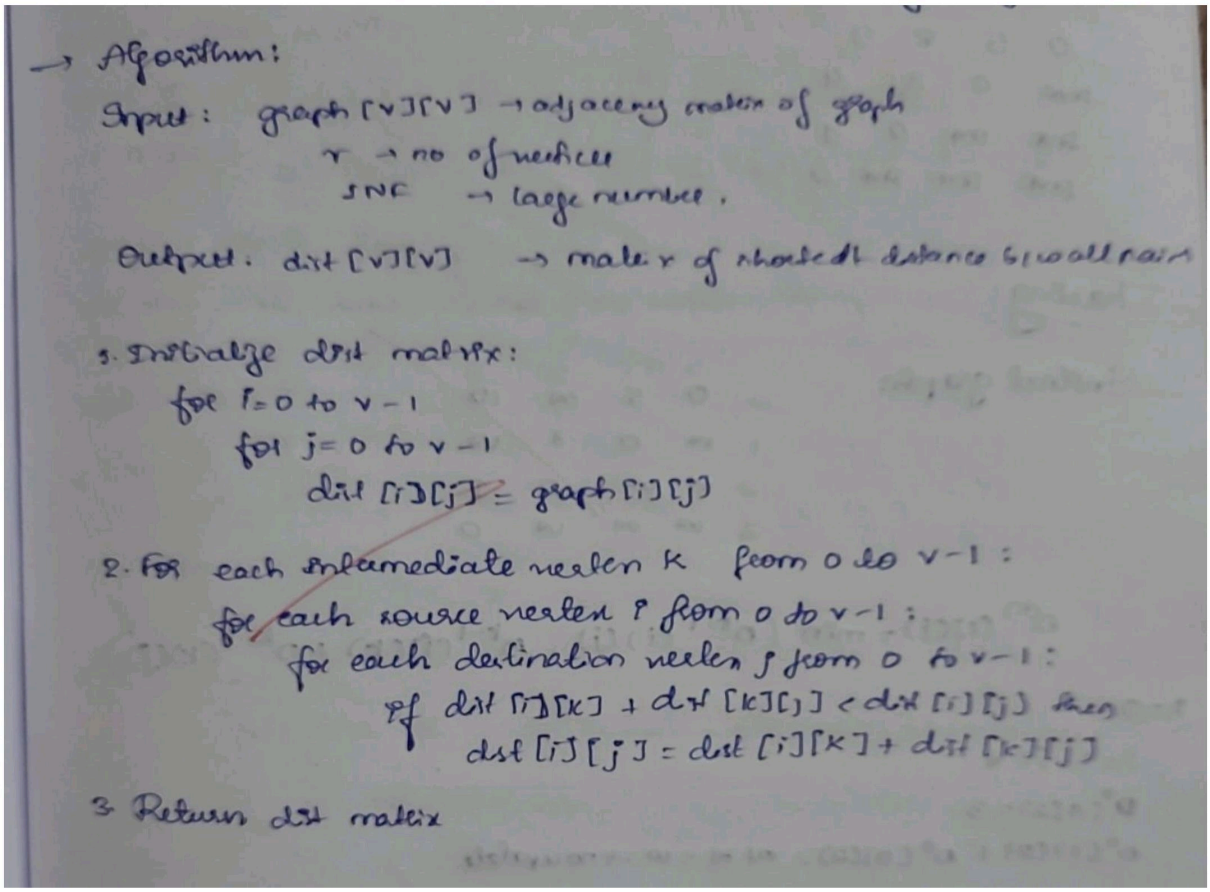
PROBLEM 6

Implement All Pair Shortest paths problem using Floyd's algorithm.

