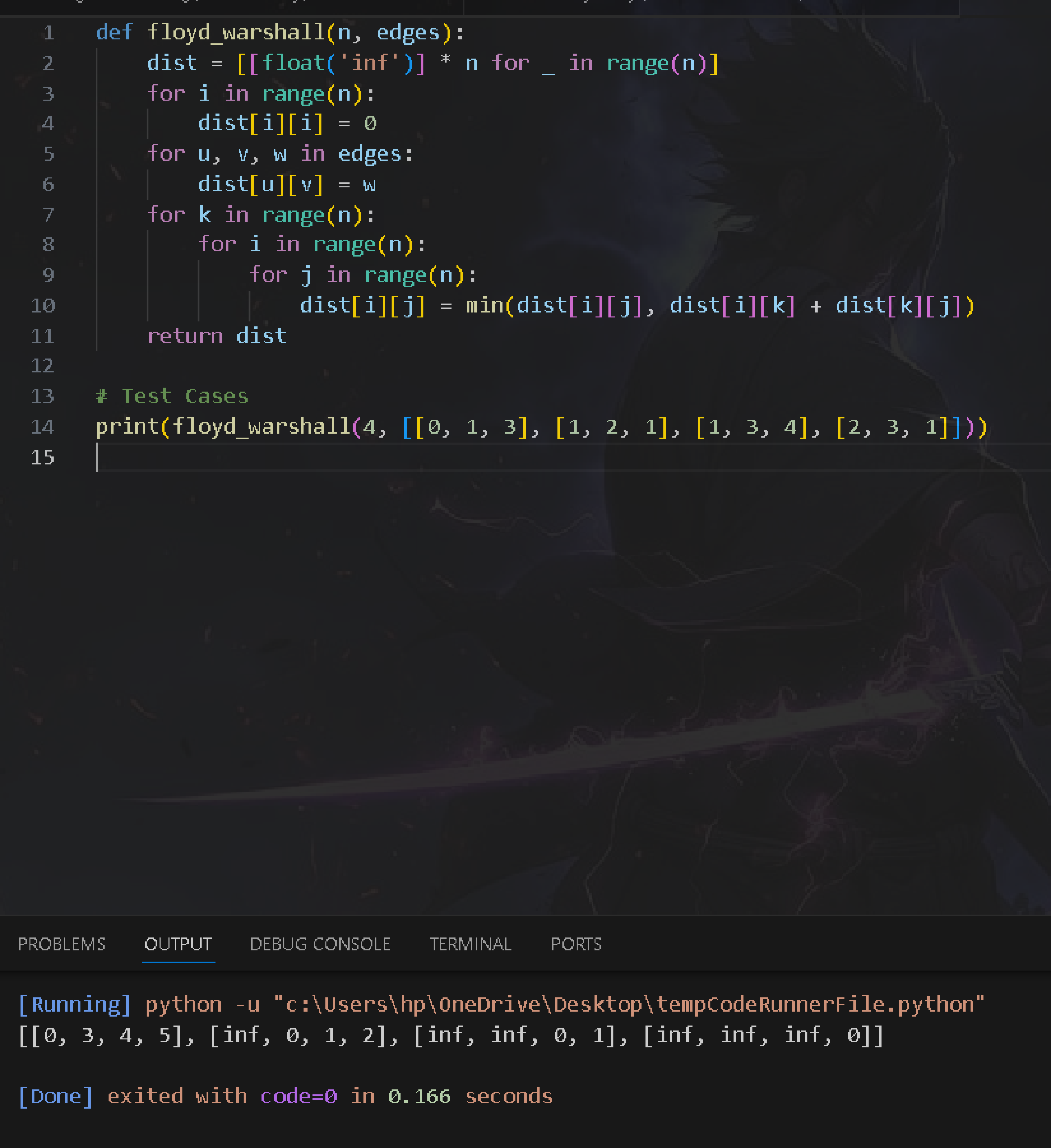
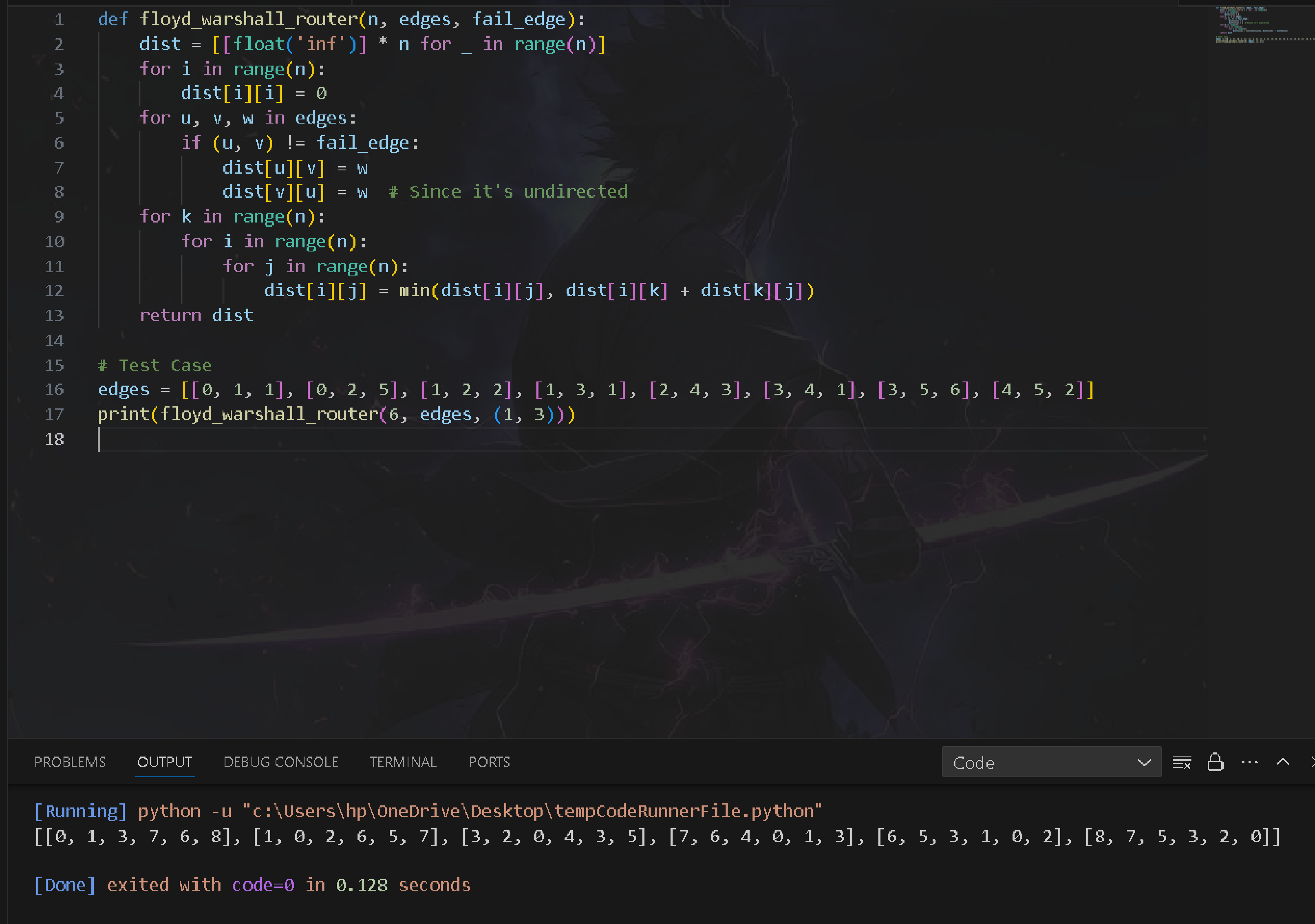


