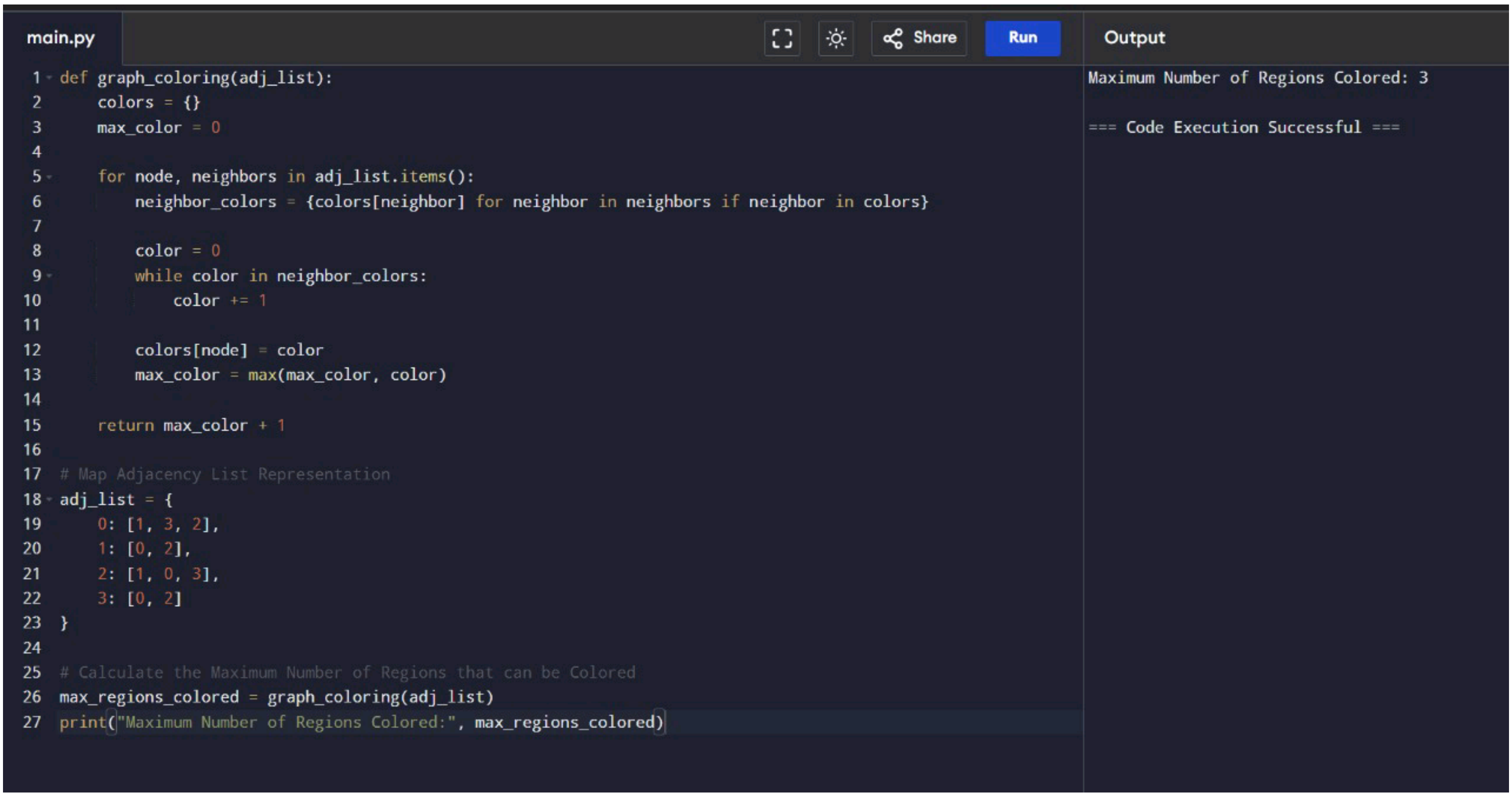
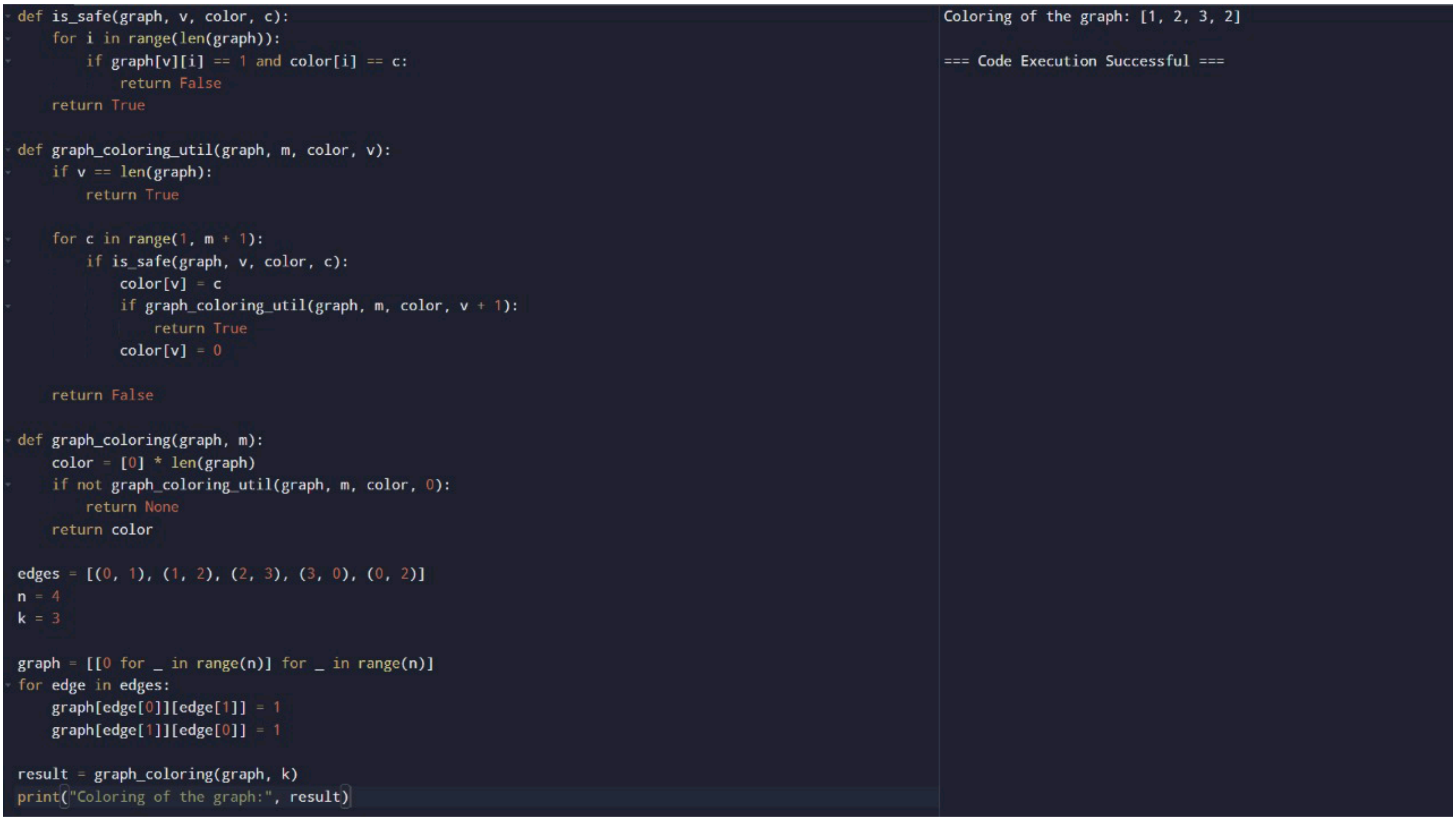


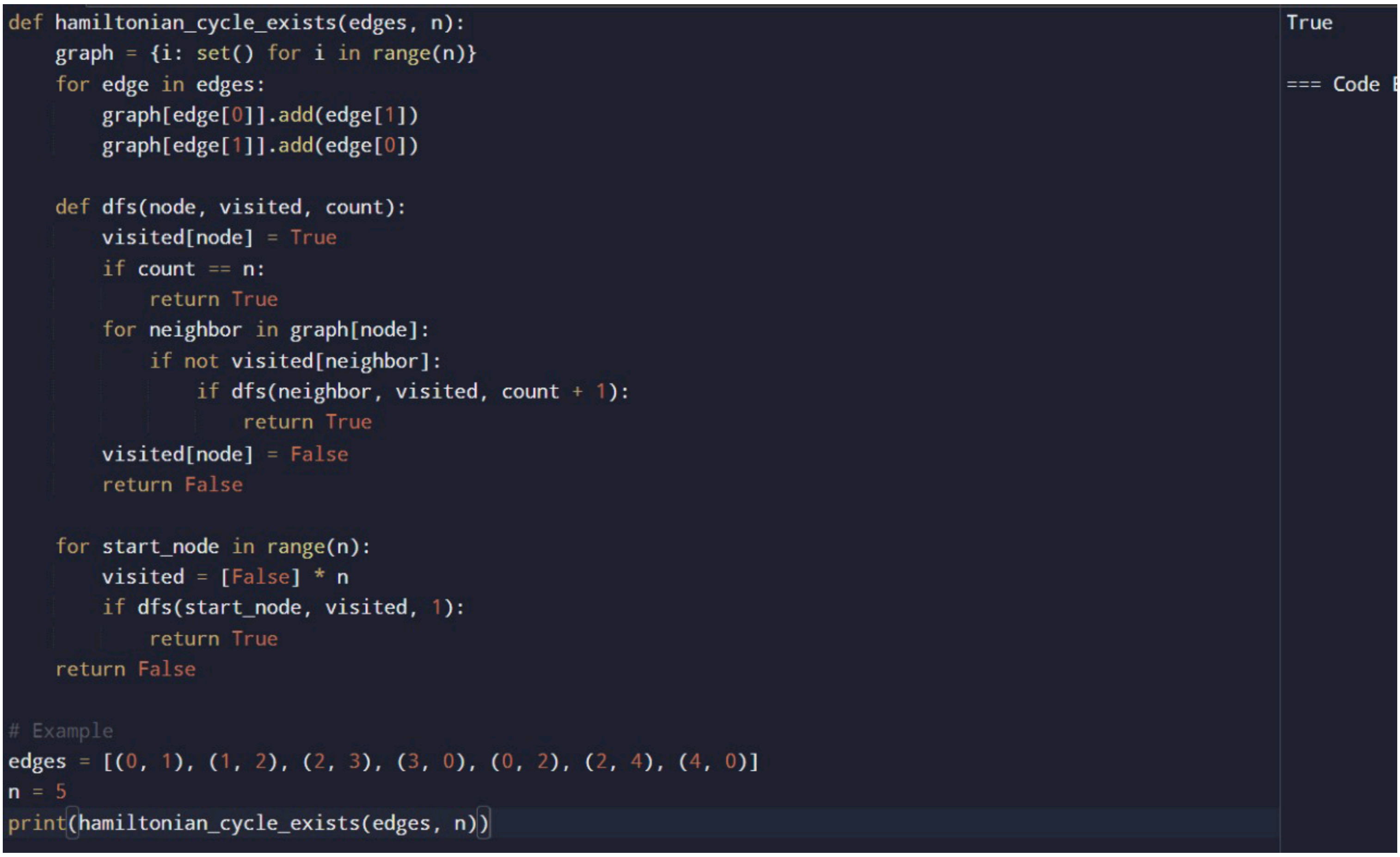
```python
def graph_coloring(adj_list):
    colors = {}
    max_color = 0

    for node, neighbors in adj_list.items():
        neighbor_colors = {colors[neighbor] for neighbor in neighbors if neighbor in colors}

        color = 0
        while color in neighbor_colors:
            color += 1

        colors[node] = color
        max_color = max(max_color, color)

    return max_color + 1

# Map Adjacency List Representation
adj_list = {
    0: [1, 3, 2],
    1: [0, 2],
    2: [1, 0, 3],
    3: [0, 2]
}

# Calculate the Maximum Number of Regions that can be Colored
max_regions_colored = graph_coloring(adj_list)
print("Maximum Number of Regions Colored:", max_regions_colored)