ALGORITHM



CODE

```
#include <stdio.h>


#define INF 99999
#define V 4  // Number of vertices


// Function to print the shortest distance matrix
void printSolution(int dist[V][V]) {
    printf("Shortest distances between every pair of vertices:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
```

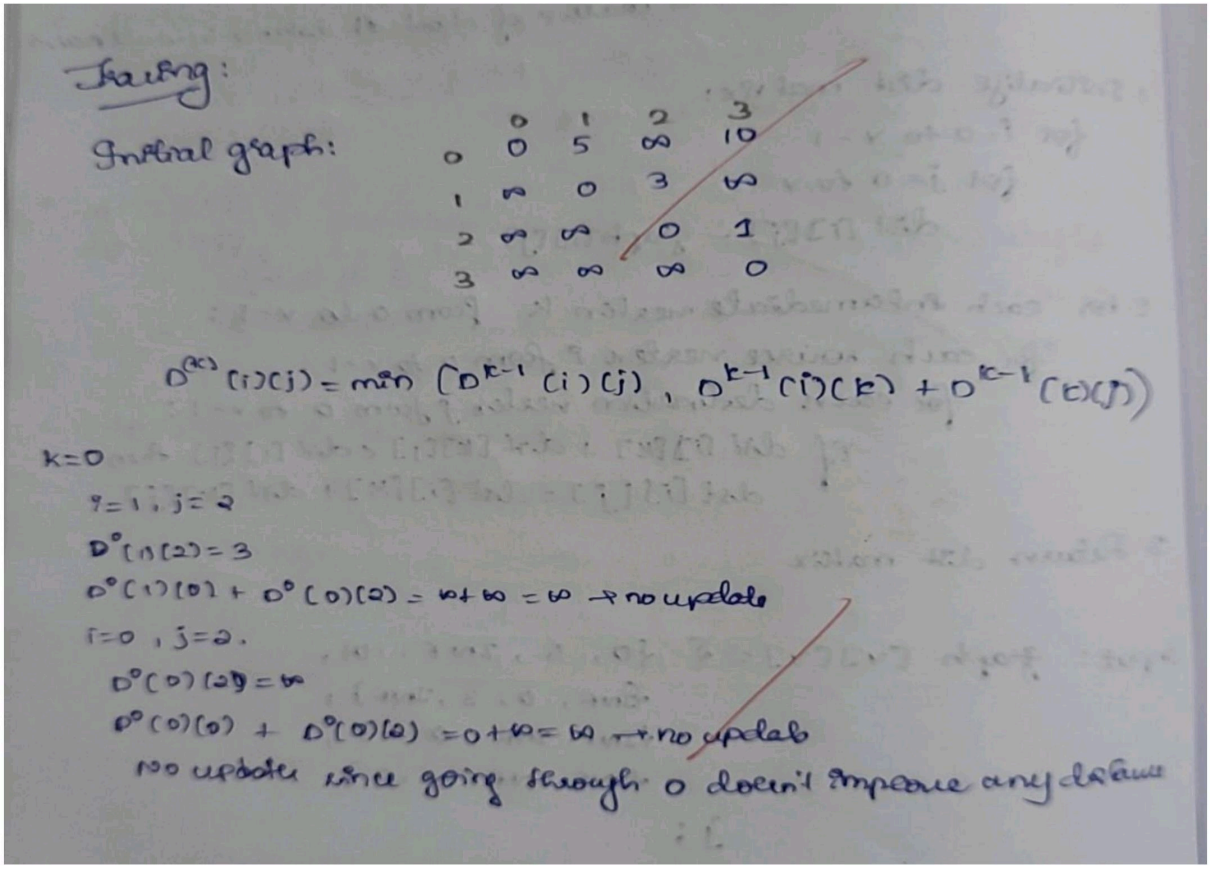
```
            printf("%7s", "INF");
        else
            printf("%7d", dist[i][j]);
    }
    printf("\n");
  }
}

// Floyd-Warshall algorithm
void floydWarshall(int graph[V][V]) {
  int dist[V][V];

  // Initialize the solution matrix with input graph
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      dist[i][j] = graph[i][j];

  // Update dist[i][j] to the shortest path
  for (int k = 0; k < V; k++) {
    for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
        if (dist[i][k] + dist[k][j] < dist[i][j])
          dist[i][j] = dist[i][k] + dist[k][j];
      }
    }
  }

  printSolution(dist);
}
```