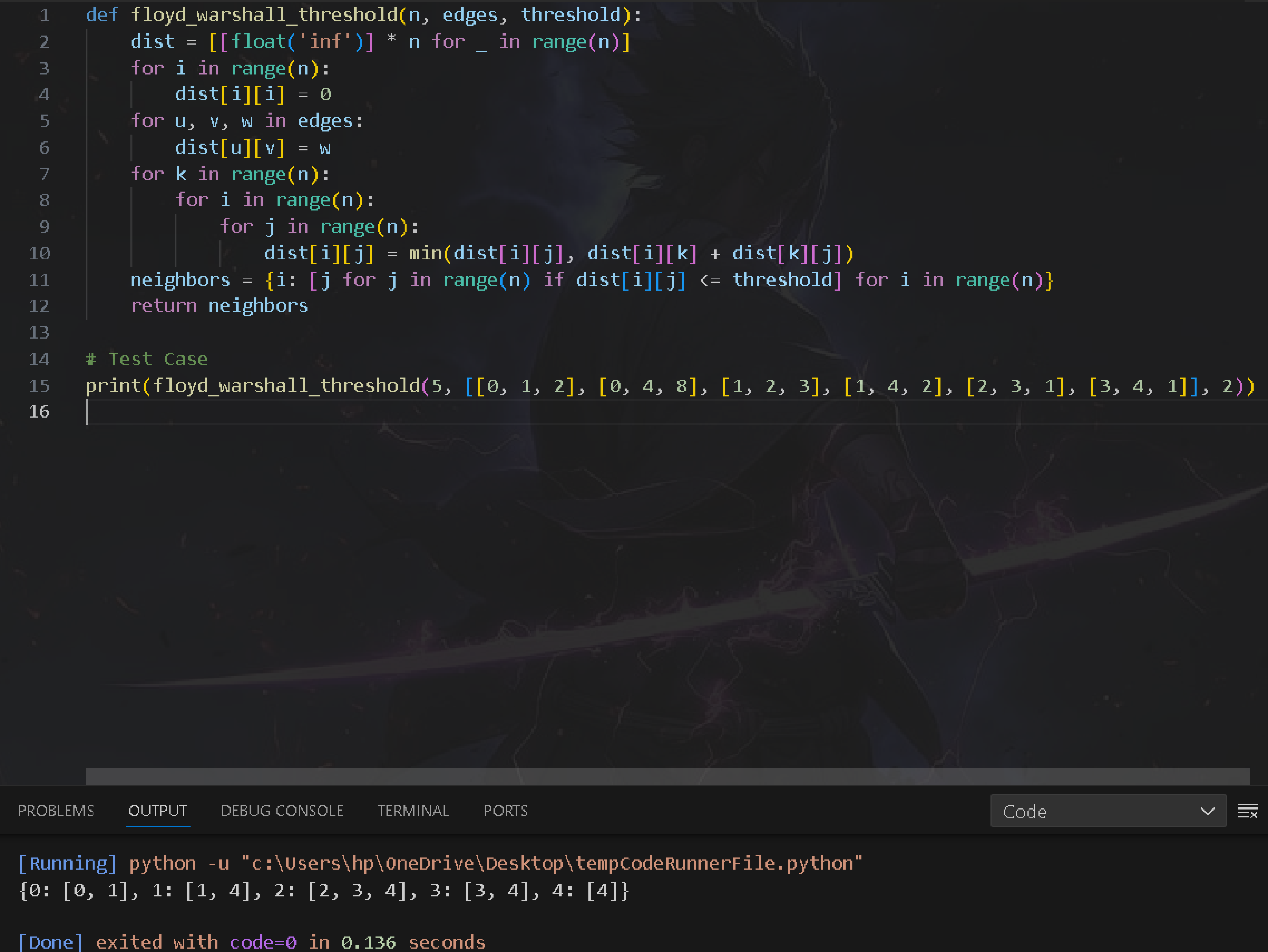
```python
def floyd_warshall(n, edges):
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
        dist[u][v] = w
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    return dist

# Test Cases
print(floyd_warshall(4, [[0, 1, 3], [1, 2, 1], [1, 3, 4], [2, 3, 1]]))

```

```
[Running] python -u "c:\Users\hp\OneDrive\Desktop\tempCodeRunnerFile.python"
[[0, 3, 4, 5], [inf, 0, 1, 2], [inf, inf, 0, 1], [inf, inf, inf, 0]]

[Done] exited with code=0 in 0.166 seconds
```



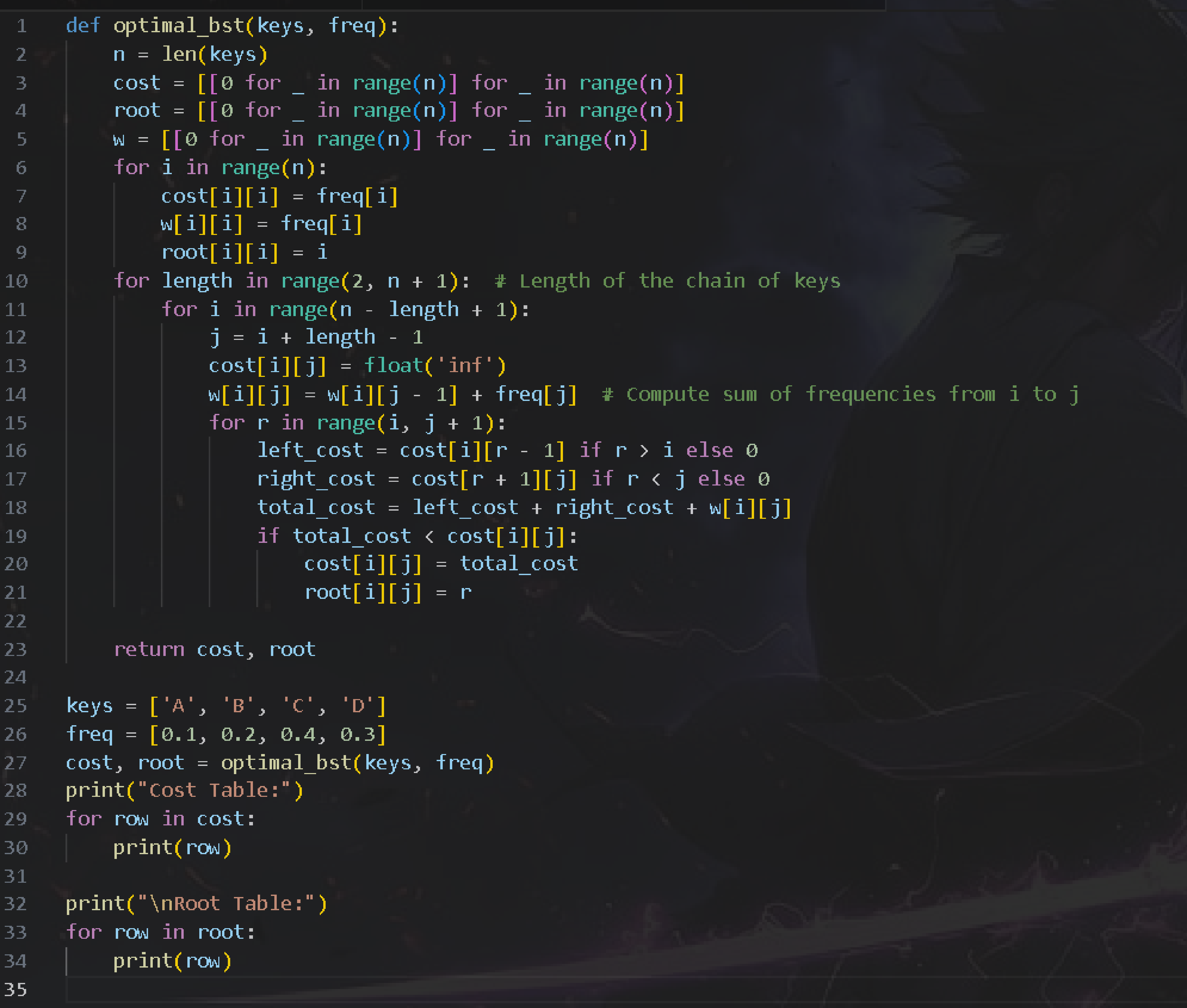
```python
def floyd_warshall_router(n, edges, fail_edge):
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
        if (u, v) != fail_edge:
            dist[u][v] = w
            dist[v][u] = w  # Since it's undirected
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    return dist

# Test Case
edges = [[0, 1, 1], [0, 2, 5], [1, 2, 2], [1, 3, 1], [2, 4, 3], [3, 4, 1], [3, 5, 6], [4, 5, 2]]
print(floyd_warshall_router(6, edges, (1, 3)))

