```
# You are using Python
given N*N
from collections import deque

def shortest_clear_path(grid):
    n = len(grid)
    if grid[0][0] == 1 or grid[n-1][n-1] == 1:
        return -1  # No path if starting or ending cell is blocked

    # Directions for 8-connectivity
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1),
                  (-1, -1), (-1, 1), (1, -1), (1, 1)]

    # BFS setup
    queue = deque([(0, 0, 1)])  # (x, y, path_length)
    visited = set((0, 0))

    while queue:
        x, y, length = queue.popleft()

        # If we reach the bottom-right corner
        if x == n - 1 and y == n - 1:
            return length

        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            # Check if the new position is within bounds and not visited
            if 0 <= nx < n and 0 <= ny < n and (nx, ny) not in visited and grid[nx][ny] == 0:
                visited.add((nx, ny))
                queue.append((nx, ny, length + 1))
```

```
    return -1  # No path found

# Input reading and processing
n = int(input().strip())
grid = [list(map(int, input().strip().split())) for _ in range(n)]

# Output the result
print(shortest_clear_path(grid))

## rat
def find_paths(maze, N):
    def dfs(x, y, path):
        # If we reach the destination
        if x == N - 1 and y == N - 1:
            paths.append(path)
            return

        # Directions: down, left, right, up
        directions = [(1, 0, 'D'), (0, -1, 'L'), (0, 1, 'R'), (-1, 0, 'U')]

        for dx, dy, direction in directions:
            nx, ny = x + dx, y + dy
            # Check if the new cell is within bounds and not visited
            if 0 <= nx < N and 0 <= ny < N and maze[nx][ny] == 1 and not visited[nx][ny]:
                visited[nx][ny] = True  # Mark the cell as visited
                dfs(nx, ny, path + direction)  # Recur with the new path
                visited[nx][ny] = False  # Backtrack and unmark the cell

    # Initial checks for the start and end cells
    if maze[0][0] == 0 or maze[N - 1][N - 1] == 0:
```

```
        return ["-1"]  # If start or end is blocked

    paths = []
    visited = [[False] * N for _ in range(N)]
    visited[0][0] = True  # Mark the starting cell as visited
    dfs(0, 0, "")  # Start DFS from the top-left corner

    if not paths:
        return ["-1"]  # If no paths were found

    paths.sort()  # Sort paths lexicographically
    return paths

# Reading input
N = int(input())
maze = [list(map(int, input().split())) for _ in range(N)]

# Finding paths
result = find_paths(maze, N)

# Printing result
print(" ".join(result))
##grid
def findLongestPath(grid, M, N):
    def dfs(x, y, start_char, visited):
        max_length = 0
        # Define possible movements: right, down, left, up
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        for dx, dy in directions:
            nx, ny = x + dx, y + dy
```

```
            # Check boundaries and if the next node is unvisited and of different type
            if 0 <= nx < M and 0 <= ny < N and (nx, ny) not in visited and grid[nx][ny] != start_char:
                visited.add((nx, ny))  # Mark as visited
                # Recursively explore the next node
                length = dfs(nx, ny, grid[nx][ny], visited)  # Change start_char to current char
                max_length = max(max_length, length)
                visited.remove((nx, ny))  # Backtrack

        return max_length + 1  # Include the current node

    longest_path = 0
    for i in range(M):
        for j in range(N):
            start_char = grid[i][j]
            visited = {(i, j)}  # Start with the current node
            # Explore paths from this node
            longest_path = max(longest_path, dfs(i, j, start_char, visited))

    return longest_path

# Reading input
M, N = map(int, input().split())
grid = [input().strip() for _ in range(M)]

# Finding the longest path
result = findLongestPath(grid, M, N)

# Printing the result
print(result)
##Shakshi
  class TreeNode:
```

```
    def _init_(self, value):
        self.value = value
        self.left = None
        self.right = None

class BinaryTree:
    def _init_(self):
        self.root = None

