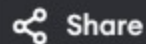




main.py



Run

Output

Clear

```
1 def optimal_bst(freq, n):
2     cost = [[0] * n for _ in range(n)]
3     for i in range(n):
4         cost[i][i] = freq[i]
5         for L in range(2, n + 1):
6             for i in range(n - L + 1):
7                 j = i + L - 1
8                 cost[i][j] = float('inf')
9                 for r in range(i, j + 1):
10                     c = (cost[i][r - 1] if r > i else 0) + (cost[r +
11                        1][j] if r < j else 0) + sum(freq[i:j + 1])
12                     cost[i][j] = min(cost[i][j], c)
13     return cost[0][n - 1]
14 freq = [0.1, 0.2, 0.3, 0.4]
15 n = len(freq)
16 print("Optimal Cost:", optimal_bst(freq, n))
```

Optimal Cost: 1.8

=== Code Execution Successful ===



JS



main.py



Share

Run

Output

Clear



JS



```
1 m, n = 3, 7
2 paths = 1
3 for i in range(1, m):
4     paths = paths * (n - 1 + i) // i
5 print(paths)
6
```

28

=== Code Execution Successful ===



main.py



Share

Run

Output

Clear

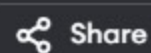
```
1 nums = [1, 2, 3, 1, 1, 3]
2 count = 0
3 freq = {}
4 for num in nums:
5     if num in freq:
6         count += freq[num]
7         freq[num] += 1
8     else:
9         freq[num] = 1
10 print(count)
```

4

=== Code Execution Successful ===



main.py



Run

Output

Clear

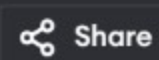
```
1 graph = [[2, 5], [3], [0, 4, 5], [1, 4, 5], [2, 3], [0, 2, 3]]
2 mouse, cat = 1, 2
3 memo = {}
4 while True:
5     if mouse == 0: print(1); break
6     if mouse == cat: print(2); break
7     if (mouse, cat) in memo: print(memo[(mouse, cat)]); break
8     if mouse == 1:
9         if any((next_mouse := m) != cat and (next_mouse, cat)
10              not in memo for m in graph[mouse]):
11             mouse = next(m for m in graph[mouse] if m != cat)
12             memo[(mouse, cat)] = 1
13         else:
14             memo[(mouse, cat)] = 0
15             print(0); break
16     else:
17         if any((next_cat := c) != 0 and (mouse, next_cat) not
18              in memo for c in graph[cat]):
19             cat = next(c for c in graph[cat] if c != 0)
20             memo[(mouse, cat)] = 2
```

0

=== Code Execution Successful ===



main.py



Run

Output

Clear



```
1 def floyd_warshall(n, edges, distanceThreshold):
2     dist = [[float('inf')] * n for _ in range(n)]
3     for i in range(n): dist[i][i] = 0
4     for u, v, w in edges: dist[u][v] = dist[v][u] = w
5     for k in range(n):
6         for i in range(n):
7             for j in range(n):
8                 dist[i][j] = min(dist[i][j], dist[i][k] +
                                   dist[k][j])
9     return max(range(n), key=lambda i: sum(dist[i][j] <=
        distanceThreshold for j in range(n) if i != j))
10 print(floyd_warshall(4, [[0, 1, 3], [1, 2, 1], [1, 3, 4], [2, 3,
    1]], 4))
11 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))
```

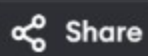
1

1

=== Code Execution Successful ===



main.py



Run

Output

Clear



```
1 INF = float('inf')
2 def floyd_warshall(n, edges):
3     dist = [[INF]*n for _ in range(n)]
4     for i in range(n): dist[i][i] = 0
5     for u, v, w in edges: dist[u][v] = dist[v][u] = w
6     for k in range(n):
7         for i in range(n):
8             for j in range(n):
9                 dist[i][j] = min(dist[i][j], dist[i][k] +
                                dist[k][j])
10    return dist
11 n = 6
12 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]]
13 dist = floyd_warshall(n, edges)
14 print(f"Router A to Router F = {dist[0][5]}")
15 dist[1][3] = dist[3][1] = INF
16 dist = floyd_warshall(n, edges)
17 print(f"Router A to Router F = {dist[0][5]}")
18
```

Router A to Router F = 5

Router A to Router F = 5

=== Code Execution Successful ===



main.py



Share

Run

Output

Clear

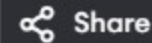
```
6     if mouse == cat: print(2); break
7     if (mouse, cat) in memo: print(memo[(mouse, cat)]); break
8     if mouse == 1:
9         if any((next_mouse := m) != cat and (next_mouse, cat)
               not in memo for m in graph[mouse]):
10            mouse = next(m for m in graph[mouse] if m != cat)
11            memo[(mouse, cat)] = 1
12        else:
13            memo[(mouse, cat)] = 0
14            print(0); break
15    else:
16        if any((next_cat := c) != 0 and (mouse, next_cat) not
               in memo for c in graph[cat]):
17            cat = next(c for c in graph[cat] if c != 0)
18            memo[(mouse, cat)] = 2
19        else:
20            memo[(mouse, cat)] = 0
21            print(0); break
22    mouse, cat = cat, mouse
```