```

```
[Running] python -u "c:\Users\hp\OneDrive\Desktop\tempCodeRunnerFile.python"
[[0, 1, 3, 7, 6, 8], [1, 0, 2, 6, 5, 7], [3, 2, 0, 4, 3, 5], [7, 6, 4, 0, 1, 3], [6, 5, 3, 1, 0, 2], [8, 7, 5, 3, 2, 0]]

[Done] exited with code=0 in 0.128 seconds
```

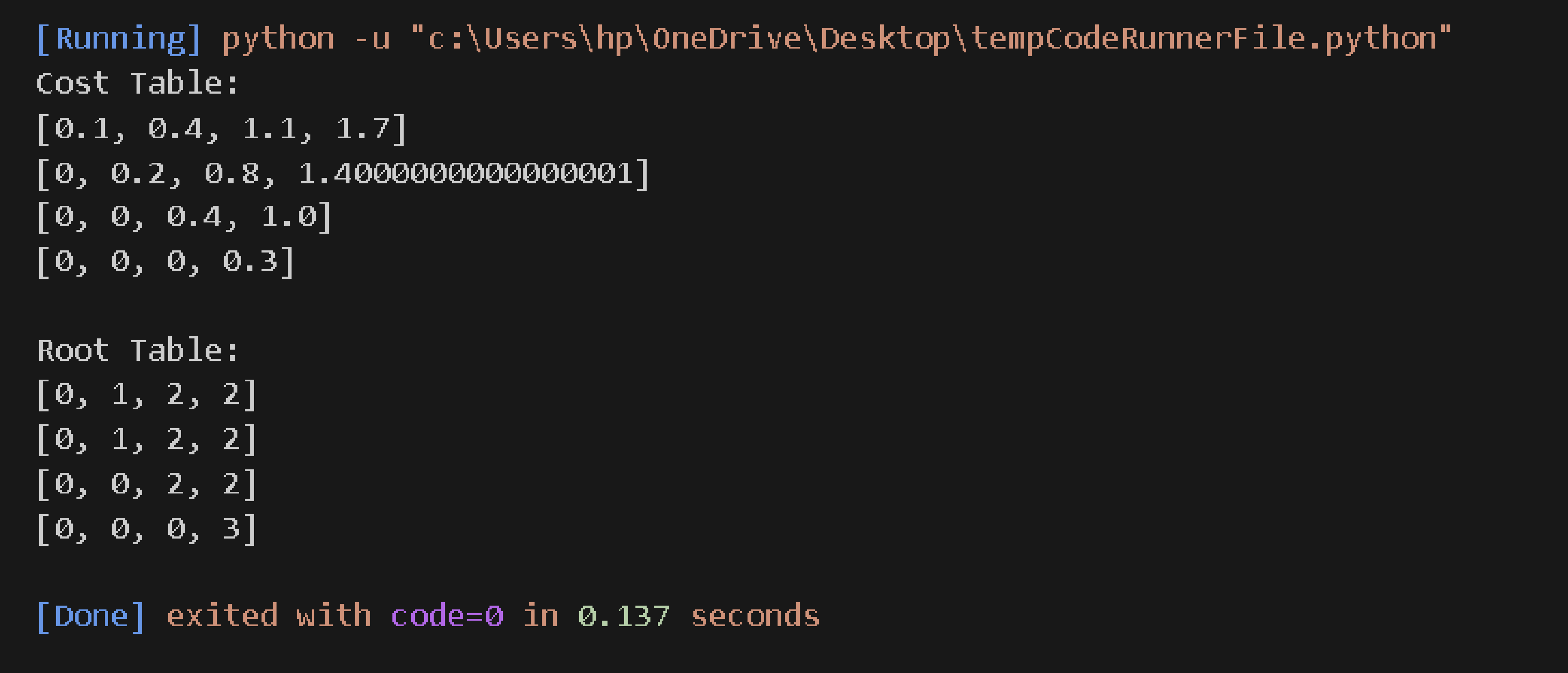


```python
def floyd_warshall_threshold(n, edges, threshold):
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
        dist[u][v] = w
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    neighbors = {i: [j for j in range(n) if dist[i][j] <= threshold] for i in range(n)}
    return neighbors

# Test Case
print(floyd_warshall_threshold(5, [[0, 1, 2], [0, 4, 8], [1, 2, 3], [1, 4, 2], [2, 3, 1], [3, 4, 1]], 2))
```

