    return mst_weight

def main():
    import sys
    input = sys.stdin.read
    data = input().split()
    index = 0

    n = int(data[index])
    m = int(data[index + 1])
    index += 2

    edges = []
    for _ in range(m):
        u = int(data[index]) - 1 # Convert to 0-based index
        v = int(data[index + 1]) - 1 # Convert to 0-based index
```

```
        weight = int(data[index + 2])
        edges.append((u, v, weight))
        index += 3

    result = kruskal_mst(n, edges)
    print(result)

if __name__ == "__main__":
    main()
```

Q6.

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1

def calculate_distance(row1, row2, m):
    return max(abs(ord(row1[i]) - ord(row2[i])) for i in range(m))

def find_least_largest_cost(n, m, grid):
```

```
    edges = []
    for i in range(n):
        for j in range(i + 1, n):
            distance = calculate_distance(grid[i], grid[j], m)
            edges.append((distance, i, j))

    edges.sort()
    uf = UnionFind(n)
    max_cost = 0

    for weight, u, v in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            max_cost = weight

    return max_cost

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    n = int(data[0])
    m = int(data[1])
    grid = data[2:(2 + n)]

    result = find_least_largest_cost(n, m, grid)
    print(result)

if __name__ == "__main__":
    main()
```

## Huffman Trees and codes

Q2.

Both First and Second
Q6.
Click submit