```

```
Maximum Number of Regions Colored: 3

=== Code Execution Successful ===
```



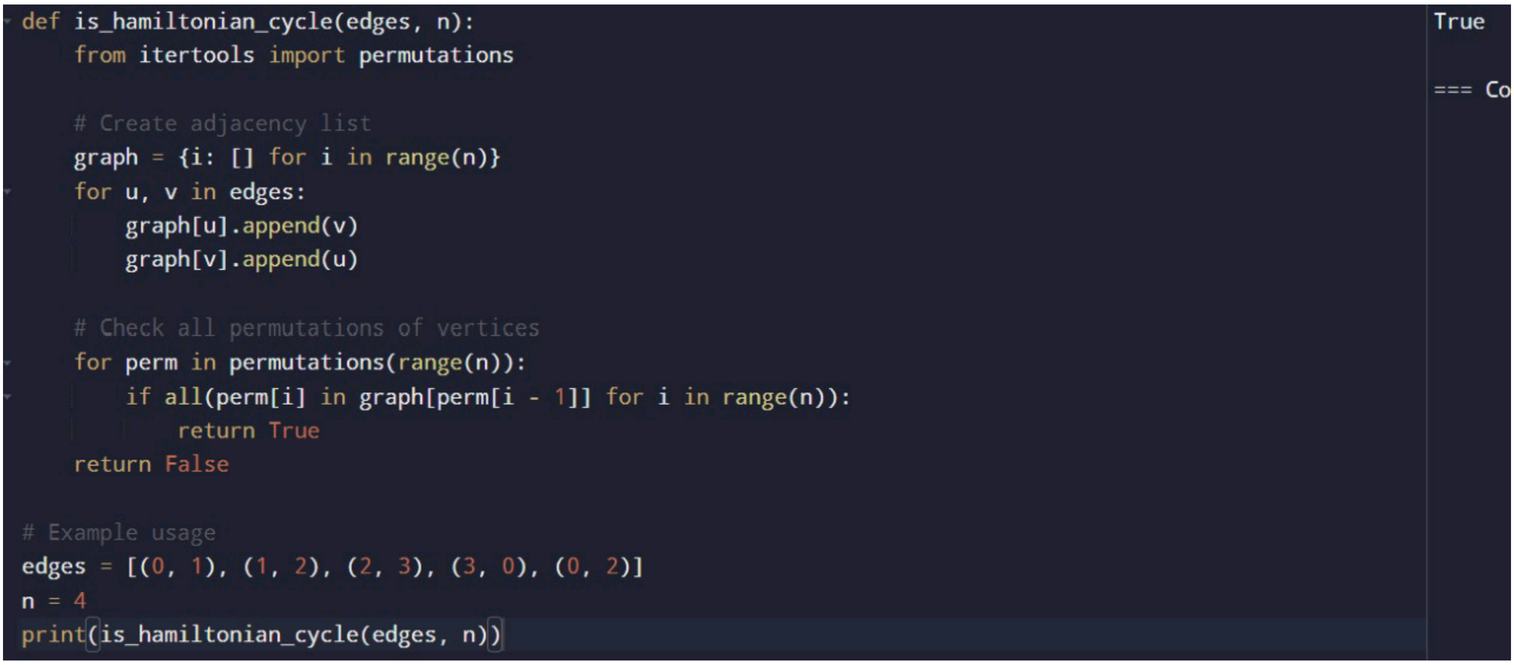
```python
def is_safe(graph, v, color, c):
    for i in range(len(graph)):
        if graph[v][i] == 1 and color[i] == c:
            return False
    return True

def graph_coloring_util(graph, m, color, v):
    if v == len(graph):
        return True

    for c in range(1, m + 1):
        if is_safe(graph, v, color, c):
            color[v] = c
            if graph_coloring_util(graph, m, color, v + 1):
                return True
            color[v] = 0

    return False

def graph_coloring(graph, m):
    color = [0] * len(graph)
    if not graph_coloring_util(graph, m, color, 0):
        return None
    return color

edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4
k = 3

graph = [[0 for _ in range(n)] for _ in range(n)]
for edge in edges:
    graph[edge[0]][edge[1]] = 1
    graph[edge[1]][edge[0]] = 1

result = graph_coloring(graph, k)
print("Coloring of the graph:", result)
```

```
Coloring of the graph: [1, 2, 3, 2]

=== Code Execution Successful ===
```



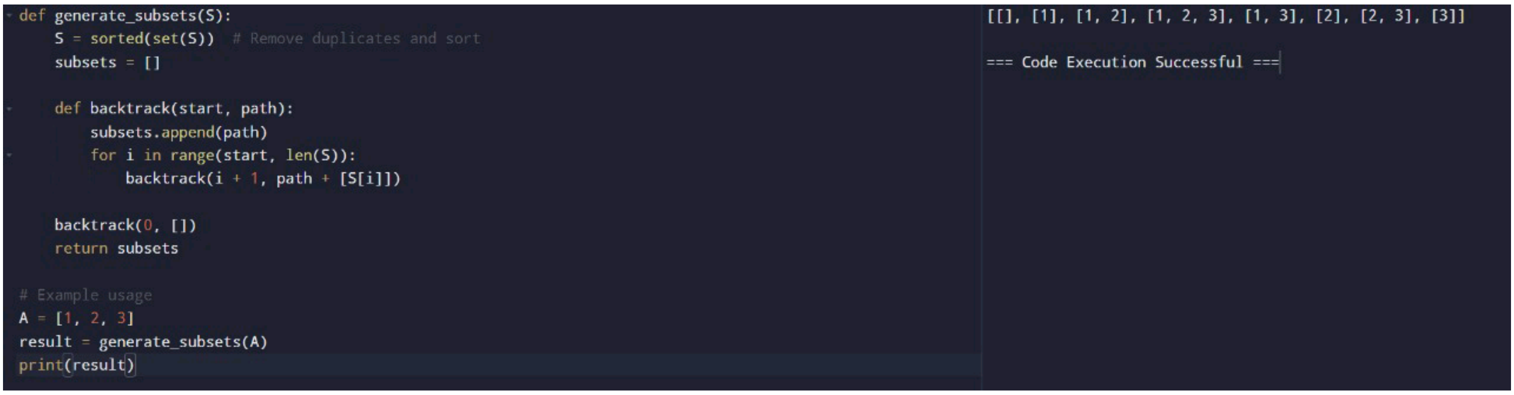
```python
def hamiltonian_cycle_exists(edges, n):
    graph = {i: set() for i in range(n)}
    for edge in edges:
        graph[edge[0]].add(edge[1])
        graph[edge[1]].add(edge[0])

    def dfs(node, visited, count):
        visited[node] = True
        if count == n:
            return True
        for neighbor in graph[node]:
            if not visited[neighbor]:
                if dfs(neighbor, visited, count + 1):