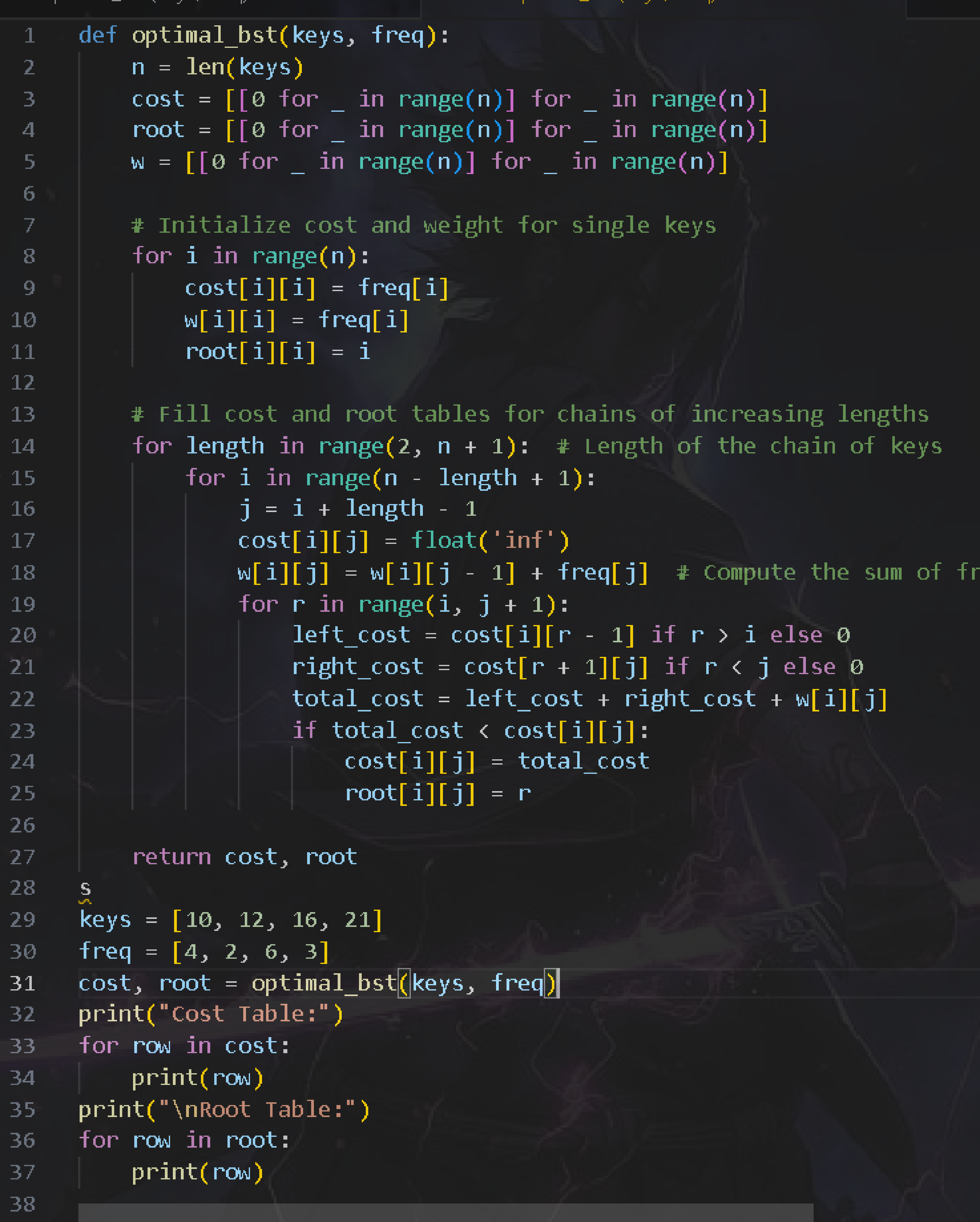
```
[Running] python -u "c:\Users\hp\OneDrive\Desktop\tempCodeRunnerFile.python"
{0: [0, 1], 1: [1, 4], 2: [2, 3, 4], 3: [3, 4], 4: [4]}

[Done] exited with code=0 in 0.136 seconds
```



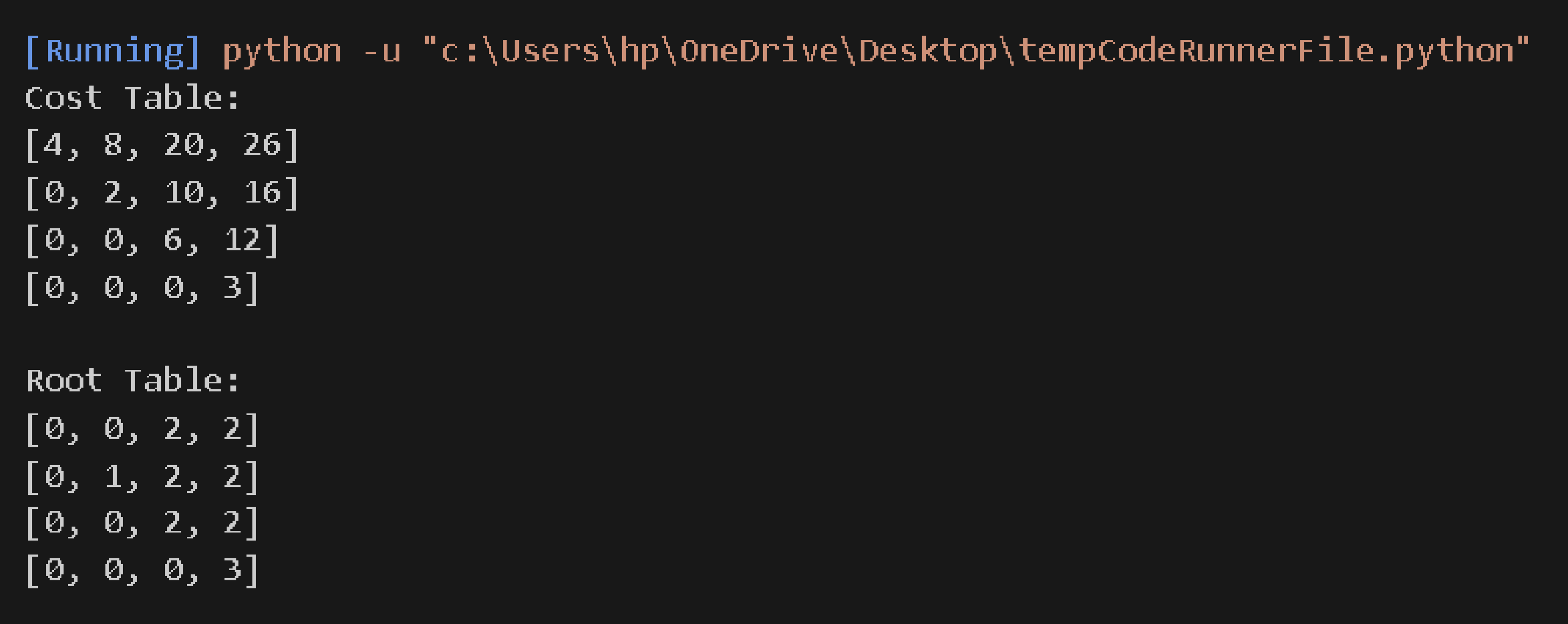
```python
def optimal_bst(keys, freq):
    n = len(keys)
    cost = [[0 for _ in range(n)] for _ in range(n)]
    root = [[0 for _ in range(n)] for _ in range(n)]
    w = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        cost[i][i] = freq[i]
        w[i][i] = freq[i]
        root[i][i] = i
    for length in range(2, n + 1):  # Length of the chain of keys
        for i in range(n - length + 1):
            j = i + length - 1
            cost[i][j] = float('inf')
            w[i][j] = w[i][j - 1] + freq[j]  # Compute sum of frequencies from i to j
            for r in range(i, j + 1):
                left_cost = cost[i][r - 1] if r > i else 0
                right_cost = cost[r + 1][j] if r < j else 0
                total_cost = left_cost + right_cost + w[i][j]
                if total_cost < cost[i][j]:
                    cost[i][j] = total_cost
                    root[i][j] = r

    return cost, root

keys = ['A', 'B', 'C', 'D']
freq = [0.1, 0.2, 0.4, 0.3]
cost, root = optimal_bst(keys, freq)
print("Cost Table:")
for row in cost:
    print(row)

print("\nRoot Table:")
for row in root:
    print(row)
```



