main.py

Share Run

Output Clear

```python
def optimal_bst(freq, n):
    cost = [[0] * n for _ in range(n)]
    for i in range(n):
        cost[i][i] = freq[i]
    for L in range(2, n + 1):
        for i in range(n - L + 1):
            j = i + L - 1
            cost[i][j] = float('inf')
            for r in range(i, j + 1):
                c = (cost[i][r - 1] if r > i else 0) + (cost[r + 1][j] if r < j else 0) + sum(freq[i:j + 1])
                cost[i][j] = min(cost[i][j], c)
    return cost[0][n - 1]
freq = [0.1, 0.2, 0.3, 0.4]
n = len(freq)
print("Optimal Cost:", optimal_bst(freq, n))
```

```
Optimal Cost: 1.8

=== Code Execution Successful ===
```

main.py

Share

Run

Output

Clear

```python
m, n = 3, 7
paths = 1
for i in range(1, m):
    paths = paths * (n - 1 + i) // i
print(paths)

```

```
28

=== Code Execution Successful ===
```

main.py

Share Run

```python
nums = [1, 2, 3, 1, 1, 3]
count = 0
freq = {}
for num in nums:
    if num in freq:
        count += freq[num]
        freq[num] += 1
    else:
        freq[num] = 1
print(count)
```

Output Clear

```
4

=== Code Execution Successful ===
```

main.py

Share Run

Output Clear

```python
graph = [[2, 5], [3], [0, 4, 5], [1, 4, 5], [2, 3], [0, 2, 3]]
mouse, cat = 1, 2
memo = {}
while True:
    if mouse == 0: print(1); break
    if mouse == cat: print(2); break
    if (mouse, cat) in memo: print(memo[(mouse, cat)]); break
    if mouse == 1:
        if any((next_mouse := m) != cat and (next_mouse, cat)
            not in memo for m in graph[mouse]):
            mouse = next(m for m in graph[mouse] if m != cat)
            memo[(mouse, cat)] = 1
        else:
            memo[(mouse, cat)] = 0
            print(0); break
    else:
        if any((next_cat := c) != 0 and (mouse, next_cat) not
            in memo for c in graph[cat]):
            cat = next(c for c in graph[cat] if c != 0)
            memo[(mouse, cat)] = 2
```

```
0

=== Code Execution Successful ===
```

main.py

Share

Run

Output

Clear

```python
def floyd_warshall(n, edges, distanceThreshold):
    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])
    return max(range(n), key=lambda i: sum(dist[i][j] <=
        distanceThreshold for j in range(n) if i != j))
print(floyd_warshall(4, [[0, 1, 3], [1, 2, 1], [1, 3, 4], [2, 3,
    1]], 4))
print(floyd_warshall(6, [[0, 1, 1], [0, 2, 5], [1, 2, 2], [1, 3,
    1], [2, 4, 3], [3, 4, 1], [3, 5, 6], [4, 5, 2]], 4))
```

```
1
1

=== Code Execution Successful ===
```

main.py

Share Run Output Clear

```python
INF = float('inf')
def floyd_warshall(n, edges):
    dist = [[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])
    return dist
n = 6
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]]
dist = floyd_warshall(n, edges)
print(f"Router A to Router F = {dist[0][5]}")
dist[1][3] = dist[3][1] = INF
dist = floyd_warshall(n, edges)
print(f"Router A to Router F = {dist[0][5]}")

```

```
Router A to Router F = 5
Router A to Router F = 5

=== Code Execution Successful ===
```

main.py

Share Run

Output Clear

```
    if mouse == cat: print(2); break
    if (mouse, cat) in memo: print(memo[(mouse, cat)]); break
    if mouse == 1:
        if any((next_mouse := m) != cat and (next_mouse, cat)
            not in memo for m in graph[mouse]):
            mouse = next(m for m in graph[mouse] if m != cat)
            memo[(mouse, cat)] = 1
        else:
            memo[(mouse, cat)] = 0
            print(0); break
    else:
        if any((next_cat := c) != 0 and (mouse, next_cat) not
            in memo for c in graph[cat]):
            cat = next(c for c in graph[cat] if c != 0)
            memo[(mouse, cat)] = 2
        else:
            memo[(mouse, cat)] = 0
            print(0); break
    mouse, cat = cat, mouse
```

```
0

=== Code Execution Successful ===
```

