Run | Debug | Stop | Share | Save | { } Beautify | Language Python 3

main.py

```
import heapq
def dijkstra(edges, source, target):
    graph = {}
    for u, v, w in edges:
        if u not in graph:
            graph[u] = []
        if v not in graph:
            graph[v] = []
        graph[u].append((v, w))
        graph[v].append((u, w))
    pq = [(0, source)]
    dist = {node: float('inf') for node in graph}
    dist[source] = 0
    while pq:
        d, node = heapq.heappop(pq)
        if d > dist[node]:
            continue
        if node == target:
            return dist[node]
        for neighbor, weight in graph[node]:
            new_dist = d + weight
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
                heapq.heappush(pq, (new_dist, neighbor))
edges1 = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15), (2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9)]
source1 = 0
target1 = 4
print(dijkstra(edges1, source1, target1))
```

input

```
20


...Program finished with exit code 0
Press ENTER to exit console.
```

Run | Debug | Stop | Share | Save | { } Beautify | Language Python 3

main.py

```python
import heapq
from collections import defaultdict
class HuffmanNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq
def huffman_codes(characters, frequencies):
    n = len(characters)
    heap = [HuffmanNode(characters[i], frequencies[i]) for i in range(n)]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    root = heap[0]
    def generate_codes(node, current_code, codes):
        if node is None:
            return
        if node.char is not None:
            codes[node.char] = current_code
        generate_codes(node.left, current_code + '0', codes)
```

input

```
[('a', '00'), ('b', '01'), ('c', '10'), ('d', '11')]


...Program finished with exit code 0
Press ENTER to exit console.
```

Run | Debug | Stop | Share | Save | { } Beautify | Language Python 3

main.py

```python
import heapq
import sys
def dijkstra_adjacency_matrix(n, graph, source):
    distances = [sys.maxsize] * n
    distances[source] = 0
    pq = [(0, source)]
    while pq:
        current_distance, u = heapq.heappop(pq)
        if current_distance > distances[u]:
            continue
        for v in range(n):
            weight = graph[u][v]
            if weight < sys.maxsize:
                distance = current_distance + weight
                if distance < distances[v]:
                    distances[v] = distance
                    heapq.heappush(pq, (distance, v))
    return distances
n1 = 5
graph1 = [
    [0, 10, 3, float('inf'), float('inf')],
    [float('inf'), 0, 1, 2, float('inf')],
    [float('inf'), 4, 0, 8, 2],
    [float('inf'), float('inf'), float('inf'), 0, 7],
    [float('inf'), float('inf'), float('inf'), 9, 0]
]
source1 = 0
print(dijkstra_adjacency_matrix(n1, graph1, source1))
```

input

```
[0, 7, 3, 9, 5]


...Program finished with exit code 0
Press ENTER to exit console.
```

main.py

```
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq
def huffman_codes(characters, frequencies):
    n = len(characters)
    heap = [HuffmanNode(characters[i], frequencies[i]) for i in range(n)]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    root = heap[0]
    def generate_codes(node, current_code, codes):
        if node is None:
            return
        if node.char is not None:
            codes[node.char] = current_code
        generate_codes(node.left, current_code + '0', codes)
        generate_codes(node.right, current_code + '1', codes)
    codes = {}
    generate_codes(root, '', codes)
    return sorted(codes.items())
characters1 = ['a', 'b', 'c', 'd']
frequencies1 = [5, 9, 12, 13]
print(huffman_codes(characters1, frequencies1))
```

input

```
[('a', '00'), ('b', '01'), ('c', '10'), ('d', '11')]


...Program finished with exit code 0
Press ENTER to exit console.
```

Run | Debug | Stop | Share | Save | { } Beautify | Language Python 3

main.py

```python
import heapq
def huffman_decode(characters, frequencies, encoded_string):
    class HuffmanNode:
        def __init__(self, char, freq):
            self.char = char
            self.freq = freq
            self.left = None
            self.right = None
        def __lt__(self, other):
            return self.freq < other.freq
    n = len(characters)
    heap = [HuffmanNode(characters[i], frequencies[i]) for i in range(n)]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    root = heap[0]
    decoded_string = ""
    current_node = root
    for bit in encoded_string:
        if bit == '0':
            current_node = current_node.left
        else:
```

input

```
dbcbdd


...Program finished with exit code 0
Press ENTER to exit console.
```

Run | Debug | Stop | Share | Save | { } Beautify | Language Python 3

main.py

```
        def __lt__(self, other):
            return self.freq < other.freq
    n = len(characters)
    heap = [HuffmanNode(characters[i], frequencies[i]) for i in range(n)]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    root = heap[0]
    decoded_string = ""
    current_node = root
    for bit in encoded_string:
        if bit == '0':
            current_node = current_node.left
        else:
            current_node = current_node.right
        if current_node.left is None and current_node.right is None:
            decoded_string += current_node.char
            current_node = root
    return decoded_string
characters1 = ['a', 'b', 'c', 'd']
frequencies1 = [5, 9, 12, 13]
encoded_string1 = '1101100111110'
print(huffman_decode(characters1, frequencies1, encoded_string1))
```

input

```
dbcbdd


...Program finished with exit code 0
Press ENTER to exit console.
```