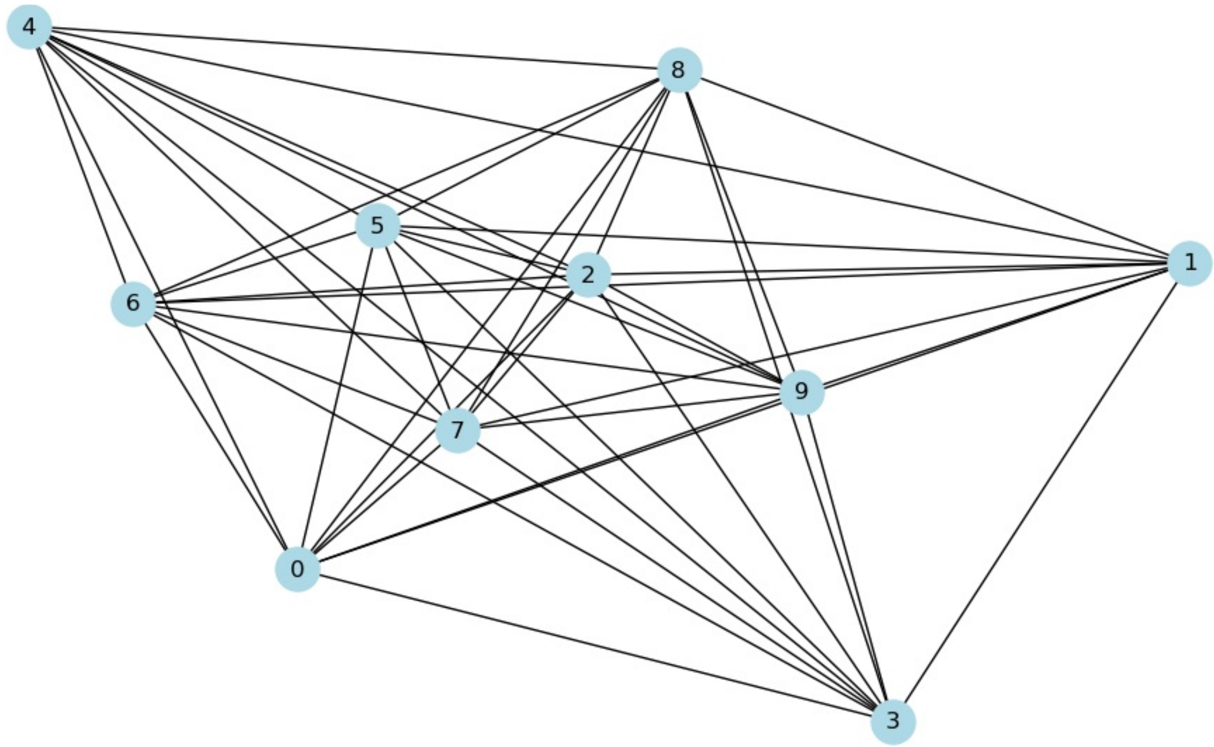
In [15]:

```
import random
import networkx as nx
import matplotlib.pyplot as plt
from itertools import combinations, groupby
import time
from tqdm import tqdm
from networkx.algorithms import tree
```