                    return True
        visited[node] = False
        return False

    for start_node in range(n):
        visited = [False] * n
        if dfs(start_node, visited, 1):
            return True
    return False

# Example
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)]
n = 5
print(hamiltonian_cycle_exists(edges, n))
```

```
True

=== Code E
```



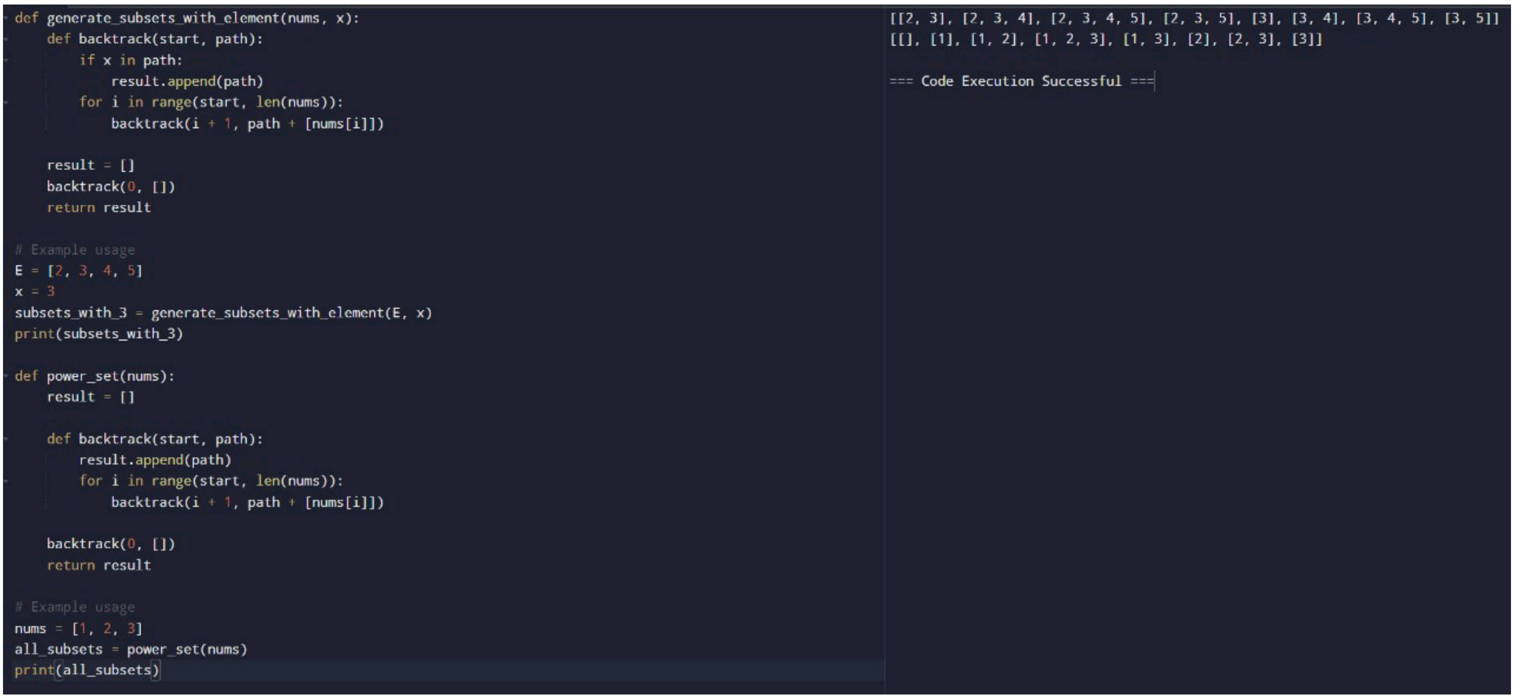
```
def is_hamiltonian_cycle(edges, n):
    from itertools import permutations

    # Create adjacency list
    graph = {i: [] for i in range(n)}
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    # Check all permutations of vertices
    for perm in permutations(range(n)):
        if all(perm[i] in graph[perm[i - 1]] for i in range(n)):
            return True
    return False