0

=== Code Execution Successful ===



main.py



Share

Run

Output

Clear

```
1 n = 4
2 edges = [[0, 1, 3], [1, 2, 1], [1, 3, 4], [2, 3, 1]]
3 threshold = 4
4 graph = [[] for _ in range(n)]
5 for u, v, w in edges:
6     graph[u].append((v, w))
7     graph[v].append((u, w))
8 def dijkstra(start):
9     dist = [float('inf')] * n
10    dist[start], queue = 0, [(0, start)]
11    while queue:
12        d, city = queue.pop(0)
13        for neighbor, weight in graph[city]:
14            if d + weight < dist[neighbor]:
15                dist[neighbor] = d + weight
16                queue.append((dist[neighbor], neighbor))
17            queue.sort()
18    return dist
19 best_city, min_count = -1, float('inf')
20 for city in range(n):
```

```
3
=== Code Execution Successful ===
```

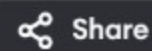


JS

GO



main.py



Run

Output

Clear

```
1 def optimal_bst(freq, n):
2     cost = [[0] * n for _ in range(n)]
3     for i in range(n):
4         cost[i][i] = freq[i]
5     for L in range(2, n + 1):
6         for i in range(n - L + 1):
7             j = i + L - 1
8             cost[i][j] = float('inf')
9             for r in range(i, j + 1):
10                 c = (cost[i][r - 1] if r > i else 0) + (cost[r +
11                     1][j] if r < j else 0) + sum(freq[i:j + 1])
12                 cost[i][j] = min(cost[i][j], c)
13     return cost[0][n - 1]
14 freq = [4, 2, 6, 3]
15 n = len(freq)
16 print("Optimal Cost:", optimal_bst(freq, n))
```

Optimal Cost: 26

=== Code Execution Successful ===

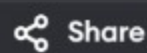


JS





main.py



Run

Output

Clear

```
7 graph[v].append((u, w))
8 def dijkstra(start):
9     dist = [float('inf')] * n
10    dist[start], queue = 0, [(0, start)]
11    while queue:
12        d, city = queue.pop(0)
13        for neighbor, weight in graph[city]:
14            if d + weight < dist[neighbor]:
15                dist[neighbor] = d + weight
16                queue.append((dist[neighbor], neighbor))
17            queue.sort()
18    return dist
19 best_city, min_count = -1, float('inf')
20 for city in range(n):
21     count = sum(d <= threshold for d in dijkstra(city)) - 1
22     if count < min_count or (count == min_count and city >
23                             best_city):
24         min_count, best_city = count, city
25 print(best_city)
```

3

=== Code Execution Successful ===



main.py



Share

Run

Output

Clear

```
1 n, start, end, graph = 3, 0, 2, {}
2 edges, succProb = [[0, 1], [1, 2], [0, 2]], [0.5, 0.5, 0.3]
3 for (a, b), prob in zip(edges, succProb):
4     graph.setdefault(a, []).append((b, prob))
5     graph.setdefault(b, []).append((a, prob))
6 maxProb = [0] * n
7 maxProb[start] = 1
8 toVisit = [(start, 1)]
9 while toVisit:
10     node, prob = toVisit.pop()
11     if node == end:
12         print(f"{prob:.5f}"); break
13     for neighbor, success in graph.get(node, []):
14         newProb = prob * success
15         if newProb > maxProb[neighbor]:
16             maxProb[neighbor] = newProb
17             toVisit.append((neighbor, newProb))
18 else:
19     print(0)
```

0.30000

=== Code Execution Successful ===



main.py



Share

Run

Output

Clear



JS



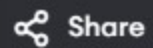
```
1 INF = float('inf')
2 def floyd_warshall(n, edges, distanceThreshold):
3     dist = [[INF]*n for _ in range(n)]
4     for i in range(n): dist[i][i] = 0
5     for u, v, w in edges: dist[u][v] = dist[v][u] = w
6     for k in range(n):
7         for i in range(n):
8             for j in range(n):
9                 dist[i][j] = min(dist[i][j], dist[i][k] +
                                   dist[k][j])
10    neighbors_count = [sum(dist[i][j] <= distanceThreshold for j
                             in range(n) if i != j) for i in range(n)]
11    return min(range(n), key=lambda i: neighbors_count[i])
12 n = 5
13 edges = [[0, 1, 2], [0, 4, 8], [1, 2, 3], [1, 4, 2], [2, 3, 1],
            [3, 4, 1]]
14 distanceThreshold = 2
15 print(floyd_warshall(n, edges, distanceThreshold))
```

0

=== Code Execution Successful ===



main.py



Run

Output

Clear

```
1 times, n, k = [[2, 1, 1], [2, 3, 1], [3, 4, 1]], 4, 2
2 graph = [[] for _ in range(n + 1)]
3 for u, v, w in times:
4     graph[u].append((v, w))
5 def dijkstra(start):
6     dist = [float('inf')] * (n + 1)
7     dist[start] = 0
8     queue = [(0, start)]
9     while queue:
10         d, node = queue.pop(0)
11         for neighbor, weight in graph[node]:
12             if d + weight < dist[neighbor]:
13                 dist[neighbor] = d + weight
14                 queue.append((dist[neighbor], neighbor))
15             queue.sort()
16     return dist[1:]
17 result = dijkstra(k)
18 max_time = max(result)
19 print(max_time if max_time < float('inf') else -1)
```

2

=== Code Execution Successful ===