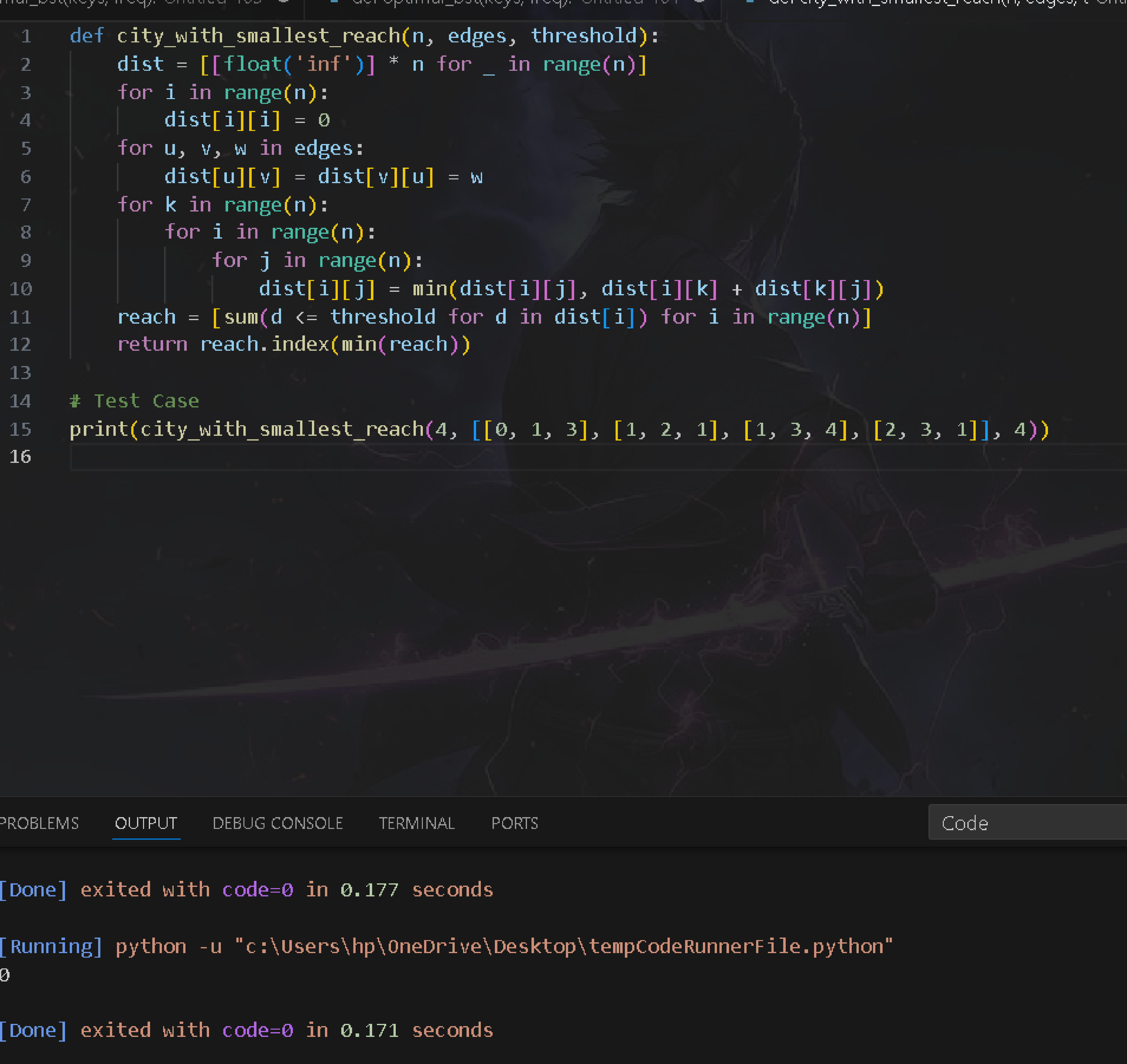
```
[Running] python -u "c:\Users\hp\OneDrive\Desktop\tempCodeRunnerFile.python"
Cost Table:
[0.1, 0.4, 1.1, 1.7]
[0, 0.2, 0.8, 1.4000000000000001]
[0, 0, 0.4, 1.0]
[0, 0, 0, 0.3]

Root Table:
[0, 1, 2, 2]
[0, 1, 2, 2]
[0, 0, 2, 2]
[0, 0, 0, 3]

[Done] exited with code=0 in 0.137 seconds
```



```python
def optimal_bst(keys, freq):
    n = len(keys)
    cost = [[0 for _ in range(n)] for _ in range(n)]
    root = [[0 for _ in range(n)] for _ in range(n)]
    w = [[0 for _ in range(n)] for _ in range(n)]

    # Initialize cost and weight for single keys
    for i in range(n):
        cost[i][i] = freq[i]
        w[i][i] = freq[i]
        root[i][i] = i

    # Fill cost and root tables for chains of increasing lengths
    for length in range(2, n + 1):  # Length of the chain of keys
        for i in range(n - length + 1):
            j = i + length - 1
            cost[i][j] = float('inf')
            w[i][j] = w[i][j - 1] + freq[j]  # Compute the sum of fr
            for r in range(i, j + 1):
                left_cost = cost[i][r - 1] if r > i else 0
                right_cost = cost[r + 1][j] if r < j else 0
                total_cost = left_cost + right_cost + w[i][j]
                if total_cost < cost[i][j]:
                    cost[i][j] = total_cost
                    root[i][j] = r

    return cost, root
s
keys = [10, 12, 16, 21]
freq = [4, 2, 6, 3]
cost, root = optimal_bst(keys, freq)
print("Cost Table:")
for row in cost:
    print(row)
print("\nRoot Table:")
for row in root:
    print(row)
```

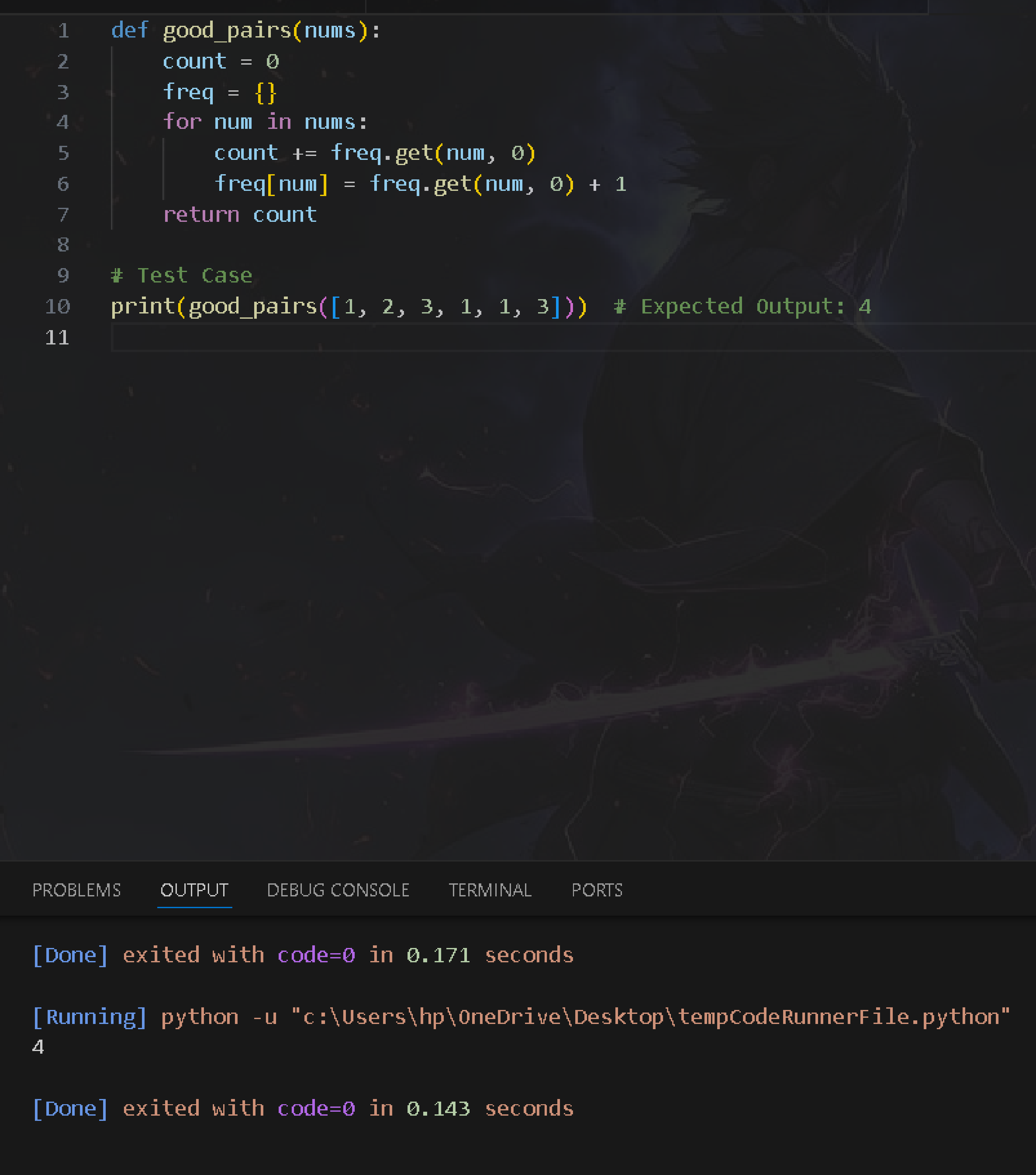


```
[Running] python -u "c:\Users\hp\OneDrive\Desktop\tempCodeRunnerFile.python"
Cost Table:
[4, 8, 20, 26]
[0, 2, 10, 16]
[0, 0, 6, 12]
[0, 0, 0, 3]

Root Table:
[0, 0, 2, 2]
[0, 1, 2, 2]
[0, 0, 2, 2]
[0, 0, 0, 3]
```