In [16]:

```
def gnp_random_connected_graph(num_of_nodes: int,
                               completeness: int,
                               directed: bool = False,
                               draw: bool = False):
    """
    Generates a random graph, similarly to an Erdős-Rényi 
    graph, but enforcing that the resulting graph is conneted (in case of undirected grap
hs)
    """


    if directed:
        G = nx.DiGraph()
    else:
        G = nx.Graph()
    edges = combinations(range(num_of_nodes), 2)
    G.add_nodes_from(range(num_of_nodes))

    for _, node_edges in groupby(edges, key = lambda x: x[0]):
        node_edges = list(node_edges)
        random_edge = random.choice(node_edges)
        if random.random() < 0.5:
            random_edge = random_edge[::-1]
        G.add_edge(*random_edge)
        for e in node_edges:
            if random.random() < completeness:
                G.add_edge(*e)

    for (u,v,w) in G.edges(data=True):
        w['weight'] = random.randint(-5, 20)
        # w = random.randint(-5, 20)


    if draw:
        plt.figure(figsize=(10,6))
        if directed:
            # draw with edge weights
            pos = nx.arf_layout(G)
            nx.draw(G,pos, node_color='lightblue',
                    with_labels=True,
                    node_size=500,
                    arrowsize=20,
                    arrows=True)
            labels = nx.get_edge_attributes(G,'weight')
            nx.draw_networkx_edge_labels(G, pos,edge_labels=labels)

        else:
            nx.draw(G, node_color='lightblue',
                with_labels=True,
                node_size=500)

    return G

G = gnp_random_connected_graph(10, 1, False, True)
```




Павлосюк Роман Коваль Вікторія Мета: реалізувати алгоритми, вивчені на лекціях, та порівняти їх з вдудованими Нижче написані алгоритми Прима та Краскала

In [17]:

```
def kruskals_algorithm(graph: nx.classes.graph.Graph) -> list:
    '''Kruskal's algorithm implementation'''

    edges = sorted([(a,b,c['weight']) for a,b,c in list(graph.edges(data=True))], key= l
ambda x: x[2])
    mst_kraskal = []
    nodes_set = disjoint_set(list(graph.nodes))

    for v,u, _ in edges:
        if find(nodes_set, v) != find(nodes_set, u):
            mst_kraskal.append((v, u))
            union(nodes_set, v, u)
    return mst_kraskal

def find(nodes_set: dict, vertex: int):
    if nodes_set[vertex] != vertex:
        nodes_set[vertex] = find(nodes_set, nodes_set[vertex])
    return nodes_set[vertex]

def disjoint_set(list_of_vertices: list):
    return {k:k for k in list_of_vertices}

def union(nodes_set: dict, vertex1:int, vertex2:int):
    nodes_set[find(nodes_set, vertex1)] = find(nodes_set, vertex2)
```

In [18]:

```
def prims_algorithm(graph: nx.classes.graph.Graph) -> list:
    '''prim's algorithm implementation'''
    prim_mst = []
    visited = set()
    visited.add(list(graph.nodes)[0])
    edges = sorted([(u, v, w['weight']) for u, v, w in list(graph.edges(data = True))],
key = lambda x: x[2])
    while visited != set(graph.nodes):
        for u, v, _ in edges:
            if (u in visited and v not in visited):
                prim_mst.append((u, v))
```

