From X to Y

For this problem, the algorithm uses a Breadth-First Search (BFS) strategy, adept at handling transformations as node explorations linked by operations and Dynamic Programming (DP) concepts to memorize already visited values to optimize the algoritm. This BFS utilizes a queue to manage current possible states and a tree to document all transformations and reconstruct the transformation sequence. Starting with X, operations are performed and new states are enqueued if not previously visited. This ensures the shortest sequence of transformations, exploring all options at one level before advancing. Originally, a bit vector was considered for tracking visited, but was dismissed due to inefficiency with unbounded values from top, making a set a more suitable choice.

1. Data Structures

1.1 Queue

- **Purpose**: To current possible states of transformed X
- **Implementation**: A class utilizing Node-List to perform queue with O(1)

```
class QNode:
    def __init__(self, value):
        self.value = value
        self.next = None

class Queue:
    def __init__(self):
        self.front = None
        self.rear = None

    def is_empty(self):
        return self.front is None

    def enqueue(self, value):
        new_node = QNode(value)
        if self.rear is None:
            self.front = self.rear = new_node
            return
        self.rear.next = new_node
        self.rear = new_node

    def dequeue(self):
        if self.is_empty():
            raise Exception("Trying to dequeue from an empty queue")
        temp = self.front
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        return temp.value

    def peek(self):
        if self.is_empty():
            raise Exception("Trying to peek from an empty queue")
        return self.front.value
```

1.2 Tree

- **Purpose**: To keep track of all possible transformations of X and recreate the shortest sequence at the end
- **Implementation**: Ordinary Tree with node containing the number of steps, parent node, and actual value.

```
class TNode:
    def __init__(self, parent, value, prev_op):
        self.parent = parent
        self.value = value
        self.prev_op = prev_op
```

1.3 Bit Vector

- **Purpose**: To keep track of visited states during tree exploration.
- **Implementation**: A bit vector is a compact data structure that uses a series of bits to represent the visited status of states. Each bit corresponds to a potential state, where a set bit (1) indicates a visited state and an unset bit (0) indicates an unvisited state.

```
class BitVector:
    def __init__(self, size):
        self.size = size
        self.bits = [0] * ((size + 7) // 8)

    def set(self, index):
        byte_index = index // 8
        bit_index = index % 8
        self.bits[byte_index] |= (1 << bit_index)

    def check(self, index):
        byte_index = index // 8
        bit_index = index % 8
        return (self.bits[byte_index] & (1 << bit_index)) != 0
```

2. Algorithm

2.1 Initialization

- **Starting Value:** Create Node(steps = 0, parent = None, value = X)
- **Add this node to Queue**

```
results = BitVector(1000 * X * m)

operations = [lambda x: op_a(x, Y, m), op_b, op_c]

q = Queue()

start = TNode(None, X, None)

q.enqueue(start)

results.set(start.value)
```

2.2 Processing Loop

- **Loop**: Continues while Queue is not empty
  - **Dequeue**: Remove the front node **parent** from the queue to process it.
  - **Perform Operations**: calculate all 3 possible instructions

$$Res1 = \min(X * m, Y * 2)$$
$$Res2 = X - 2$$
$$Res3 = X - 1$$

    - **For each result (res)**:
      - **Unique Check:** If the result is not in **results** vector:
        - add res to **results** vector
        - create Node(steps = parent.steps + 1, parent = parent, value = res)
        - add Node to queue for further calculations
      - **Exit Check**: If res == Y, return Created Node

```
    if start.value == Y:
        return start
    else:
        while not q.is_empty():
            cur = q.dequeue()

            for op in operations:
                res, name = op(cur.value)
                if res == Y:
                    return TNode(cur, res, name)
                if not results.check(res):
                    results.set(res)
                    q.enqueue(TNode(cur, res, name))
    return -1
```

Recreate sequence: Move all the way up to the root using Node parent, adding value and operation details to list for each iteration to recreate the sequence.

```
def build_path(end_node):
    path = []
    current_node = end_node
    while current_node is not None:
        if current_node.prev_op is not None:
            path.append((current_node.value, current_node.prev_op))
        else:
            path.append((current_node.value, 'Start'))
        current_node = current_node.parent

    path.reverse()
    return path
```

3. Complexity Analysis

- 3.1 **Time Complexity**: The complexity of the BFS-based approach for transforming a value *X* into *Y* is challenging to pinpoint due to several variables:
- **Effect of Operations**: The transformations applied to *X* can produce a wide or potentially unbounded range of outcomes, complicating the prediction of the total number of unique states.
- **Branching Factor**: Each state can generate three new states, but actual outcomes depend heavily on the specifics of the operations and constraints of the values.
- **Unknown Bounds**: Without specific values for *m* and *Y* and clear constraints on transformations, estimating the total number of states and the overall computational load is uncertain.

4. Conclusion

This BFS-based algorithm efficiently determines the shortest sequence of transformations from *X* to *Y*, if such a sequence exists. By systematically exploring each transformation and its implications before advancing deeper, the algorithm guarantees that the shortest path is found by minimizing the number of steps required. The use of a queue ensures that earlier discovered states are processed before newer ones, maintaining the BFS's characteristic of level-by-level exploration. The use of a set for tracking visited states, instead of a bit vector, allows the algorithm to efficiently handle a potentially unlimited range of transformations, which is crucial given the problem's constraints. This approach not only ensures the optimality of the solution but also maintains a manageable computational load, making it suitable for a broad range of practical applications.