In [ ]:
```
def slice_and_reverse(lst):
    chunk_size = len(lst) // 3
    chunks = [lst[i:i+chunk_size] for i in range(0, len(lst), chunk_size)]

    reversed_chunks = [chunk[::-1] for chunk in chunks]
    return reversed_chunks

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
result = slice_and_reverse(my_list)
print(result)
```

In [ ]:
```
def even(arr):
    chunk = len(arr)// 3
    return [arr[0:i: chunk] for i in range(0, len(arr), chunk)]

arr = [3,4,6,7,5,4,5,7,8,8]
print("Original: ", arr)
print("Chunks: ", even(arr))
```

In [ ]:
```
def count_characters(s):
    result = {}
    for char in s:
        if char in result:
            result[char] += 1
        else:
            result[char] = 1
    return result

# Example usage:
string = "programming"
result = count_characters(string)
print(result)
# Output: {'p': 1, 'r': 2, 'o': 1, 'g': 2, 'a': 1, 'm': 1, 'i': 1, 'n': 1}
```

In [ ]:
```
def is_palindrome(s):
    s = ''.join(c.lower() for c in s if c.isalnum())
    return s == s[::-1]

# Example usage:
print(is_palindrome("A man a plan a canal Panama"))
# Output: True
```

In [ ]:
```
def merge_dictionaries(dict1, dict2):
    result = dict1.copy()
    for key, value in dict2.items():
        if key in result:
            if isinstance(result[key], list):
                result[key].append(value)
            else:
                result[key] = [result[key], value]
        else:
            result[key] = value
    return result

# Example usage:
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = {'b': 4, 'd': 5, 'e': 6}
```

```
result = merge_dictionaries(dict1, dict2)
print(result)
# Output: {'a': 1, 'b': [2, 4], 'c': 3, 'd': 5, 'e': 6}
```

In [ ]:
```
def find_common_elements(list1, list2):
    return list(set(list1) & set(list2))

# Example usage:
list1 = [1, 2, 3, 4, 5]
list2 = [ 4, 5, 7]
result = find_common_elements(list1, list2)
print(result)
# Output: [3, 4, 5]
```

In [ ]:
```
def fizzbuzz():
    for i in range(1, 21):
        if i % 3 == 0 and i % 5 == 0:
            print("FizzBuzz")
        elif i % 3 == 0:
            print("Fizz")
        elif i % 5 == 0:
            print("Buzz")
        else:
            print(i)

# Example usage:
fizzbuzz()
# Output: 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz, ...
```

In [ ]:
```
def is_prime(number):
    if number < 2:
        return False
    for i in range(2, int(number**0.5) + 1):
        if number % i == 0:
            return False
    return True

# Example usage:
print(is_prime(7))
# Output: True
```

In [ ]:
```
def reverse_words(sentence):
    words = sentence.split()
    reversed_sentence = ' '.join(reversed(words))
    return reversed_sentence

# Example usage:
sentence = "Hello World, how are you?"
result = reverse_words(sentence)
print(result)
# Output: "you? are how World, Hello"
```

In [ ]:
```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

```
# Example usage:
result = factorial(5)
print(result)
# Output: 120
```

In [ ]:
```
def remove_duplicates(lst):
    unique_elements = []
    for item in lst:
        if item not in unique_elements:
            unique_elements.append(item)
    return unique_elements

# Example usage:
my_list = [1, 2, 2, 3, 4, 4, 5]
result = remove_duplicates(my_list)
print(result)
# Output: [1, 2, 3, 4, 5]
```

In [ ]:
```
def fibonacci_sequence(n):
    sequence = [0, 1]
    while sequence[-1] + sequence[-2] <= n:
        sequence.append(sequence[-1] + sequence[-2])
    return sequence

# Example usage:
result = fibonacci_sequence(20)
print(result)
# Output: [0, 1, 1, 2, 3, 5, 8, 13]
```

In [ ]:
```
def capitalize_words(sentence):
    words = sentence.split()
    capitalized_words = [word.capitalize() for word in words]
    return ' '.join(capitalized_words)

# Example usage:
sentence = "hello world, how are you?"
result = capitalize_words(sentence)
print(result)
# Output: "Hello World, How Are You?"
```

In [ ]:
```
def capitalize_and_small(sentence):
    sentence_upper = sentence.upper()
    sentence_lower = sentence.lower()
    return sentence_upper, sentence_lower

# Example usage:
sentence = "Hello World, how are you?"
result_upper, result_lower = capitalize_and_small(sentence)

print("Uppercase version:", result_upper)
print("Lowercase version:", result_lower)
```

In [ ]:
```
def majority_element(nums):
    count, candidate = 0, None
    for num in nums:
        if count == 0:
            candidate = num
        count += (1 if num == candidate else -1)
```