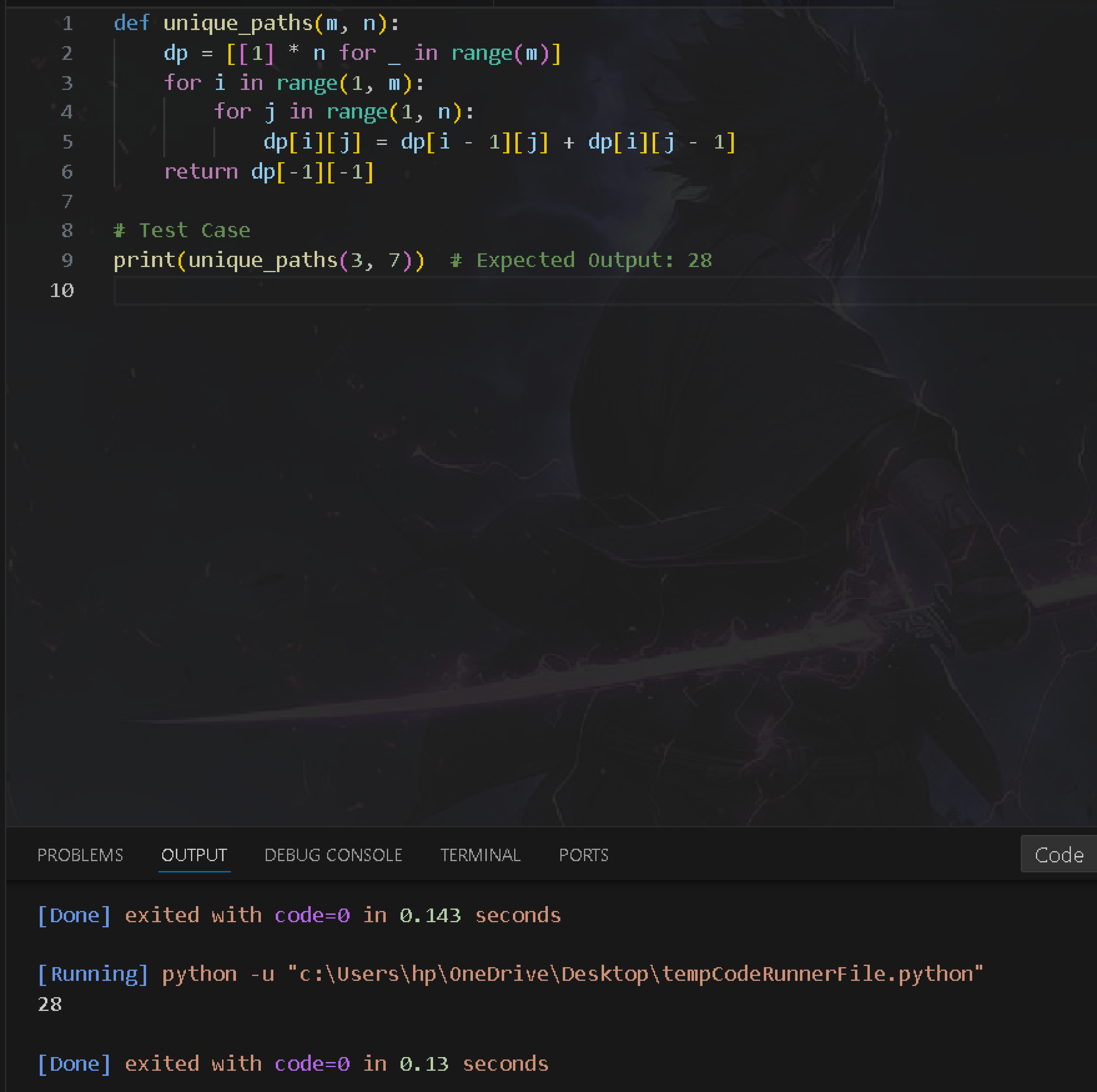
```python
def city_with_smallest_reach(n, edges, threshold):
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
        dist[u][v] = dist[v][u] = w
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    reach = [sum(d <= threshold for d in dist[i]) for i in range(n)]
    return reach.index(min(reach))

# Test Case
print(city_with_smallest_reach(4, [[0, 1, 3], [1, 2, 1], [1, 3, 4], [2, 3, 1]], 4))
```

```
[Done] exited with code=0 in 0.177 seconds

[Running] python -u "c:\Users\hp\OneDrive\Desktop\tempCodeRunnerFile.python"
0

[Done] exited with code=0 in 0.171 seconds
```