main.py

Share Run

Output Clear

```python
n = 4
edges = [[0, 1, 3], [1, 2, 1], [1, 3, 4], [2, 3, 1]]
threshold = 4
graph = [[] for _ in range(n)]
for u, v, w in edges:
    graph[u].append((v, w))
    graph[v].append((u, w))
def dijkstra(start):
    dist = [float('inf')] * n
    dist[start], queue = 0, [(0, start)]
    while queue:
        d, city = queue.pop(0)
        for neighbor, weight in graph[city]:
            if d + weight < dist[neighbor]:
                dist[neighbor] = d + weight
                queue.append((dist[neighbor], neighbor))
                queue.sort()
    return dist
best_city, min_count = -1, float('inf')
for city in range(n):
```

```
3

=== Code Execution Successful ===
```

main.py

Share Run

Output Clear

```python
def optimal_bst(freq, n):
    cost = [[0] * n for _ in range(n)]
    for i in range(n):
        cost[i][i] = freq[i]
    for L in range(2, n + 1):
        for i in range(n - L + 1):
            j = i + L - 1
            cost[i][j] = float('inf')
            for r in range(i, j + 1):
                c = (cost[i][r - 1] if r > i else 0) + (cost[r + 1][j] if r < j else 0) + sum(freq[i:j + 1])
                cost[i][j] = min(cost[i][j], c)
    return cost[0][n - 1]
freq = [4,2,6,3]
n = len(freq)
print("Optimal Cost:", optimal_bst(freq, n))
```

```
Optimal Cost: 26

=== Code Execution Successful ===
```

main.py

Share Run Output Clear

```
    graph[v].append((u, w))
def dijkstra(start):
    dist = [float('inf')] * n
    dist[start], queue = 0, [(0, start)]
    while queue:
        d, city = queue.pop(0)
        for neighbor, weight in graph[city]:
            if d + weight < dist[neighbor]:
                dist[neighbor] = d + weight
                queue.append((dist[neighbor], neighbor))
                queue.sort()
    return dist
best_city, min_count = -1, float('inf')
for city in range(n):
    count = sum(d <= threshold for d in dijkstra(city)) - 1
    if count < min_count or (count == min_count and city >
        best_city):
        min_count, best_city = count, city
print(best_city)
```

```
3

=== Code Execution Successful ===
```

main.py

Share Run

Output Clear

```python
n,start, end,graph = 3,0,2,{}
edges,succProb = [[0, 1], [1, 2], [0, 2]],[0.5, 0.5, 0.3]
for (a, b), prob in zip(edges, succProb):
    graph.setdefault(a, []).append((b, prob))
    graph.setdefault(b, []).append((a, prob))
maxProb = [0] * n
maxProb[start] = 1
toVisit = [(start, 1)]
while toVisit:
    node, prob = toVisit.pop()
    if node == end:
        print(f"{prob:.5f}"); break
    for neighbor, success in graph.get(node, []):
        newProb = prob * success
        if newProb > maxProb[neighbor]:
            maxProb[neighbor] = newProb
            toVisit.append((neighbor, newProb))
else:
    print(0)
```

```
0.30000

=== Code Execution Successful ===
```

main.py

Share Run

```python
INF = float('inf')
def floyd_warshall(n, edges, distanceThreshold):
    dist = [[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])
    neighbors_count = [sum(dist[i][j] <= distanceThreshold for j
        in range(n) if i != j) for i in range(n)]
    return min(range(n), key=lambda i: neighbors_count[i])
n = 5
edges = [[0, 1, 2], [0, 4, 8], [1, 2, 3], [1, 4, 2], [2, 3, 1],
    [3, 4, 1]]
distanceThreshold = 2
print(floyd_warshall(n, edges, distanceThreshold))
```

Output

Clear

```
0

=== Code Execution Successful ===
```

main.py

Share Run

```python
times, n, k = [[2, 1, 1], [2, 3, 1], [3, 4, 1]], 4, 2
graph = [[] for _ in range(n + 1)]
for u, v, w in times:
    graph[u].append((v, w))
def dijkstra(start):
    dist = [float('inf')] * (n + 1)
    dist[start] = 0
    queue = [(0, start)]
    while queue:
        d, node = queue.pop(0)
        for neighbor, weight in graph[node]:
            if d + weight < dist[neighbor]:
                dist[neighbor] = d + weight
                queue.append((dist[neighbor], neighbor))
                queue.sort()
    return dist[1:]
result = dijkstra(k)
max_time = max(result)
print(max_time if max_time < float('inf') else -1)
```

Output

Clear

```
2

=== Code Execution Successful ===
```