    def insert_level_order(self, values):
        if not values:
            return

        self.root = TreeNode(values[0])
        queue = [self.root]
        index = 1

        while index < len(values):
            current = queue.pop(0)

            if index < len(values):
                current.left = TreeNode(values[index])
                queue.append(current.left)
                index += 1

            if index < len(values):
                current.right = TreeNode(values[index])
                queue.append(current.right)
                index += 1

    def post_order_cubed(self):
```

```
        result = []
        self._post_order_cubed_rec(self.root, result)
        return result

    def _post_order_cubed_rec(self, node, result):
        if node:
            self._post_order_cubed_rec(node.left, result)
            self._post_order_cubed_rec(node.right, result)
            result.append(node.value ** 3)

def main():
    n = int(input().strip())
    values = list(map(int, input().strip().split()))

    # Construct binary tree
    binary_tree = BinaryTree()
    binary_tree.insert_level_order(values)

    # Get post-order traversal with cubed values
    post_order_result = binary_tree.post_order_cubed()

    # Print the result
    print(" ".join(map(str, post_order_result)))

if _name_ == "_main_":
    main()
##you r
def minimax(scores, depth, is_maximizing):
    # Base case: If we are at the leaf level
    if depth == 0:
        return scores[0]
```

```
    # Calculate the number of scores at the current depth
    num_scores = 2 ** depth
    left_scores = scores[:num_scores // 2]
    right_scores = scores[num_scores // 2:num_scores]

    if is_maximizing:
        # Maximizing player's turn
        left_value = minimax(left_scores, depth - 1, False)
        right_value = minimax(right_scores, depth - 1, False)
        return max(left_value, right_value)
    else:
        # Minimizing player's turn
        left_value = minimax(left_scores, depth - 1, True)
        right_value = minimax(right_scores, depth - 1, True)
        return min(left_value, right_value)

def optimal_value(scores):
    # Calculate the depth based on the number of scores
    n = len(scores)
    depth = n.bit_length() - 1  # log2(n)
    return minimax(scores, depth, True)

if _name_ == "_main_":
    n = int(input())
    scores = list(map(int, input().split()))
    result = optimal_value(scores)
    print(result)
##alex
def minimax_with_alpha_beta(values, depth, is_maximizing, alpha, beta):
    # Base case: if at leaf node, return the value
```

```
    if depth == 0:
        return values[0]

    num_values = len(values)

    # Split values into left and right children
    left_values = values[:num_values // 2]
    right_values = values[num_values // 2:num_values:]

    if is_maximizing:
        max_value = float('-inf')
        for child in [left_values, right_values]:
            value = minimax_with_alpha_beta(child, depth - 1, False, alpha, beta)
            max_value = max(max_value, value)
            alpha = max(alpha, max_value)
            if beta <= alpha:
                break  # Beta cut-off
        return max_value
    else:
        min_value = float('inf')
        for child in [left_values, right_values]:
            value = minimax_with_alpha_beta(child, depth - 1, True, alpha, beta)
            min_value = min(min_value, value)
            beta = min(beta, min_value)
            if beta <= alpha:
                break  # Alpha cut-off
        return min_value

def optimal_supply_allocation(depth, leaf_values):
    # Compute the optimal value using Minimax with Alpha-Beta pruning
    optimal_value = minimax_with_alpha_beta(leaf_values, depth, True, float('-inf'), float('inf'))
```

```
    # Add reserve
    value_with_reserve = optimal_value + 10
    return optimal_value, value_with_reserve

def main():
    d = int(input())
    leaf_values = list(map(int, input().split()))
    optimal_value, value_with_reserve = optimal_supply_allocation(d, leaf_values)

    # Print the output in the required format
    print(f"Optimal value: {optimal_value}\nValue with 10 units reserve: {value_with_reserve}")

if _name_ == "_main_":
    main()s
```