# Example usage
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
n = 4
print(is_hamiltonian_cycle(edges, n))
```

```
True

=== Co
```



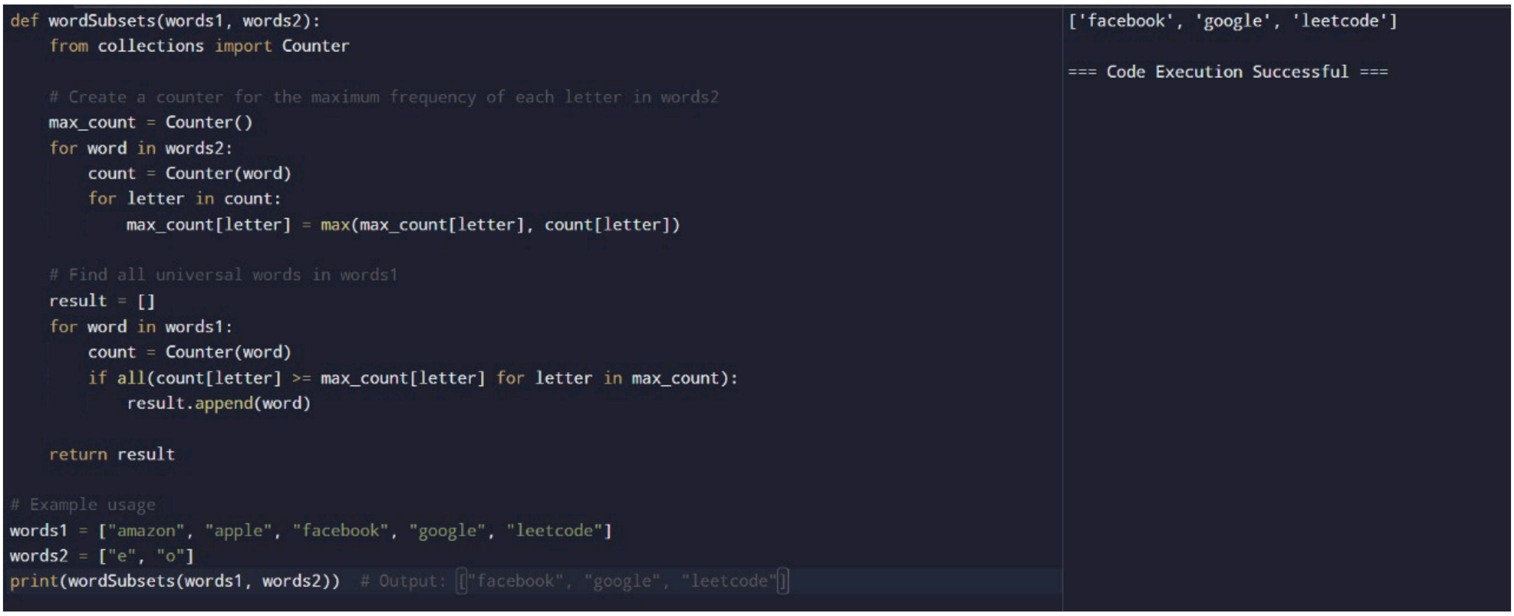
```python
def generate_subsets(S):
    S = sorted(set(S))  # Remove duplicates and sort
    subsets = []

    def backtrack(start, path):
        subsets.append(path)
        for i in range(start, len(S)):
            backtrack(i + 1, path + [S[i]])

    backtrack(0, [])
    return subsets

# Example usage
A = [1, 2, 3]
result = generate_subsets(A)
print(result)
```

```
[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]

=== Code Execution Successful ===
```



```python
def generate_subsets_with_element(nums, x):
    def backtrack(start, path):
        if x in path:
            result.append(path)