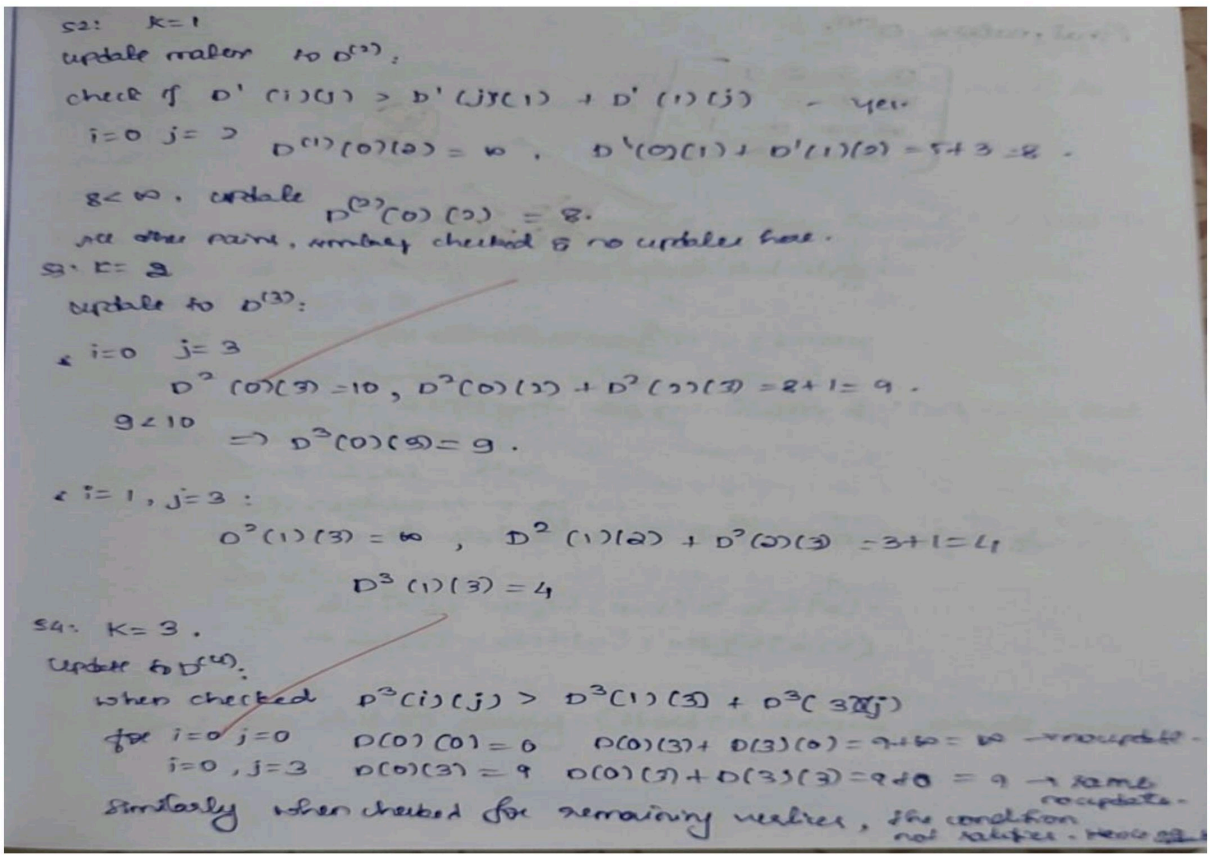
```
// Driver code
int main() {
    // Example graph with 4 vertices
    int graph[V][V] = {
        {0,   5,  INF, 10},
        {INF, 0,   3,  INF},
        {INF, INF, 0,   1},
        {INF, INF, INF, 0}
    };

    floydWarshall(graph);

    return 0;
}
```

OUTPUT:

```
Shortest distances between every pair of vertices:
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0
```

TRACING


Tracing:
Initial graph:
0 1 2 3
0 0 5 ∞ 10
1 ∞ 0 3 ∞
2 ∞ ∞ 0 1
3 ∞ ∞ ∞ 0
D^(k)[i][j] = min (D^(k-1)[i][j], D^(k-1)[i][k] + D^(k-1)[k][j])
k=0
i=1, j=2
D°[1][2] = 3
D°[1][0] + D°[0][2] = ∞+∞ = ∞ → no update
i=0, j=2.
D°[0][2] = ∞
D°[0][0] + D°[0][2] = 0+∞ = ∞ → no update
No update since going through 0 doesn't improve any distance