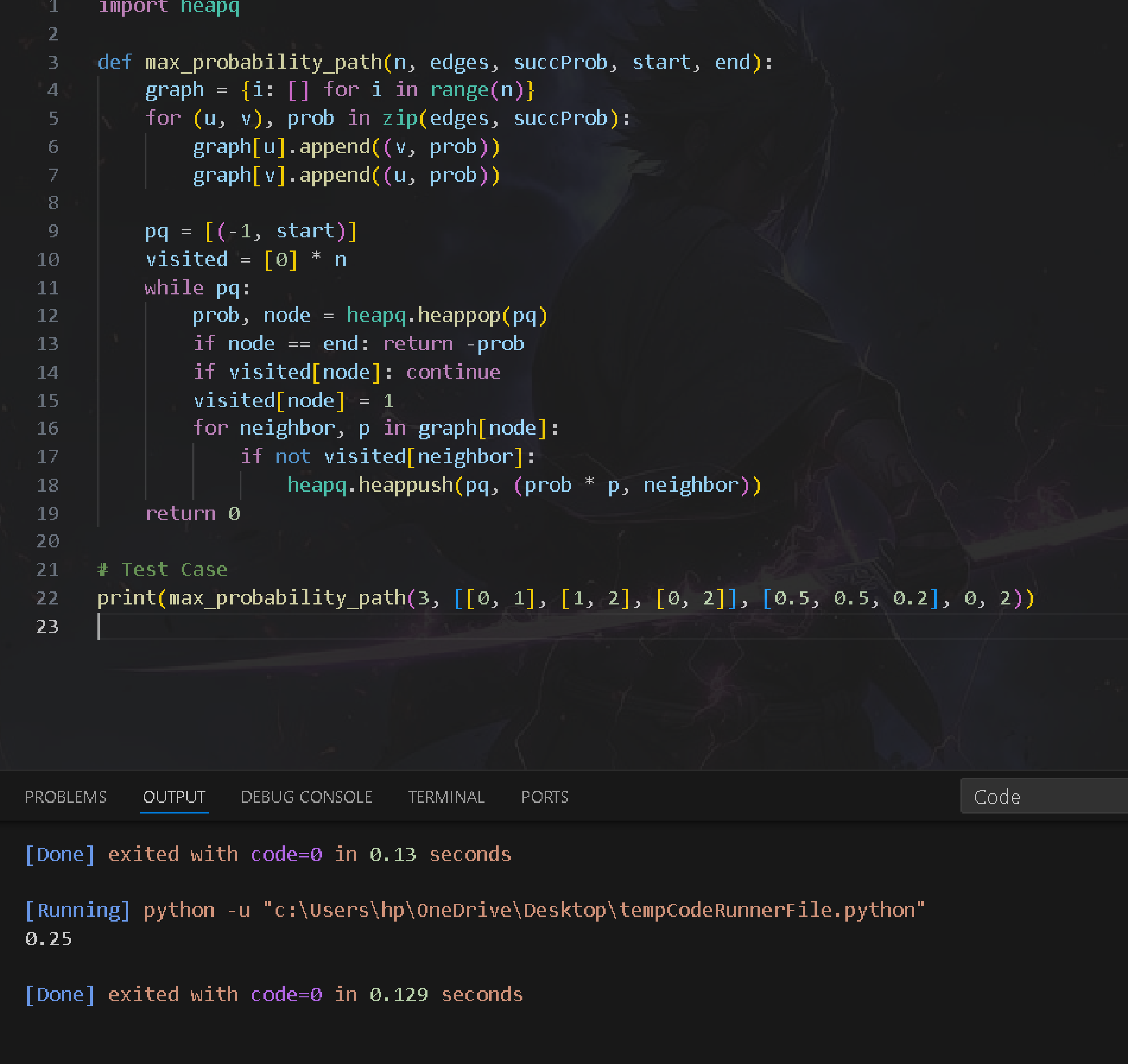
```python
def good_pairs(nums):
    count = 0
    freq = {}
    for num in nums:
        count += freq.get(num, 0)
        freq[num] = freq.get(num, 0) + 1
    return count

# Test Case
print(good_pairs([1, 2, 3, 1, 1, 3]))  # Expected Output: 4

```

```
[Done] exited with code=0 in 0.171 seconds

[Running] python -u "c:\Users\hp\OneDrive\Desktop\tempCodeRunnerFile.python"
4

[Done] exited with code=0 in 0.143 seconds
```



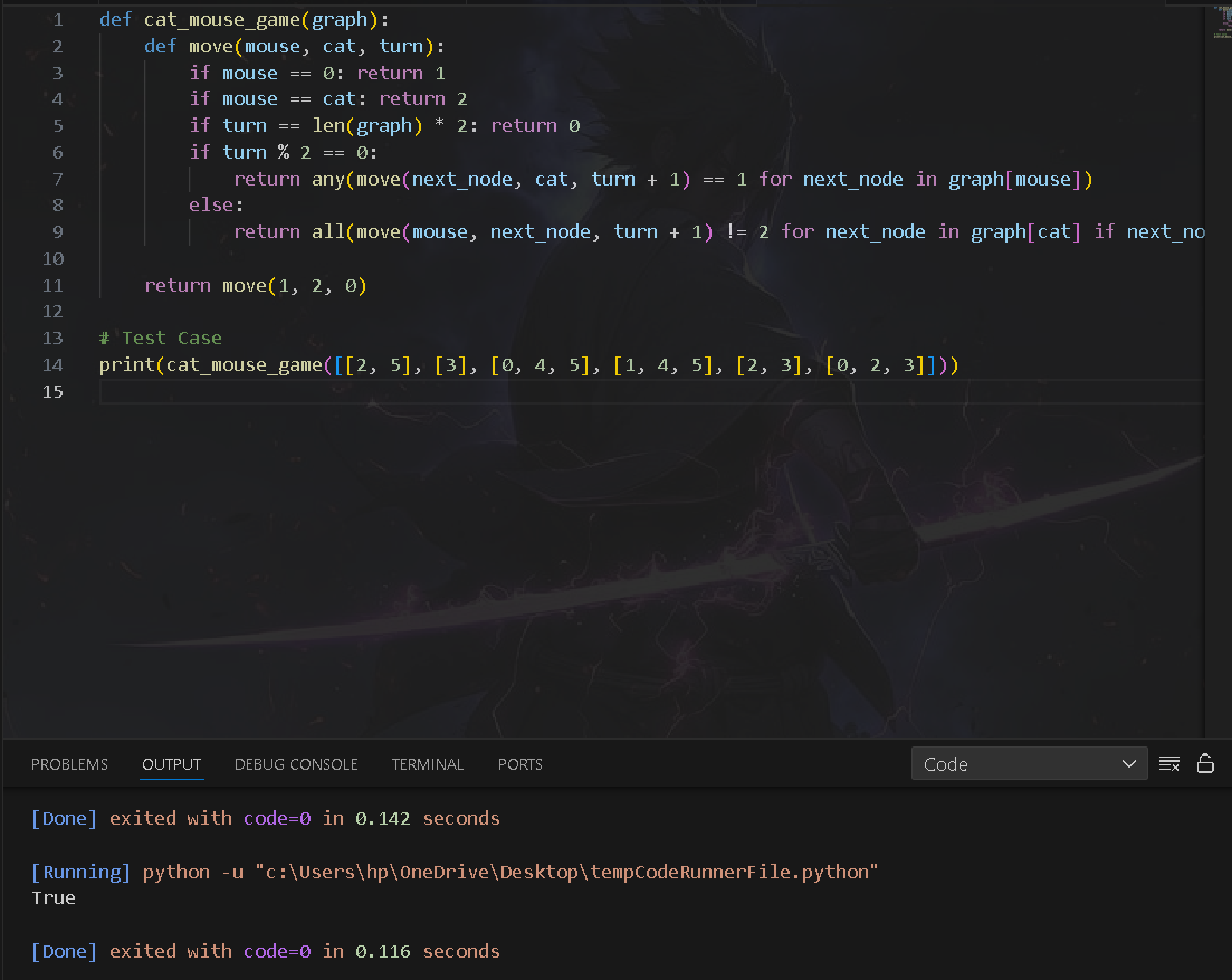
```python
def unique_paths(m, n):
    dp = [[1] * n for _ in range(m)]
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
    return dp[-1][-1]

# Test Case
print(unique_paths(3, 7))  # Expected Output: 28

