```
import sys

def dijkstra(graph, src):

    num_vertices = len(graph)
    dist = [sys.maxsize] * num_vertices
    dist[src] = 0
    visited = [False] * num_vertices

    for _ in range(num_vertices):

        min_distance = sys.maxsize
        min_index = -1

        for v in range(num_vertices):
            if not visited[v] and dist[v] < min_distance:
                min_distance = dist[v]
                min_index = v

        visited[min_index] = True

        for u, weight in enumerate(graph[min_index]):
            if weight > 0 and not visited[u] and dist[min_index] + weight < dist[u]:
                dist[u] = dist[min_index] + weight

    return dist

def adjacency_matrix(data, n):

    graph = [[0] * n for _ in range(n)]

    for line in data:
```

```
        parts = line.split()
        node = int(parts[0]) - 1

        for edge in parts[1:]:
            neighbor, weight = map(int, edge.split(','))
            graph[node][neighbor - 1] = weight
            graph[neighbor - 1][node] = weight

    return graph

import time

with open('test.txt', 'r') as file:
    test = file.readlines()

matrix_test = adjacency_matrix(test, 8)

start_time = time.time()
min_dist = dijkstra(matrix_test, 0)
end_time = time.time()

print(f'Время выполнения наивного алгоритма дейкстры: {end_time - start_time} секунд')
print(f'Минимальное расстояние до каждого из объектов: {min_dist}')

import time

with open('data.txt', 'r') as file:
    data = file.readlines()

matrix = adjacency_matrix(data, 200)
```

```
start_time = time.time()
min_dist = dijkstra(matrix, 0)
end_time = time.time()

print(f'Время выполнения наивного алгоритма дейкстры: {end_time - start_time} секунд')
print(f'Минимальное расстояние до каждого из объектов: {min_dist}')

import sys

class BinaryHeap:
    def __init__(self):
        self.heap = []  # Массив для хранения элементов ((вес, узел))
        self.positions = {}  # Словарь для отслеживания позиций узлов в куче (для быстрого доступа при уменьшении ключа)

    def insert(self, weight, node):
        self.heap.append((weight, node))
        self.positions[node] = len(self.heap) - 1
        self.sift_up(len(self.heap) - 1)

    def extract_min(self):

        if not self.heap:
            return None

        min_elem = self.heap[0]
        last_elem = self.heap.pop()

        if self.heap:
            self.heap[0] = last_elem
            self.positions[last_elem[1]] = 0
            self.sift_down(0)
```

```
        del self.positions[min_elem[1]]
        return min_elem

    def decrease_key(self, node, new_weight):
        index = self.positions[node]
        self.heap[index] = (new_weight, node)
        self.sift_up(index)

    def sift_up(self, index):

        while index > 0:

            parent_index = (index - 1) // 2

            if self.heap[index][0] < self.heap[parent_index][0]:
                self.swap(index, parent_index)
                index = parent_index
            else:
                break

    def sift_down(self, index):

        while 2 * index + 1 < len(self.heap):

            left_child = 2 * index + 1
            right_child = 2 * index + 2
            smallest = left_child

            if right_child < len(self.heap) and self.heap[right_child][0] < self.heap[left_child][0]:
                smallest = right_child

            if self.heap[index][0] > self.heap[smallest][0]:
```

```
                self.swap(index, smallest)
                index = smallest
            else:
                break

    def swap(self, i, j):
        self.positions[self.heap[i][1]] = j
        self.positions[self.heap[j][1]] = i
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def is_empty(self):
        return len(self.heap) == 0

def dijkstra_heap(graph, src):

    num_vertices = len(graph)
    dist = [sys.maxsize] * num_vertices
    dist[src] = 0
    min_heap = BinaryHeap()
    min_heap.insert(0, src)

    while not min_heap.is_empty():

        min_dist, u = min_heap.extract_min()

        for v, weight in enumerate(graph[u]):

            if weight > 0 and dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight

                if v in min_heap.positions:
                    min_heap.decrease_key(v, dist[v])
```

```
            else:
                min_heap.insert(dist[v], v)

    return dist

import time

with open('test.txt', 'r') as file:
    test = file.readlines()

matrix_test = adjacency_matrix(test, 8)

start_time = time.time()
min_dist = dijkstra_heap(matrix_test, 0)
end_time = time.time()

print(f'Время выполнения алгоритма Дейкстры на двоичной куче: {end_time - start_time} секунд')
print(f'Минимальное расстояние до каждого из объектов: {min_dist}')

import time

with open('data.txt', 'r') as file:
    data = file.readlines()

matrix = adjacency_matrix(data, 200)

start_time = time.time()
min_dist = dijkstra_heap(matrix, 0)
end_time = time.time()

print(f'Время выполнения алгоритма Дейкстры на двоичной куче: {end_time - start_time} секунд')
```

```
print(f'Минимальное расстояние до каждого из объектов: {min_dist}')


ver = [7, 37, 59, 82, 99, 115, 133, 165, 188, 197]


for v in ver:
    print(f'Минимальное расстояние до вершины {v}: {min_dist[v-1]}')
```