S2: k=1
update matrix to D(1):
check if D'[i][j] > D'[i][1] + D'[1][j] - yes
i=0 j=2 D(1)[0][2] = ∞, D'[0][1] + D'[1][2] = 5+3 = 8.
8 < ∞. update D(2)[0][2] = 8.
All other pairs, similarly checked & no updates done.
S3: k=2
update to D(3):
i=0 j=3
D²[0][3] = 10, D²[0][2] + D²[2][3] = 8+1 = 9.
9 < 10
⇒ D³[0][3] = 9.
i=1, j=3:
D²[1][3] = ∞, D²[1][2] + D²[2][3] = 3+1 = 4
D³[1][3] = 4
S4: k=3.
Update to D(4):
When checked D³[i][j] > D³[1][3] + D³[3][j]
for i=0 j=0 D[0][0] = 0 D[0][3] + D[3][0] = 9+∞ = ∞ → no update
i=0, j=3 D[0][3] = 9 D[0][3] + D[3][3] = 9+0 = 9 → same
Similarly when checked for remaining values, the condition not satisfies. No update.

Final matrix $D^{(4)}$:

$$\begin{bmatrix} 0 & 5 & 8 & 9 \\ \infty & 0 & 3 & 4 \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 0 \end{bmatrix}.$$

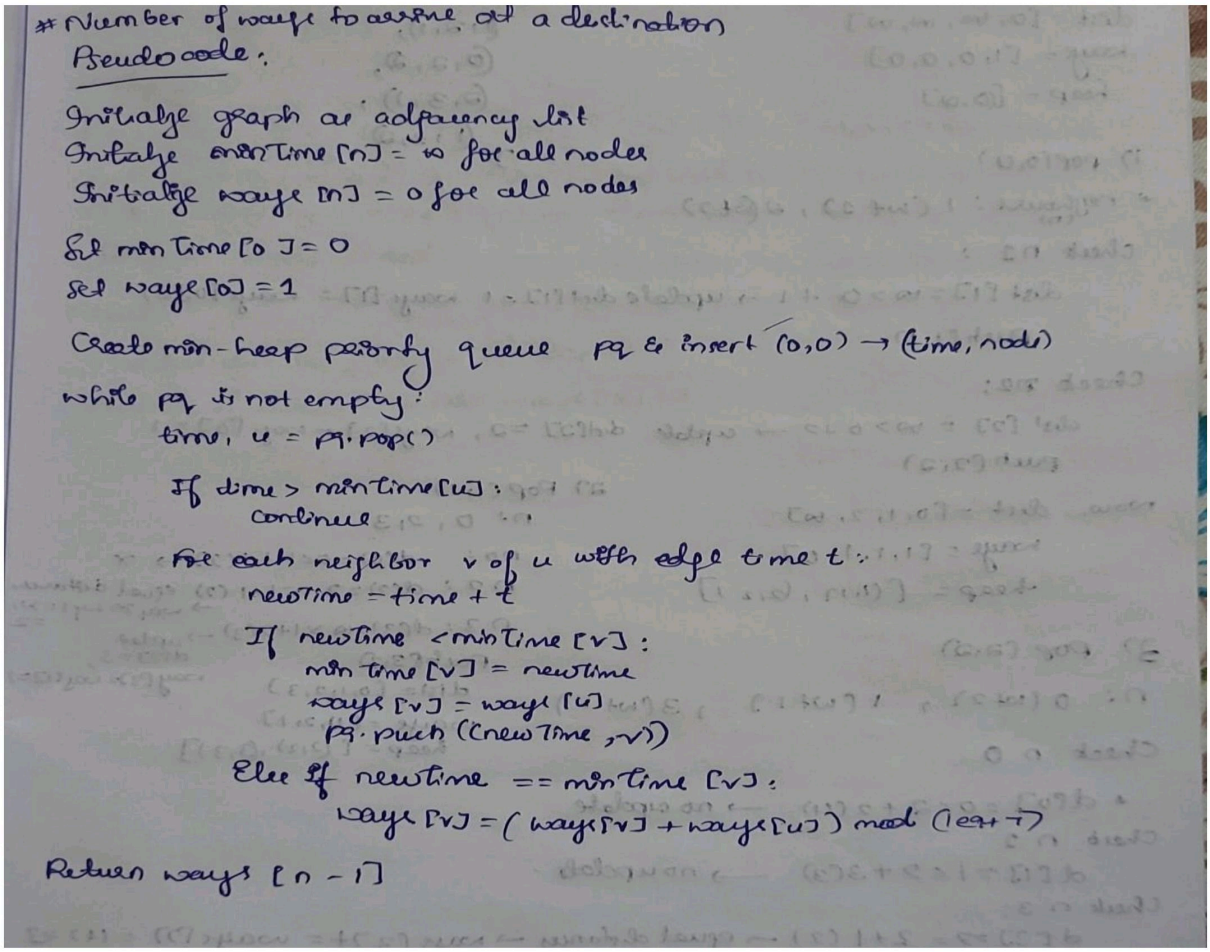
6/6

17/5/25

LEETCODE 5

NUMBER OF WAYS TO ARRIVE AT DESTINATION

ALGORITHM



CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MOD 1000000007

typedef struct {
    int to, weight;
} Edge;
```

```
typedef struct {
    Edge* edges;
    int size, capacity;
} AdjList;

typedef struct {
    long long time;
    int node;
} HeapNode;

typedef struct {
    HeapNode* arr;
    int size;
} MinHeap;

void swap(HeapNode* a, HeapNode* b) {
    HeapNode tmp = *a;
    *a = *b;
    *b = tmp;
}

void heapify_up(MinHeap* h, int i) {
    while (i > 0 && h->arr[i].time < h->arr[(i - 1) / 2].time) {
        swap(&h->arr[i], &h->arr[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

void heapify_down(MinHeap* h, int i) {
    int left, right, smallest;
```

```
    while (2 * i + 1 < h->size) {
        left = 2 * i + 1;
        right = 2 * i + 2;
        smallest = i;

        if (left < h->size && h->arr[left].time < h->arr[smallest].time)
            smallest = left;
        if (right < h->size && h->arr[right].time < h->arr[smallest].time)
            smallest = right;

        if (smallest == i) break;
        swap(&h->arr[i], &h->arr[smallest]);
        i = smallest;
    }
}

void push(MinHeap* h, long long time, int node) {
    h->arr[h->size++] = (HeapNode){time, node};
    heapify_up(h, h->size - 1);
}

HeapNode pop(MinHeap* h) {
    HeapNode top = h->arr[0];
    h->arr[0] = h->arr[--h->size];
    heapify_down(h, 0);
    return top;
}

int countPaths(int n, int** roads, int roadsSize, int* roadsColSize) {
    // Step 1: Build graph
```

```
AdjList* graph = malloc(n * sizeof(AdjList));
for (int i = 0; i < n; ++i) {
    graph[i].edges = malloc(100 * sizeof(Edge));
    graph[i].size = 0;
    graph[i].capacity = 100;
}

for (int i = 0; i < roadsSize; ++i) {
    int u = roads[i][0], v = roads[i][1], w = roads[i][2];
    graph[u].edges[graph[u].size++] = (Edge){v, w};
    graph[v].edges[graph[v].size++] = (Edge){u, w};
}

// Step 2: Initialize minTime and ways
long long* minTime = malloc(n * sizeof(long long));
int* ways = malloc(n * sizeof(int));
for (int i = 0; i < n; ++i) {
    minTime[i] = LLONG_MAX;
    ways[i] = 0;
}
minTime[0] = 0;
ways[0] = 1;

// Step 3: Min-heap priority queue
MinHeap heap;
heap.arr = malloc(100000 * sizeof(HeapNode));
heap.size = 0;
push(&heap, 0, 0); // (time, node)

while (heap.size > 0) {
```

```
    HeapNode curr = pop(&heap);
    long long time = curr.time;
    int u = curr.node;

    if (time > minTime[u]) continue;

    for (int i = 0; i < graph[u].size; ++i) {
        int v = graph[u].edges[i].to;
        int t = graph[u].edges[i].weight;
        long long newTime = time + t;

        if (newTime < minTime[v]) {
            minTime[v] = newTime;
            ways[v] = ways[u];
            push(&heap, newTime, v);
        } else if (newTime == minTime[v]) {
            ways[v] = (ways[v] + ways[u]) % MOD;
        }
    }
}

int result = ways[n - 1];