```python
    return candidate

# Example usage:
nums = [3, 3, 4, 2, 4, 4, 2, 4, 4]
result = majority_element(nums)
print(result)
# Output: 4
```

In [ ]:

```python
from collections import deque

def DFS(graph, start):
    visit = set()
    stack = [start]
    tree = []
    while stack:
        node = stack.pop()
        if node not in visit:
            tree.append(node)
            visit.add(node)
            stack.extend(neighbor for neighbor in graph[node] if neighbor not in visit
    return tree

def BFS(graph, start):
    visit = set()
    queue = deque([start])
    tree = []
    while queue:
        node = queue.popleft()
        if node not in visit:
            tree.append(node)
            visit.add(node)
            queue.extend(neighbor for neighbor in graph[node] if neighbor not in visit
    return tree

def DFS_PATH(graph, start, end, path = []):
    path = path + [start]
    if start not in graph:
        return None
    if start == end:
        return path
    for neighbor in graph[start]:
        if neighbor not in path:
            new_path = DFS_PATH(graph, neighbor, end, path)
            if new_path:
                return new_path
    return None

def BFS_PATH(graph, start, end):
    if start not in graph:
        return None
    queue = deque([(start, [start])])
    while queue:
        node, path = queue.popleft()
        if node == end:
            return path
        for neighbor in graph[node]:
            if neighbor not in path:
                queue.append((neighbor, path +[neighbor]))
```

```
def costcal(graph, path):
    total_cost = 0
    for i in range(len(path) - 1):
        current_node = path[i]
        next_node = path[i + 1]
        if current_node in graph and next_node in graph[current_node]:
            edge_cost = graph[current_node][next_node]
            total_cost += edge_cost
        else:
            return None
    return total_cost

my_graph = {
    '1': {'2': 9, '4': 2},
    '2': {'1': 2, '4': 5},
    '3': {},
    '4': {'6': 4, '8': 2},
    '5': {'8': 1},
    '6': {},
    '7': {'5': 2, '3': 6, '6': 3},
    '8': {},
}

print("DFS Searched: ", DFS(my_graph, '1'))
print("DFS_PATH :", DFS_PATH(my_graph, '1', '8'))
print("Cost: ", costcal(my_graph, DFS_PATH(my_graph, '1', '8')))

print("\n")

print("BFS Searched: ", BFS(my_graph, '1'))
print("BFS_PATH :", BFS_PATH(my_graph, '1', '8'))
print("Cost: ", costcal(my_graph, BFS_PATH(my_graph, '1', '8')))
```

In [ ]:

```
import queue as q
Graph = {
    'S': {'A':(1, 3),'G':(10, 0)},
    'A': {'B':(2, 4),'C':(1, 2)},
    'B': {'D':(5, 6)},
    'C': {'D':(3, 6),'G':(4, 0)},
    'D': {'G':(6, 0)},
    'G': {}
}
startingHeuristic = 5

def A_STR(MyGraph1, end, start):
    QU = q.PriorityQueue()
    value = startingHeuristic
    w = (0, (value, start))
    path = []
    QU.put(w)
    while QU:
        vertex = QU.get(0)
        print(vertex[0])
        n = vertex[-1][-1]
        if n not in path:
            path.append(n)
            if n == end:
                s = str(vertex)
                return path
```

```
                #return[s, vertex[0]]
            edges = list(MyGraph1[n].keys())
            print(edges)
            for i in range(len(edges)):
                add = list(vertex)
                add.append(edges[i])
                cost = vertex[0] + MyGraph1[n][edges[i]][0] + MyGraph1[n][edges[i]][1]
                t = (cost, add)
                QU.put(t)
                print(t)



cost = 0
path = []

#for i in range(0, len(arr)-1):
abc = A_STR(Graph, 'G', 'S')
 #   print("Path from ", arr[i], "to ", arr[i+1],  abc[0])
 #   print("cost: ", abc[1])
 #   path = path + list(abc[0])
 #   cost = cost + int(abc[1])
```

In [ ]:

```
import heapq

def ucs(graph, start, goal):
    priority_queue = [(0, start, [])]

    while priority_queue:
        cost, current_node, path = heapq.heappop(priority_queue)

        if current_node == goal:
            return path + [current_node]

        for neighbor, (edge_cost, _) in graph[current_node].items():
            heapq.heappush(priority_queue, (cost + edge_cost, neighbor, path + [currer

    return None  # Goal not reached

# Example usage:
graph = {
    'S': {'A': (1, 3), 'G': (10, 0)},
    'A': {'B': (2, 4), 'C': (1, 2)},
    'B': {'D': (5, 6)},
    'C': {'D': (3, 6), 'G': (4, 0)},
    'D': {'G': (6, 0)},
    'G': {}
}

start_node = 'S'
goal_node = 'G'

result = ucs(graph, start_node, goal_node)
print("UCS Path:", result)
```

In [ ]:

```
import heapq
from collections import deque

def dfs(graph, start, goal):
```

```
    stack = [start]
    visited = set()

    while stack:
        current_node = stack.pop()
        if current_node == goal:
            return True
        if current_node not in visited:
            visited.add(current_node)
            stack.extend(neighbor for neighbor in graph[current_node] if neighbor not

    return False

def bfs(graph, start, goal):
    queue = deque([start])
    visited = set()

    while queue:
        current_node = queue.popleft()
        if current_node == goal:
            return True
        if current_node not in visited:
            visited.add(current_node)
            queue.extend(neighbor for neighbor in graph[current_node] if neighbor not

    return False

def ucs(graph, start, goal):
    priority_queue = [(0, start, [])]

    while priority_queue:
        cost, current_node, path = heapq.heappop(priority_queue)

        if current_node == goal:
            return path + [current_node]

        for neighbor, (edge_cost, _) in graph[current_node].items():
            heapq.heappush(priority_queue, (cost + edge_cost, neighbor, path + [currer

    return None

def astar(graph, start, goal):
    priority_queue = [(0, start, [])]

    while priority_queue:
        _, current_node, path = heapq.heappop(priority_queue)

        if current_node == goal:
            return path + [current_node]

        for neighbor, (edge_cost, heuristic) in graph[current_node].items():
            heapq.heappush(priority_queue, (cost + edge_cost + heuristic, neighbor, pa

    return None

# Example graph
graph = {
    'S': {'A': (1, 3), 'G': (10, 0)},
    'A': {'B': (2, 4), 'C': (1, 2)},
    'B': {'D': (5, 6)},
```

```
    'C': {'D': (3, 6), 'G': (4, 0)},
    'D': {'G': (6, 0)},
    'G': {}
}

start_node = 'S'
goal_node = 'G'

# DFS
dfs_result = dfs(graph, start_node, goal_node)
print("DFS Path:", dfs_result)

# BFS
bfs_result = bfs(graph, start_node, goal_node)
print("BFS Path:", bfs_result)

# UCS
ucs_result = ucs(graph, start_node, goal_node)
print("UCS Path:", ucs_result)

# A*
astar_result = astar(graph, start_node, goal_node)
print("A* Path:", astar_result)
```

In [ ]:

```
import heapq
from collections import deque

def dfs(graph, start, goal):
    stack = [(start, 0, [])]
    visited = set()

    while stack:
        current_node, cost, path = stack.pop()
        if current_node == goal:
            return path + [current_node], cost
        if current_node not in visited:
            visited.add(current_node)
            stack.extend((neighbor, cost + edge_cost, path + [current_node]) for neigh

    return None, None

def bfs(graph, start, goal):
    queue = deque([(start, 0, [])])
    visited = set()

    while queue:
        current_node, cost, path = queue.popleft()
        if current_node == goal:
            return path + [current_node], cost
        if current_node not in visited:
            visited.add(current_node)
            queue.extend((neighbor, cost + edge_cost, path + [current_node]) for neigh

    return None, None

def ucs(graph, start, goal):
    priority_queue = [(0, start, [])]

    while priority_queue:
```

```
        cost, current_node, path = heapq.heappop(priority_queue)

        if current_node == goal:
            return path + [current_node], cost

        for neighbor, (edge_cost, _) in graph[current_node].items():
            heapq.heappush(priority_queue, (cost + edge_cost, neighbor, path + [currer

    return None, None

def astar(graph, start, goal):
    priority_queue = [(0, start, [])]

    while priority_queue:
        _, current_node, path = heapq.heappop(priority_queue)

        if current_node == goal:
            return path + [current_node], calculate_cost(path, graph)

        for neighbor, (edge_cost, heuristic) in graph[current_node].items():
            heapq.heappush(priority_queue, (cost + edge_cost + heuristic, neighbor, pa

    return None, None

def calculate_cost(path, graph):
    cost = 0
    for i in range(len(path) - 1):
        current_node = path[i]
        next_node = path[i + 1]
        cost += graph[current_node][next_node][0]  # Edge cost
    return cost

# Example graph
graph = {
    'S': {'A': (1, 3), 'G': (10, 0)},
    'A': {'B': (2, 4), 'C': (1, 2)},
    'B': {'D': (5, 6)},
    'C': {'D': (3, 6), 'G': (4, 0)},
    'D': {'G': (6, 0)},
    'G': {}
}

start_node = 'S'
goal_node = 'G'

