```
def read_input(file_path):
    with open(file_path, 'r') as f:
        data = f.read().splitlines()
    n, m = map(int, data[0].split())
    edges = [tuple(map(int, line.split())) for line in data[1:]]

    # Коррекция нумерации вершин (если нужно)
    edges = [(u-1, v-1, cost) for u, v, cost in edges]
    return n, edges

n_test, edges_test = read_input("test.txt")
n_data, edges_data = read_input("data.txt")

print(n_test, len(edges_test))
print(edges_test)

def prim(n, edges):

    visited = [False] * n
    mst = []
    mst_cost = 0
    visited[0] = True

    while len(mst) < n - 1:
        min_edge = None
        min_cost = float('inf')

        for u, v, cost in edges:
            if 0 <= u < n and 0 <= v < n:  # Проверка индексов
                if visited[u] and not visited[v] and cost < min_cost:
                    min_edge = (u, v, cost)
                    min_cost = cost
```

```
            elif visited[v] and not visited[u] and cost < min_cost:
                min_edge = (v, u, cost)
                min_cost = cost

        if min_edge:
            mst.append(min_edge)
            mst_cost += min_edge[2]
            visited[min_edge[1]] = True
    return mst_cost, mst

print(prim(n_test, edges_test)[1])

import time

start_time = time.time()
prim_test = prim(n_test, edges_test)
end_time = time.time()

print(f'Стоимость MST графа на тестовом наборе данных с помощью алгоритма Прима: {prim_test[0]}')
print(f'Время работы алгоритма Прима на тестовом наборе данных: {end_time - start_time} секунд')

import time

start_time = time.time()
prim_data = prim(n_data, edges_data)
end_time = time.time()

print(f'Стоимость MST графа на усложненном наборе данных с помощью алгоритма Прима: {prim_data[0]}')
print(f'Время работы алгоритма Прима на усложненном наборе данных: {end_time - start_time} секунд')
```

```
def kruskal_naive(n, edges):
    # Сортируем рёбра по весу
    edges.sort(key=lambda x: x[2])

    # Список для хранения минимального остовного дерева
    mst = []
    mst_cost = 0

    # Изначально каждая вершина принадлежит своей компоненте
    components = {i: {i} for i in range(n)}

    for u, v, cost in edges:
        # Если вершины принадлежат разным компонентам, добавляем ребро
        if components[u] != components[v]:
            mst.append((u, v, cost))
            mst_cost += cost

            # Объединяем компоненты
            comp_u = components[u]
            comp_v = components[v]
            merged = comp_u.union(comp_v)

            for node in merged:
                components[node] = merged

            # Останавливаемся, если в дереве уже n-1 рёбер
            if len(mst) == n - 1:
                break

    return mst_cost, mst

import time
```

```
start_time = time.time()

kruskal_test = kruskal_naive(n_test, edges_test)

end_time = time.time()


print(f'Стоимость MST графа на тестовом наборе данных с помощью алгоритма Крускала (наивный подход): {kruskal_test[0]}')

print(f'Время работы алгоритма Крускала (наивный подход) на тестовом наборе данных: {end_time - start_time} секунд')


import time


start_time = time.time()

kruskal_data = kruskal_naive(n_data, edges_data)

end_time = time.time()


print(f'Стоимость MST графа на усложненном наборе данных с помощью алгоритма Крускала (наивный подход): {kruskal_data[0]}')

print(f'Время работы алгоритма Крускала (наивный подход) на усложненном наборе данных: {end_time - start_time} секунд')


class UnionFind:

    def __init__(self, n):

        self.parent = list(range(n))

        self.rank = [0] * n


    def find(self, node):

        if self.parent[node] != node:

            self.parent[node] = self.find(self.parent[node])

        return self.parent[node]


    def union(self, node1, node2):

        root1 = self.find(node1)

        root2 = self.find(node2)
```

```
        if root1 != root2:
            if self.rank[root1] > self.rank[root2]:
                self.parent[root2] = root1
            elif self.rank[root1] < self.rank[root2]:
                self.parent[root1] = root2
            else:
                self.parent[root2] = root1
                self.rank[root1] += 1


def kruskal(n, edges):
    edges.sort(key=lambda x: x[2])  # Сортировка рёбер по весу
    uf = UnionFind(n)
    mst = []
    mst_cost = 0

    for u, v, cost in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, cost))
            mst_cost += cost

    return mst_cost, mst

import time

start_time = time.time()
kruskal_test = kruskal(n_test, edges_test)
end_time = time.time()

print(f'Стоимость MST графа на тестовом наборе данных с помощью алгоритма Крускала (Union_Find): {kruskal_test[0]}')
```

```
print(f'Время работы алгоритма Крускала (Union_Find) на тестовом наборе данных: {end_time - start_time} секунд')


import time


start_time = time.time()

kruskal_data = kruskal(n_data, edges_data)

end_time = time.time()


print(f'Стоимость MST графа на усложненном наборе данных с помощью алгоритма Крускала (Union_Find): {kruskal_data[0]}')

print(f'Время работы алгоритма Крускала (Union_Find) на усложненном наборе данных: {end_time - start_time} секунд')
```