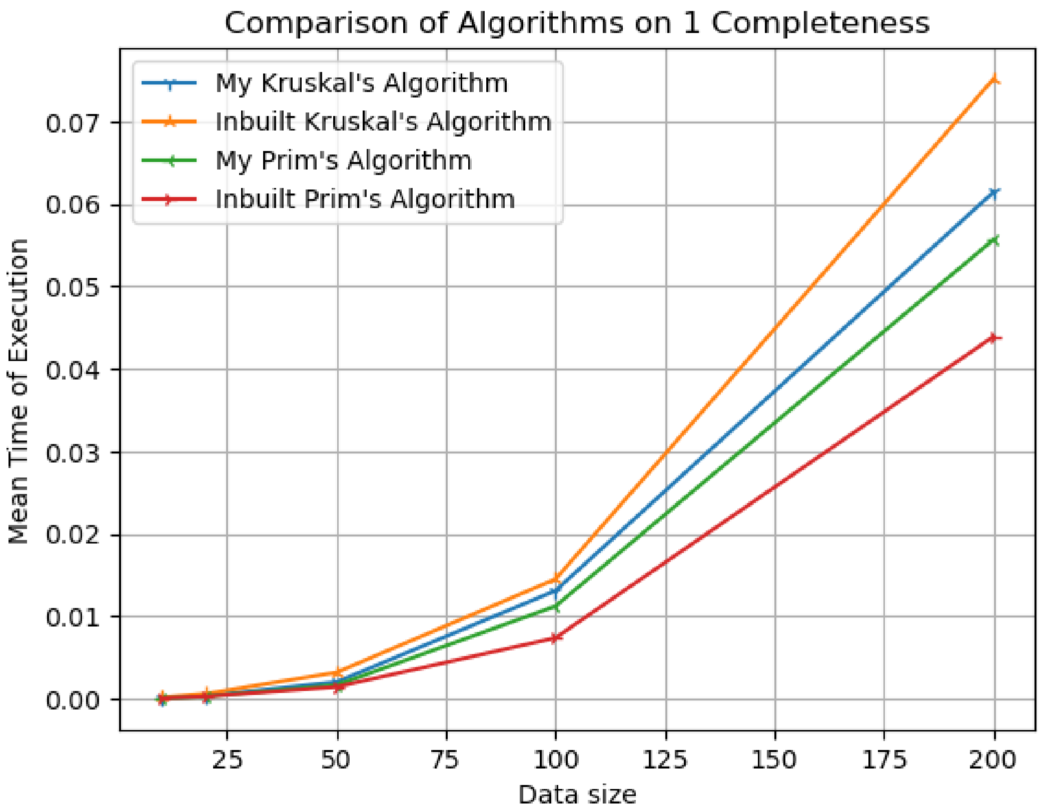
```python
                visited.add(v)
                edges.remove((u,v,_))
                break
            if (v in visited and u not in visited):
                prim_mst.append((u, v))
                visited.add(u)
                edges.remove((u,v,_))
                break
    return prim_mst
```

In [29]:

```python
list_num_nodes = [10, 20, 50, 100, 200]
# 1 completeness:
my_krusk_1 = [0.000095, 0.000347, 0.002031, 0.013096, 0.061401]
my_prim_1 = [0.000056, 0.000284, 0.001650, 0.011220, 0.055621]
notmy_krusk_1 = [0.000181, 0.000607, 0.003195, 0.014495, 0.075110]
notmy_prim_1 = [0.000084, 0.000286, 0.001445, 0.00738, 0.043895]
# 0.4 completeness
my_krusk_0 = [0.000053, 0.000196, 0.000865, 0.004372, 0.024299]
my_prim_0 = [0.000053, 0.000163, 0.000683, 0.003535, 0.021788]
notmy_krusk_0 = [0.000123, 0.000483, 0.001881, 0.005845, 0.025764]
notmy_prim_0 = [0.000072, 0.000225, 0.000872, 0.002903, 0.014101]
```

In [35]:

```python
plt.plot(list_num_nodes, my_krusk_1, marker='1', label="My Kruskal's Algorithm")
plt.plot(list_num_nodes, notmy_krusk_1, marker='2', label="Inbuilt Kruskal's Algorithm")
plt.plot(list_num_nodes, my_prim_1, marker='3', label="My Prim's Algorithm")
plt.plot(list_num_nodes, notmy_prim_1, marker='4', label="Inbuilt Prim's Algorithm")

plt.xlabel('Data size')
plt.ylabel('Mean Time of Execution')
plt.title('Comparison of Algorithms on 1 Completeness')
plt.legend()

plt.grid(True)
plt.show()
```

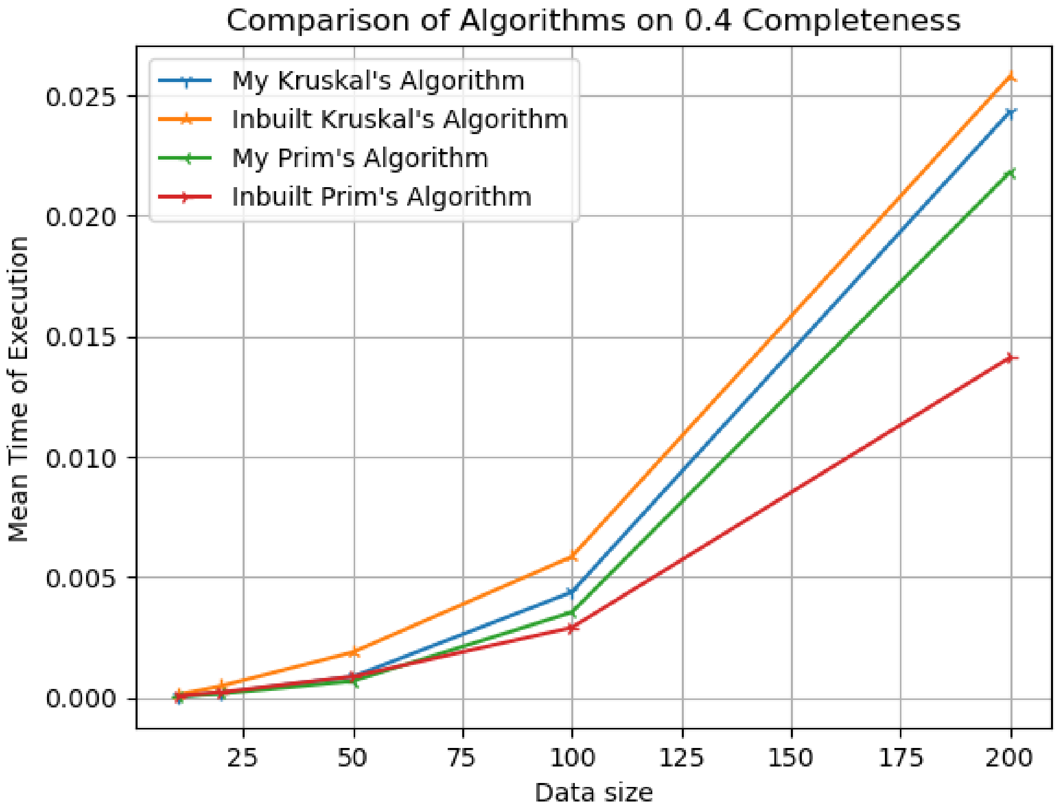



Як видно на графіку, мій алгоритм Краскала є кращим за вбудований. Я використовував **find-union method** для перевірки на цикл. Загальна складність мого алгоритму **V(n).** Але водночас вбудований алгоритм прима в рази швидший, особливо це помітно зі збільшенням к-сть вершин та заповненості графа.

Висновок: найкраще і найшвидше працює вбудований алгоритм Прима і загалом він є кращим за Краскала.

In [37]:

```
plt.plot(list_num_nodes, my_krusk_0, marker='1', label="My Kruskal's Algorithm")
plt.plot(list_num_nodes, notmy_krusk_0, marker='2', label="Inbuilt Kruskal's Algorithm")
plt.plot(list_num_nodes, my_prim_0, marker='3', label="My Prim's Algorithm")
plt.plot(list_num_nodes, notmy_prim_0, marker='4', label="Inbuilt Prim's Algorithm")

plt.xlabel('Data size')
plt.ylabel('Mean Time of Execution')
plt.title('Comparison of Algorithms on 0.4 Completeness')
plt.legend()

plt.grid(True)
plt.show()
```




In [ ]:

```
# NUM_OF_ITERATIONS = 1000
# time_taken = 0
# lst = []
# for num_node in list_num_nodes:
#     for i in tqdm(range(NUM_OF_ITERATIONS)):
#         # note that we should not measure time of graph creation
#         G = gnp_random_connected_graph(num_node, 0.4, False)
#         start = time.time()
#         # tree.minimum_spanning_tree(G, algorithm="kruskal")
#         tree.minimum_spanning_tree(G, algorithm="prim")
#         # kruskals_algorithm(G)
#         # prims_algorithm(G)
#         end = time.time()

#         time_taken += end - start
#     lst.append(round(time_taken / NUM_OF_ITERATIONS, 6))
# print(lst)
```