        for i in range(start, len(nums)):
            backtrack(i + 1, path + [nums[i]])

    result = []
    backtrack(0, [])
    return result

# Example usage
E = [2, 3, 4, 5]
x = 3
subsets_with_3 = generate_subsets_with_element(E, x)
print(subsets_with_3)

def power_set(nums):
    result = []

    def backtrack(start, path):
        result.append(path)
        for i in range(start, len(nums)):
            backtrack(i + 1, path + [nums[i]])

    backtrack(0, [])
    return result

# Example usage
nums = [1, 2, 3]
all_subsets = power_set(nums)
print(all_subsets)
```

```
[[2, 3], [2, 3, 4], [2, 3, 4, 5], [2, 3, 5], [3], [3, 4], [3, 4, 5], [3, 5]]
[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]

=== Code Execution Successful ===
```



```python
def wordSubsets(words1, words2):
    from collections import Counter

    # Create a counter for the maximum frequency of each letter in words2
    max_count = Counter()
    for word in words2:
        count = Counter(word)
        for letter in count:
            max_count[letter] = max(max_count[letter], count[letter])

    # Find all universal words in words1
    result = []
    for word in words1:
        count = Counter(word)
        if all(count[letter] >= max_count[letter] for letter in max_count):
            result.append(word)

    return result

# Example usage
words1 = ["amazon", "apple", "facebook", "google", "leetcode"]
words2 = ["e", "o"]
print(wordSubsets(words1, words2))  # Output: ["facebook", "google", "leetcode"]
```

```
['facebook', 'google', 'leetcode']

=== Code Execution Successful ===
```