```python
# Write a program that, given a graph with costs and two vertices, finds a lowest cost walk between the given vertices,
# or prints a message if there are negative cost cycles in the graph. The program shall use the matrix multiplication algorithm.

INF = float('inf')
intermediate_matrices = []

# initializes the matrix of the graph with the costs and a matrix of predecessors so that the actual lowest cost walk can be computed
def init_graph_matrices(file_name:str) -> tuple:
    with open(file_name, "r") as file:
        n = file.readline().removesuffix('\n')
        n = int(n)
        mat = [[INF for _ in range (n)] for _ in range(n)] # INF meaning as for now we dont know if there is a walk from any pair of vertices
        pred = [[None for _ in range (n)] for _ in range(n)] # None meaning as for now, any pair of vertices has no "ideal" predecessor

        for i in range(n):
            mat[i][i] = 0 # we can get from vertex i to itself with cost 0
            pred[i][i] = i

        lines = file.readlines()
        for line in lines:
            tokens = line.strip().split()
            mat[int(tokens[0])][int(tokens[1])] = int(tokens[2])
            pred[int(tokens[0])][int(tokens[1])] = int(tokens[0]) # direct edge from i to j means pred[i][j] = j

    return (mat, pred)

# saves the intermediate matrices to a file
def save_intermediate_matrices(file_name:str):
    with open(file_name, "w") as file:
        n = len(intermediate_matrices)
        m = len(intermediate_matrices[0])

        for i in range(n):
            str_matrix = ""
            for j in range(m):
                str_matrix += str(intermediate_matrices[i][j]) + '\n'

            str_matrix += '\n'
            file.writelines(str_matrix)

# computes and returns the lowest cost between 2 vertices
def lowest_cost_walk(mat:list, pred:list, v1:int, v2:int) -> int:
    apsp_matrix = all_pairs_shortest_path(mat, pred) # construct the matrix
    return apsp_matrix[v1][v2]

# actual algorithm to compute the lowest cost between any pair of vertices
def all_pairs_shortest_path(mat:list, pred:list) -> list:
    n = len(mat)
    weights = [row[:] for row in mat]
    for _ in range(1, n):
        intermediate_matrices.append([row[:] for row in mat]) # make a copy of the current matrix and append it to the list of intermediate matrices
        mat = multiply_matrices(mat, weights, pred, n) # basically mat^n = mat^(n - 1) * mat; pred is the matrix of predecessors that will potentially change

    if not stable_matrix(mat):
        raise Exception("The graph has negative cycles!\n")

    return mat

# matrix multiplication algorithm
def multiply_matrices(mat1:list, mat2:list, pred:list, n:int) -> list:
    mat = [[INF for _ in range (n)] for _ in range(n)]

    for i in range(n):
        for j in range(n):
            for k in range(n):
                if mat1[i][k] != INF and mat2[k][j] != INF and mat[i][j] > mat1[i][k] + mat2[k][j]: # if we found a better path from i to j (from i to k and th
                    mat[i][j] = mat1[i][k] + mat2[k][j] # update the cost of the walk
                    pred[i][j] = pred[k][j] # update the predecessor matrix

    return mat

# checks if the graph matrix has any negative cycles
def stable_matrix(mat:list) -> bool:
    n = len(mat)
    for i in range(n):
        if mat[i][i] < 0: # if a walk from vertex i to itself has a negative cost => negative cycle => matrix is not stable
            return False
    return True

# finds the lowest cost path between 2 vertices
def find_min_path(v1:int, v2:int, pred:list) -> list:
    if pred[v1][v2] == None:
        return []

    path = []
    while v1 != v2:
        path.append(v2)
        v2 = pred[v1][v2]

    path.append(v1)
    path.reverse()
    return path

# used to solve the entire problem with the functions above
def solve():
    file_name = "graph2_pb6.txt"
    output_file = "intermediate_matrices.txt"
    graph_matrix, pred = init_graph_matrices(file_name)

    try:
        v1, v2 = tuple(input("Insert both nodes here: ").split())
        v1 = int(v1)
        v2 = int(v2)
        lcw = lowest_cost_walk(graph_matrix, pred, v1, v2)
        path = find_min_path(v1, v2, pred)
        if len(path) == 0:
            print("No path has been found.\n")
        else:
            print(f"The path is: {path}.")
            print(f"The lowest cost walk from {v1} to {v2}: {lcw}.\n")
        save_intermediate_matrices(output_file)
    except Exception as e:
        print(e)
```