Tricky Elevator Problem: A Graph Theory Approach

Problem Description

Consider a building with N floors (numbered $1, 2, \ldots, N$) and an elevator with two buttons:

- **Up by** U: Moves the elevator up U floors (if possible).
- **Down by** D: Moves the elevator down D floors (if possible).

Starting at floor X, the objective is to reach floor Y using the minimum number of button presses. If the destination is unreachable, the output should be "TAKE THE STAIRS!".

Graph Theoretic Model

We model the elevator system as a directed, unweighted graph $G = (V, E)$ where:

- **Vertices (V)**: Each floor i such that $1 \leq i \leq N$ is a vertex.
- **Edges (E)**: For each floor i:
  - There is an edge from i to $i+U$ if $i+U \leq N$ (corresponding to the "Up" button).
  - There is an edge from i to $i-D$ if $i-D \geq 1$ (corresponding to the "Down" button).

This abstraction allows us to interpret the problem as finding the shortest path from vertex X to vertex Y.

Breadth-First Search (BFS)

Since the graph is unweighted, the optimal solution is obtained by finding the shortest path via Breadth-First Search (BFS). As presented in the lecture, BFS is implemented by:

- Marking each vertex with a color: white (unvisited), gray (discovered), or black (fully explored).
- Maintaining a distance array $d[v]$ which stores the number of moves from the source X to vertex v.

- Using a predecessor array $\pi[v]$ to reconstruct the shortest path, if needed.

The lecture provided the following pseudocode for BFS:

```
BFS(G=(V,E), s):
   for all v in V:
      colour[v] ← white
      d[v] \leftarrow \infty
      \pi[v] \leftarrow NIL
   colour[s] ← gray
   d[s] ← 0
   Enqueue(Q, s)
   while Q is not empty:
      u ← Dequeue(Q)
      for each neighbour v of u:
         if colour[v] = white:
            colour[v] ← gray
            d[v] ← d[u] + 1
            \(\pi\)[v] ← u
            Enqueue(Q, v)
      colour[u] ← black
```

Application to the Elevator Problem

In the context of our elevator problem:

- The source vertex is $s = X$ and the target is Y.
- The neighbors of a vertex (floor) i are:
  - $i + U$ (if $i + U \leq N$)
  - $i - D$ (if $i - D \geq 1$)

By applying BFS starting from X, the first time we encounter Y the value $d[Y]$ is the minimum number of button presses required. If Y remains unreachable (i.e., $d[Y] = \infty$), then the solution is to "TAKE THE STAIRS!".

Python Implementation

Below is a Python implementation of the solution using the BFS approach inspired by the lecture. The function `elevator` takes the number of floors N, the starting floor X, the destination floor Y, and the values of U and D as input. It returns the minimum number of button presses required to reach the destination floor.

```
from collections import deque

def elevator(N, X, Y, U, D):
    # If the starting floor is the destination.
    if X == Y:
        return 0

    # Initialize "colour" (visited), distance and queue.
    # In our implementation, 'visited' acts as the colour marker.
    visited = [False] * (N + 1)
    distance = [float('inf')] * (N + 1)

    queue = deque([X])
    visited[X] = True
    distance[X] = 0

    while queue:
        current = queue.popleft()

        # Explore the "Up" neighbour.
        next_up = current + U
        if next_up <= N and not visited[next_up]:
            visited[next_up] = True
            distance[next_up] = distance[current] + 1
            if next_up == Y:
                return distance[next_up]
            queue.append(next_up)

```

```
        # Explore the "Down" neighbour.
        next_down = current - D
        if next_down >= 1 and not visited[next_down]:
            visited[next_down] = True
            distance[next_down] = distance[current] + 1
            if next_down == Y:
                return distance[next_down]
            queue.append(next_down)

    return "TAKE THE STAIRS!"

# Sample test cases:
print(elevator(10, 1, 10, 2, 1))  # Expected output: 6
print(elevator(10, 2, 3, 2, 2))   # Expected output: TAKE THE STAIRS
    !
```

Extension: Reconstructing the Sequence of Moves

If required to output the actual sequence of button presses, we can modify BFS to store the predecessor (using the π array from the lecture) and then reconstruct the path:

```
def elevator_path(N, X, Y, U, D):
    if X == Y:
        return ["Already at destination."]

    visited = [False] * (N + 1)
    distance = [float('inf')] * (N + 1)
    predecessor = [None] * (N + 1)

    queue = deque([X])
    visited[X] = True
    distance[X] = 0

    while queue:
        current = queue.popleft()
```

```
15
16          next_up = current + U
17          if next_up <= N and not visited[next_up]:
18              visited[next_up] = True
19              distance[next_up] = distance[current] + 1
20              predecessor[next_up] = current
21              if next_up == Y:
22                  break
23              queue.append(next_up)
24
25          next_down = current - D
26          if next_down >= 1 and not visited[next_down]:
27              visited[next_down] = True
28              distance[next_down] = distance[current] + 1
29              predecessor[next_down] = current
30              if next_down == Y:
31                  break
32              queue.append(next_down)
33
34      if not visited[Y]:
35          return "TAKE THE STAIRS!"
36
37      # Reconstruct the path from Y back to X.
38      path = []
39      node = Y
40      while node is not None:
41          path.append(node)
42          node = predecessor[node]
43      path.reverse()
44      return path
45
46  # Example usage:
47  print(elevator_path(10, 1, 10, 2, 1))
```

Conclusion

By modeling the elevator system as a graph with floors as vertices and valid moves as edges, and applying BFS (as detailed in the lecture, we can efficiently determine the minimum number of button presses required to reach the target floor. Moreover, by recording predecessors, we can reconstruct the sequence of moves, thus providing a comprehensive graph theory solution to the Tricky Elevator problem.