```

```
[Done] exited with code=0 in 0.143 seconds

[Running] python -u "c:\Users\hp\OneDrive\Desktop\tempCodeRunnerFile.python"
28

[Done] exited with code=0 in 0.13 seconds
```



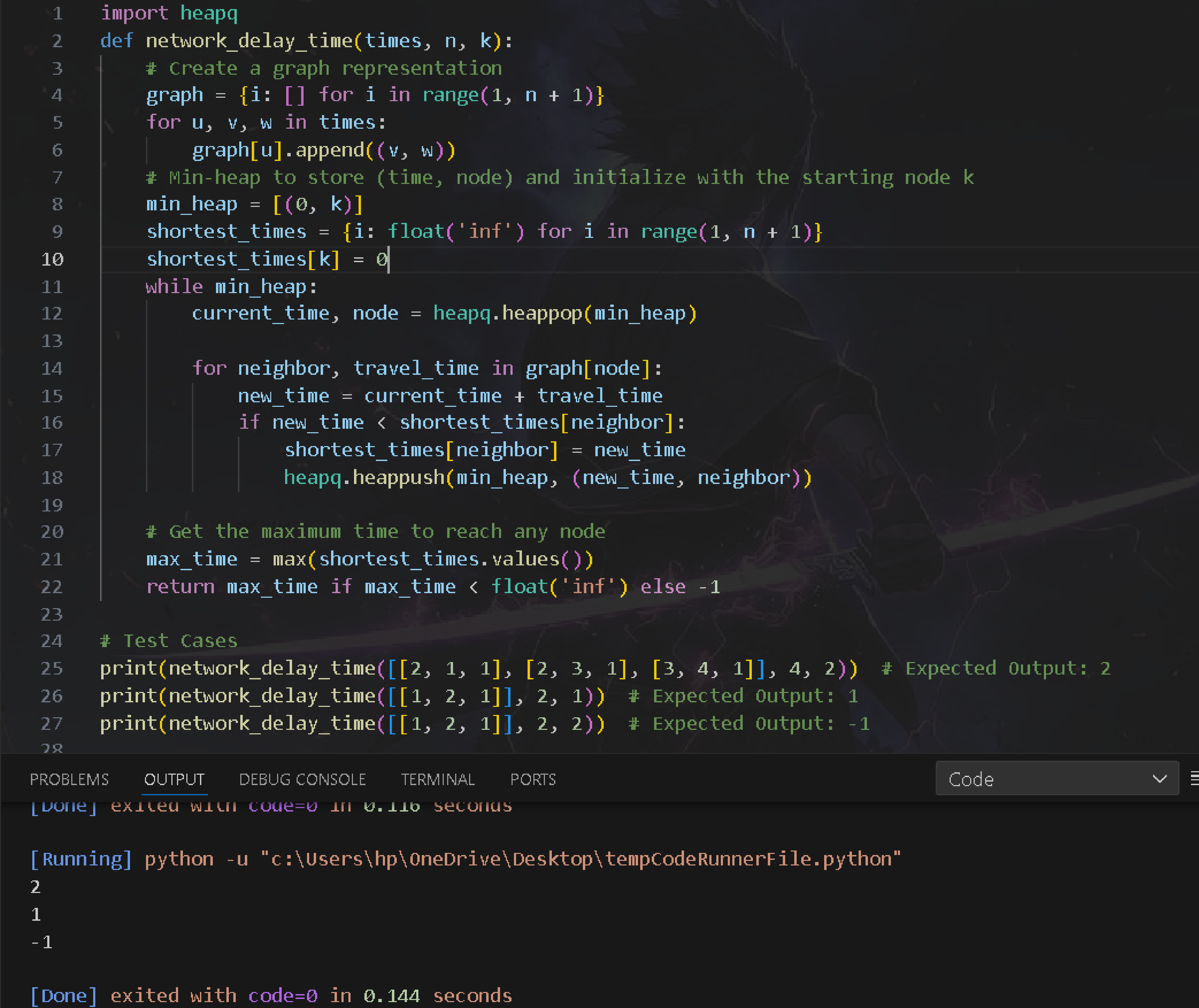
```python
import heapq

def max_probability_path(n, edges, succProb, start, end):
    graph = {i: [] for i in range(n)}
    for (u, v), prob in zip(edges, succProb):
        graph[u].append((v, prob))
        graph[v].append((u, prob))

    pq = [(-1, start)]
    visited = [0] * n
    while pq:
        prob, node = heapq.heappop(pq)
        if node == end: return -prob
        if visited[node]: continue
        visited[node] = 1
        for neighbor, p in graph[node]:
            if not visited[neighbor]:
                heapq.heappush(pq, (prob * p, neighbor))
    return 0

# Test Case
print(max_probability_path(3, [[0, 1], [1, 2], [0, 2]], [0.5, 0.5, 0.2], 0, 2))
```

```
[Done] exited with code=0 in 0.13 seconds

[Running] python -u "c:\Users\hp\OneDrive\Desktop\tempCodeRunnerFile.python"
0.25

[Done] exited with code=0 in 0.129 seconds
```



```python
def cat_mouse_game(graph):
    def move(mouse, cat, turn):
        if mouse == 0: return 1
        if mouse == cat: return 2
        if turn == len(graph) * 2: return 0
        if turn % 2 == 0:
            return any(move(next_node, cat, turn + 1) == 1 for next_node in graph[mouse])
        else:
            return all(move(mouse, next_node, turn + 1) != 2 for next_node in graph[cat] if next_no

    return move(1, 2, 0)

# Test Case
print(cat_mouse_game([[2, 5], [3], [0, 4, 5], [1, 4, 5], [2, 3], [0, 2, 3]]))
```

```
[Done] exited with code=0 in 0.142 seconds

[Running] python -u "c:\Users\hp\OneDrive\Desktop\tempCodeRunnerFile.python"
True

[Done] exited with code=0 in 0.116 seconds
```



```python
import heapq
def network_delay_time(times, n, k):
    # Create a graph representation
    graph = {i: [] for i in range(1, n + 1)}
    for u, v, w in times:
        graph[u].append((v, w))
    # Min-heap to store (time, node) and initialize with the starting node k
    min_heap = [(0, k)]
    shortest_times = {i: float('inf') for i in range(1, n + 1)}
    shortest_times[k] = 0
    while min_heap:
        current_time, node = heapq.heappop(min_heap)

        for neighbor, travel_time in graph[node]:
            new_time = current_time + travel_time
            if new_time < shortest_times[neighbor]:
                shortest_times[neighbor] = new_time
                heapq.heappush(min_heap, (new_time, neighbor))

    # Get the maximum time to reach any node
    max_time = max(shortest_times.values())
    return max_time if max_time < float('inf') else -1

# Test Cases
print(network_delay_time([[2, 1, 1], [2, 3, 1], [3, 4, 1]], 4, 2))  # Expected Output: 2
print(network_delay_time([[1, 2, 1]], 2, 1))  # Expected Output: 1
print(network_delay_time([[1, 2, 1]], 2, 2))  # Expected Output: -1
```

```
[Done] exited with code=0 in 0.116 seconds

[Running] python -u "c:\Users\hp\OneDrive\Desktop\tempCodeRunnerFile.python"
2
1
-1

[Done] exited with code=0 in 0.144 seconds
```