// Clean up
for (int i = 0; i < n; ++i) free(graph[i].edges);
free(graph);
free(minTime);
free(ways);
free(heap.arr);
```

```
    return result;
}
```

OUTPUT

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

n =
7

roads =
[[0,6,7],[0,1,2],[1,2,3],[1,3,3],[6,3
,1],[6,5,1],[2,5,1],[0,4,5],[4,6,2]]

Output

4

Expected

4

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

n =
2

roads =
[[1,0,10]]

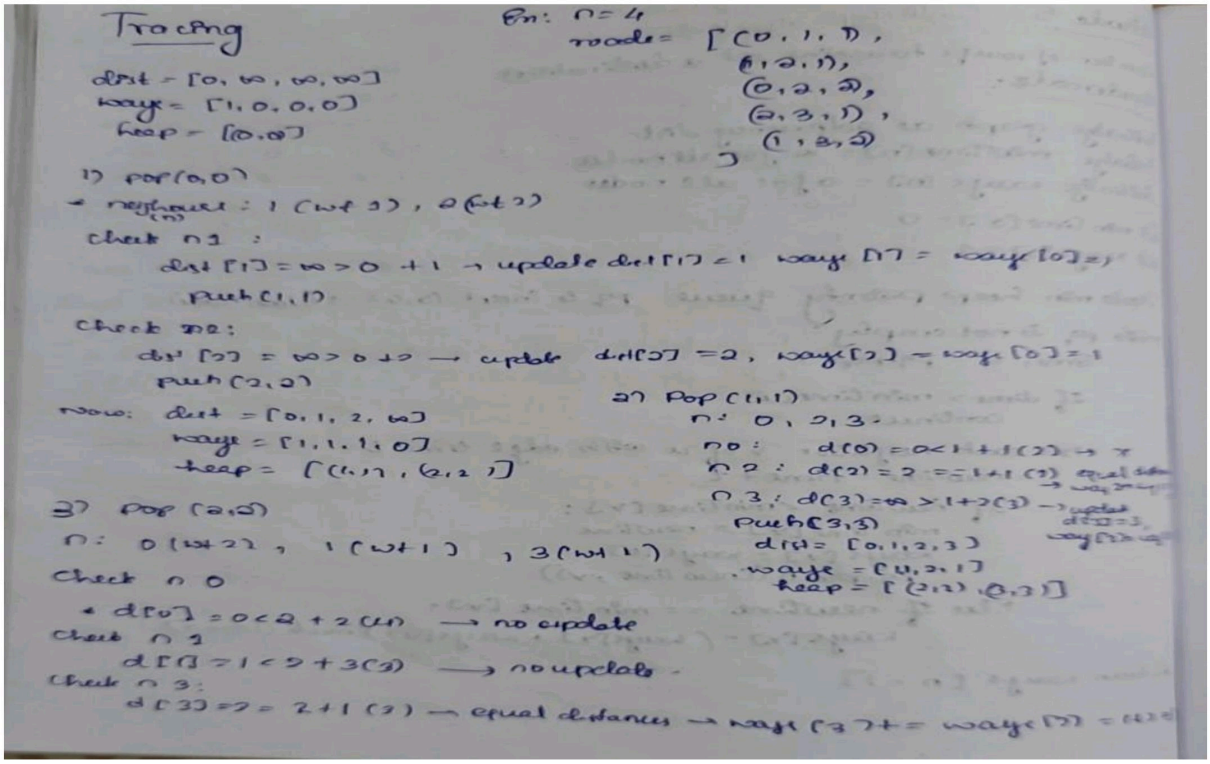
Output

1

Expected

1

TRACING



Now: dist = [0, 1, 2, 3]

ways = [1, 1, 2, 3]

heap = (3, 3)

4) Pop (3, 3)

* n: 2 (wt 1), 1 (wt 5)

n 2: d[2] = 2 < 3+1 (4) → no update

n 1: d[1] = 1 < 3+2 (5) → no update

Heap → empty → end.

Result:

Shortest distance from 0 to 3 is 3

Number of shortest paths ways[3] = 3