# DFS
dfs_path, dfs_cost = dfs(graph, start_node, goal_node)
print("DFS Path:", dfs_path)
print("DFS Cost:", dfs_cost)

# BFS
bfs_path, bfs_cost = bfs(graph, start_node, goal_node)
print("BFS Path:", bfs_path)
print("BFS Cost:", bfs_cost)

# UCS
ucs_path, ucs_cost = ucs(graph, start_node, goal_node)
print("UCS Path:", ucs_path)
print("UCS Cost:", ucs_cost)
```

```
# A*
astar_path, astar_cost = astar(graph, start_node, goal_node)
print("A* Path:", astar_path)
print("A* Cost:", astar_cost)
```

In [ ]:
```
import heapq
from collections import deque

# Define the initial and goal states
initial_state = [
    [1, 2, 3],
    [0, 4, 6],
    [7, 5, 8]
]

goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

def print_puzzle(puzzle):
    for row in puzzle:
        print(row)
    print()

def find_zero_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def swap(state, row1, col1, row2, col2):
    new_state = [row.copy() for row in state]
    new_state[row1][col1], new_state[row2][col2] = new_state[row2][col2], new_state[rc
    return new_state

def get_neighbors(state):
    neighbors = []
    zero_row, zero_col = find_zero_position(state)

    # Try moving the empty space (0) to the left
    if zero_col > 0:
        neighbors.append(swap(state, zero_row, zero_col, zero_row, zero_col - 1))

    # Try moving the empty space to the right
    if zero_col < 2:
        neighbors.append(swap(state, zero_row, zero_col, zero_row, zero_col + 1))

    # Try moving the empty space upward
    if zero_row > 0:
        neighbors.append(swap(state, zero_row, zero_col, zero_row - 1, zero_col))

    # Try moving the empty space downward
    if zero_row < 2:
        neighbors.append(swap(state, zero_row, zero_col, zero_row + 1, zero_col))

    return neighbors
```

```
def heuristic(state, goal_state):
    h = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                goal_row, goal_col = divmod(goal_state.index(value), 3)
                h += abs(i - goal_row) + abs(j - goal_col)
    return h

def dfs_puzzle(initial_state, goal_state, max_depth=float('inf')):
    stack = [(tuple(map(tuple, initial_state)), [])]
    visited = set()

    while stack:
        current_state, path = stack.pop()
        if current_state == goal_state:
            return path
        if current_state not in visited and len(path) < max_depth:  # Limit the depth
            visited.add(current_state)
            stack.extend((tuple(map(tuple, next_state)), path + [current_state]) for r

    return None



def bfs_puzzle(initial_state, goal_state):
    queue = deque([(initial_state, [])])
    visited = set()

    while queue:
        current_state, path = queue.popleft()
        if current_state == goal_state:
            return path
        if current_state not in visited:
            visited.add(current_state)
            queue.extend((next_state, path + [current_state]) for next_state in get_ne

    return None

def ucs_puzzle(initial_state, goal_state):
    priority_queue = [(0, initial_state, [])]
    visited = set()

    while priority_queue:
        cost, current_state, path = heapq.heappop(priority_queue)
        if current_state == goal_state:
            return path
        if current_state not in visited:
            visited.add(current_state)
            priority_queue.extend((cost + 1, next_state, path + [current_state]) for r

    return None

def astar_puzzle(initial_state, goal_state):
    priority_queue = [(0, initial_state, [])]
    visited = set()

    while priority_queue:
        _, current_state, path = heapq.heappop(priority_queue)
```

```
        if current_state == goal_state:
            return path
        if current_state not in visited:
            visited.add(current_state)
            priority_queue.extend((cost + 1 + heuristic(next_state, goal_state), next_

    return None

print("Initial State:")
print_puzzle(initial_state)

# Solve the puzzle using DFS
dfs_path = dfs_puzzle(initial_state, goal_state)
if dfs_path:
    print("DFS Solution Path:")
    for step, state in enumerate(dfs_path):
        print(f"Step {step + 1}:")
        print_puzzle(state)
else:
    print("DFS: No solution found.")

# Solve the puzzle using BFS
bfs_path = bfs_puzzle(initial_state, goal_state)
if bfs_path:
    print("BFS Solution Path:")
    for step, state in enumerate(bfs_path):
        print(f"Step {step + 1}:")
        print_puzzle(state)
else:
    print("BFS: No solution found.")

# Solve the puzzle using UCS
ucs_path = ucs_puzzle(initial_state, goal_state)
if ucs_path:
    print("UCS Solution Path:")
    for step, state in enumerate(ucs_path):
        print(f"Step {step + 1}:")
        print_puzzle(state)
else:
    print("UCS: No solution found.")

# Solve the puzzle using A*
astar_path = astar_puzzle(initial_state, goal_state)
if astar_path:
    print("A* Solution Path:")
    for step, state in enumerate(astar_path):
        print(f"Step {step + 1}:")
        print_puzzle(state)
else:
    print("A*: No solution found.")
```

In [ ]: