```
1 def floyds_algorithm(n, edges,
        distanceThreshold):
        INF = float('inf')
 2
        dist = [[INF for _ in range(n)] for _
 3
            in range(n)]
        for i in range(n):
 4 -
 5
            dist[i][i] = 0
 for u, v, w in edges:
7
            dist[u][v] = w
8
     for k in range(n):
            for i in range(n):
 9 -
10 -
                for j in range(n):
                    if dist[i][k] + dist[k][j]
11
                        < dist[i][j]:
12
                        dist[i][j] =
                         dist[i][k] +
                         dist[k][j]
        shortest_path_count = sum(1 for row in
13
            dist for d in row if d <=
            distanceThreshold)
        return shortest_path_count
14
15
   n = 4
    edges = [[0, 1, 3], [1, 2, 1], [1, 3, 4],
16
        [2, 3, 1]]
   distanceThreshold = 4
17
    shortest_path = floyds_algorithm(n, edges,
18
        distanceThreshold)
19 print(shortest_path)
```

main.py Output ∴∴ ♣ ▷

9
=== Code Execution Successful ===

```
INF = 999999
 1
    graph = [[0, 5, INF, 10],
 2
             [INF, 0, 3, INF],
 3
             [INF, INF, 0, 1],
 4
 5
             [INF, INF, INF, 0]]
6 def floyd_warshall(graph):
 7
        n = len(graph)
        for k in range(n):
8
            for i in range(n):
9 -
                for j in range(n):
10
11
                     graph[i][j] = min
                         (graph[i][j],
                         graph[i][k] +
                         graph[k][j])
12
   floyd_warshall(graph)
    print("Router A to Router F =",graph[0][3]
13
    graph[1][3] = INF
14
   graph[3][1] = INF
15
16
   floyd_warshall(graph)
   print("Router A to Router F =",graph[0][3]
17
        )
```

main.py Output → ♣ ♦ ▶

Router A to Router F = 9

Router A to Router F = 9

=== Code Execution Successful ===

```
INF = 999999
 1
 2
    cities = ['A', 'B', 'C', 'D']
3
    distances = [[0, 2, INF, INF],
 4
                 [INF, 0, INF, INF],
                 [INF, 7, 0, 1],
5
6
                 [6, INF, INF, 0]]
7 for k in range(len(cities)):
8
        for i in range(len(cities)):
9 -
            for j in range(len(cities)):
                if distances[i][j]
10
                    >distances[i][k] +
                    distances[k][j]:
                    distances[i][j] =
11
                        distances[i][k] +
                         distances[k][j]
    start_city = 'C'
12
13
    end_city = 'A'
    start_index = cities.index(start_city)
14
15
    end_index = cities.index(end_city)
16
   shortest_distance =
        distances[start_index][end_index]
   print(f"Shortest path from {start_city} to
17
        {end_city} is {shortest_distance}")
```

main.py Output

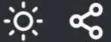
Shortest path from C to A is 7

```
import numpy as np
    def optimal_bst(keys, freq):
        n = len(keys)
 3
        cost = np.zeros((n, n))
 4
 5
        root = np.zeros((n, n))
 6
        for i in range(n):
 7
            cost[i][i] = freq[i]
 8
            root[i][i] = i
 9 for L in range(2, n + 1):
10
            for i in range(n - L + 1):
11
                j = i + L - 1
                cost[i][j] = float('inf')
12
13
                for r in range(i, j + 1):
                    c = cost[i][r - 1] if r >
14
                         i else 0
15
                    c += cost[r + 1][j] if r <
                         j else 0
16
                    c += sum(freq[i:j + 1])
17
                    if c < cost[i][j]:</pre>
18
                         cost[i][j] = c
19
                         root[i][j] = r
20
        return cost[0][n - 1]
   keys = ['A', 'B', 'C', 'D']
21
   freq = [0.1, 0.2, 0.4, 0.3]
22
23
   result = optimal_bst(keys, freq)
    print(result)
24
```

main.py Output -☆- ぱ ►>>

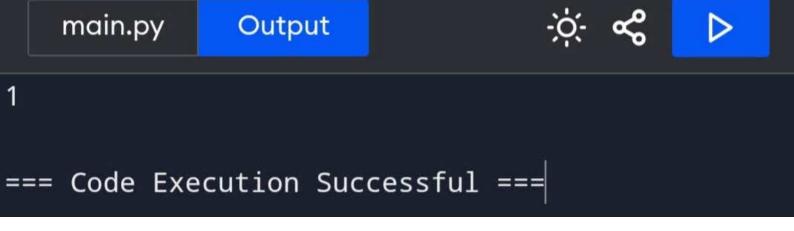
1.7

```
1 def optimal_bst(keys, freq):
        n = len(keys)
2
        cost = [[0 for _ in range(n)] for _ in
3
            range(n)]
        root = [[0 for _ in range(n)] for _ in
4
            range(n)]
5 -
        for i in range(n):
6
            cost[i][i] = freq[i]
        for L in range(2, n + 1):
7 -
8
            for i in range(n - L + 1):
                j = i + L - 1
9
                cost[i][j] = float('inf')
10
                for r in range(i, j + 1):
11 -
                    c = cost[i][r - 1] if r >
12
                         i else 0
13
                    c += cost[r + 1][j] if r <
                         j else 0
                    c += sum(freq[i:j + 1])
14
15
                    if c < cost[i][j]:
16
                         cost[i][j] = c
                         root[i][j] = r
17
18
        return cost[0][n - 1]
19
    keys = [10, 12, 16, 21]
    freq = [4, 2, 6, 3]
20
   result = optimal_bst(keys, freq)
21
    print(result)
22
```





```
1 def cat_mouse_game(graph):
        n = len(graph)
 2
        dp = [[[0]*n for _ in range(n)] for _
 3
            in range(2)]
        for i in range(n):
 4
 5
            for j in range(n):
 6
                dp[0][i][j] = 1
7
                dp[1][i][j] = 2
8 -
        for i in range(n):
            dp[0][0][i] = dp[1][0][i] = 2
9
10
        for i in range(n):
11
            dp[0][i][i] = dp[1][i][i] = 0
        for i in range(n):
12 -
13
            dp[0][i][0] = dp[1][i][0] = 1
14
        for i in range(n):
            dp[0][i][n-1] = dp[1][i][n-1] = 2
15
16
        return dp[0][1][2]
17
   graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3]
        ,[0,2,3]]
   result = cat_mouse_game(graph)
18
    print(result)
19
```



```
1 import math 28
2 m, n = 3, 7
3 print(math.comb(m + n - 2, m - 1)) === Code Execution Successful ===
```

```
from collections import defaultdict
    import heapq
 2
 3 def maxProbability(n, edges, succProb,
        start, end):
        graph = defaultdict(list)
 4
        for i, (a, b) in enumerate(edges):
 5 -
            graph[a].append((b, succProb[i]))
 6
 7
            graph[b].append((a, succProb[i]))
        pq = [(-1, start)]
8
        probs = [0] * n
 9
        probs[start] = 1
10
       while pq:
11 -
            prob, node = heapq.heappop(pq)
12
13
            if node == end:
14
                return -prob
            for nei, nei_prob in graph[node]:
15
                if -prob * nei_prob >
16
                    probs[nei]:
                    probs[nei] = -prob *
17
                        nei_prob
                    heapq.heappush(pq, (prob *
18
                        nei_prob, nei))
19
        return 0
20
   n = 3
    edges = [[0, 1], [1, 2], [0, 2]]
21
    succProb = [0.5, 0.5, 0.2]
22
23
   start = 0
   end = 2
24
   print(maxProbability(n, edges, succProb,
25
        start, end))
```

main.py Output

0.25

```
1 n, edges, dist = 4, [[0,1,3],[1,2,1],[1,3,4],[2 3
        ,-, III, ·
                                                      === Code Execution Successful ===
 2 d = [[float('inf')]*n for _ in range(n)]
 3 for i, j, w in edges: d[i][j] = d[j][i] = w
 4 for k in range(n):
        for i in range(n):
 5
            for j in range(n):
 6
                d[i][j] = min(d[i][j], d[i][k] +
                    d[k][j])
 8 res, min_reach = 0, n
 9 for 1 in range(n):
        reach = sum(1 for j in range(n) if 1 != j
10
            and d[i][j] \Leftarrow dist)
        if reach <= min_reach: res, min_reach = 1,</pre>
11
12 print(res)
13
```

```
import heapq
 1
    times = [[2,1,1],[2,3,1],[3,4,1]]
 2
   n = 4
3
4 k = 2
    graph = \{i: [] for i in range(1, n + 1)\}
 5
6 for u, v, w in times:
        graph[u].append((v, w))
 7
    dist = {i: float('inf') for i in range(1,
 8
        n + 1)
   dist[k] = 0
 9
    pq = [(0, k)]
10
11 while pq:
        d, node = heapq.heappop(pq)
12
13     if d > dist[node]:
            continue
14
for neighbor, weight in graph[node]:
            if d + weight < dist[neighbor]:</pre>
16 -
                dist[neighbor] = d + weight
17
                heapq.heappush(pq,
18
                    (dist[neighbor], neighbor
    print(max(dist.values()) if all(d < float</pre>
19
        ('inf') for d in dist.values()) else
        -1)
```

main.py Output

2