1. Kadane's Algorithm

Problem: Given an integer array, find the contiguous subarray with the maximum sum.

Example Input:

```
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
```

Expected Output: `6` (The subarray `[4, -1, 2, 1]` has the maximum sum.)

Logic Behind the Algorithm: Kadane's Algorithm is designed to solve the "Maximum Subarray Sum" problem efficiently in linear time. The core idea is to maintain two variables:

- `max_ending_here`: This keeps track of the maximum subarray sum that ends at the current position.
- `max_so_far`: This keeps track of the overall maximum subarray sum found so far.

As you iterate through the array, you decide at each element whether to add it to the existing subarray or start a new subarray with the current element.

Code:

```
def kadane(arr):
    max_so_far = max_ending_here = arr[0]
    for num in arr[1:]:
        max_ending_here = max(num, max_ending_here + num)  # Choose max of current num or
current num + previous sum
        max_so_far = max(max_so_far, max_ending_here)  # Update overall max if needed
    return max_so_far

# Example usage
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(kadane(arr))  # Output: 6
```

Time Complexity: O(n) - The algorithm makes a single pass through the array.
Space Complexity: O(1) - No additional space is used aside from a few variables.

---

2. Binary Search

Problem: Given a sorted array of integers, write a function to search for a target value. If found, return its index. Otherwise, return `-1`.

Example Input:

```
arr = [1, 2, 3, 4, 5]
target = 3
```

Expected Output: `2` (The index of the target value.)

Logic Behind the Algorithm: Binary Search works on the principle of divide and conquer. Since the array is sorted, you can efficiently narrow down the search space by repeatedly dividing it in half:

1. Check the middle element of the array.
2. If it matches the target, return the index.
3. If the middle element is greater than the target, narrow your search to the left half.
4. If it's less, search the right half.
5. Repeat until the target is found or the search space is exhausted.

Code:

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# Example usage
arr = [1, 2, 3, 4, 5]
target = 3
print(binary_search(arr, target))  # Output: 2
```

Time Complexity: O(log n) - Each comparison cuts the search space in half.
Space Complexity: O(1) - Only a few pointers are used.

---

3. Merge Sort

Problem: Sort an array using the merge sort algorithm.

Example Input:

```
arr = [38, 27, 43, 3, 9, 82, 10]
```

Expected Output: `[3, 9, 10, 27, 38, 43, 82]`

Logic Behind the Algorithm: Merge Sort is a classic divide-and-conquer algorithm:

1. Split the array into halves recursively until each subarray has one element (base case).
2. Merge the sorted halves back together:
   - Compare the smallest elements of both halves and build a new sorted array.

Code:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)

def merge(left, right):
    sorted_arr = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_arr.append(left[i])
            i += 1
        else:
            sorted_arr.append(right[j])
            j += 1
    sorted_arr.extend(left[i:])
    sorted_arr.extend(right[j:])
    return sorted_arr

# Example usage
arr = [38, 27, 43, 3, 9, 82, 10]
print(merge_sort(arr))  # Output: [3, 9, 10, 27, 38, 43, 82]
```

Time Complexity: O(n log n) - Splitting takes log n and merging takes O(n).
Space Complexity: O(n) - Requires additional space for the temporary arrays used during

merging.

---

4. Quick Sort

Problem: Sort an array using the quick sort algorithm.

Example Input:

```
arr = [10, 7, 8, 9, 1, 5]
```

Expected Output: `[1, 5, 7, 8, 9, 10]`

Logic Behind the Algorithm: Quick Sort is another divide-and-conquer algorithm:

1. Choose a "pivot" element from the array.
2. Partition the other elements into two subarrays according to whether they are less than or greater than the pivot.
3. Recursively apply the same strategy to the subarrays.
4. The base case is when the array has one or zero elements, which are inherently sorted.

Code:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]  # Choose pivot
    left = [x for x in arr if x < pivot]  # Elements less than pivot
    middle = [x for x in arr if x == pivot]  # Elements equal to pivot
    right = [x for x in arr if x > pivot]  # Elements greater than pivot
    return quick_sort(left) + middle + quick_sort(right)

# Example usage
arr = [10, 7, 8, 9, 1, 5]
print(quick_sort(arr))  # Output: [1, 5, 7, 8, 9, 10]
```

Time Complexity: O(n log n) on average, $O(n^2)$ in the worst case (when the pivot is the smallest/largest element repeatedly).
Space Complexity: O(log n) for the recursive stack space.

---

5. Depth-First Search (DFS)

Problem: Given a graph represented as an adjacency list, perform a DFS starting from a given vertex.

Example Input:

```
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
}
```

Starting Vertex: `A`

Expected Output: `{'A', 'B', 'C', 'D', 'E'}` (A set of all visited nodes.)

Logic Behind the Algorithm: DFS explores as far as possible along each branch before backing up. The algorithm uses recursion (or an explicit stack) to remember which nodes to visit next:

1. Start from the source node, mark it as visited.
2. For each unvisited neighbor, recursively perform DFS.
3. Continue until all nodes reachable from the source have been visited.

Code:

```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()  # Create a set to track visited nodes
    visited.add(start)  # Mark the current node as visited
    for neighbor in graph[start]:  # Explore neighbors
        if neighbor not in visited:
            dfs(graph, neighbor, visited)  # Recursive call for unvisited neighbors
    return visited

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
}
print(dfs(graph, 'A'))  # Output: {'A', 'B', 'C', 'D', 'E'}
```

Time Complexity: O(V + E) - Where V is vertices and E is edges.
Space Complexity: O(V) - For the visited set and the call stack.

6. Breadth-First Search (BFS)

Problem: Given a graph represented as an adjacency list, perform a BFS starting from a given vertex.

Example Input:

```python
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],


    'E': []
}
```

Starting Vertex: `A`

Expected Output: `{'A', 'B', 'C', 'D', 'E'}` (A set of all visited nodes.)

Logic Behind the Algorithm: BFS explores all neighbors at the present depth prior to moving on to nodes at the next depth level. It uses a queue to track which node to visit next:

1. Start from the source node, mark it as visited, and enqueue it.
2. While the queue is not empty, dequeue a node, visit all its unvisited neighbors, mark them as visited, and enqueue them.
3. Continue until the queue is empty.

Code:

```python
from collections import deque

def bfs(graph, start):
    visited = set()  # Track visited nodes
    queue = deque([start])  # Initialize the queue with the start node
    visited.add(start)
    while queue:
        vertex = queue.popleft()  # Dequeue the next node
        for neighbor in graph[vertex]:  # Explore neighbors
            if neighbor not in visited:
                visited.add(neighbor)  # Mark as visited
                queue.append(neighbor)  # Enqueue unvisited neighbors
    return visited

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
}
print(bfs(graph, 'A'))  # Output: {'A', 'B', 'C', 'D', 'E'}
```

Time Complexity: O(V + E)
Space Complexity: O(V) - For the queue and visited set.

---

7. Dijkstra's Algorithm

Problem: Given a graph represented as an adjacency list with weights, find the shortest path from a source vertex to all other vertices.

Example Input:

```
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
```

Starting Vertex: `A`

Expected Output: `{'A': 0, 'B': 1, 'C': 3, 'D': 4}` (Shortest paths from A.)

Logic Behind the Algorithm: Dijkstra's algorithm finds the shortest paths from a source to all vertices in a weighted graph:

1. Initialize distances from the source to all vertices as infinity and set the distance to the source itself as zero.
2. Use a priority queue (min-heap) to always expand the closest vertex.
3. For each neighbor of the current vertex, calculate the potential new distance. If it's shorter, update the distance and enqueue the neighbor.
4. Repeat until all reachable vertices have been processed.

Code:

```
import heapq

def dijkstra(graph, start):
    min_heap = [(0, start)]  # (distance, vertex)
    distances = {vertex: float('infinity') for vertex in graph}  # Initialize distances
    distances[start] = 0  # Distance to the start node is 0

    while min_heap:
        current_distance, current_vertex = heapq.heappop(min_heap)  # Get the vertex with the
smallest distance

        if current_distance > distances[current_vertex]:
            continue  # Skip if we found a better path already

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight  # Calculate distance to neighbor

            if distance < distances[neighbor]:  # Update if the new distance is shorter
                distances[neighbor] = distance
                heapq.heappush(min_heap, (distance, neighbor))  # Add neighbor to the queue

    return distances

# Example usage
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
print(dijkstra(graph, 'A'))  # Output: {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

Time Complexity: O(E log V) - Each edge is processed once, and the priority queue operations are logarithmic.
Space Complexity: O(V) - For the distances dictionary and the priority queue.

---

8. Floyd-Warshall Algorithm

Problem: Find the shortest paths between all pairs of vertices in a weighted graph.

Example Input:

```
graph = [
    [0, 3, float('inf'), 5],
    [2, 0, float('inf'), 4],
    [float('inf'), 1, 0, float('inf')],
    [float('inf'), float('inf'), 2, 0]
]
```

Expected Output: Shortest distance matrix.

Logic Behind the Algorithm: Floyd-Warshall is an all-pairs shortest path algorithm:

1. Initialize a distance matrix where `dist[i][j]` is the weight of the edge from i to j, or infinity if there's no direct edge.
2. For each vertex k, update the distance from i to j as the minimum of the current distance and the distance via k.
3. This is repeated for all vertices, resulting in the shortest path distances between all pairs.

Code:

```
def floyd_warshall(graph):
    dist = list(map(lambda i: list(map(lambda j: j, i)), graph))  # Initialize distance array

    for k in range(len(graph)):  # Intermediate vertex
        for i in range(len(graph)):  # Start vertex
            for j in range(len(graph)):  # End vertex
                if dist[i][j] > dist[i][k] + dist[k][j]:  # Check for shorter path
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

# Example usage
graph = [
    [0, 3, float('inf'), 5],
    [2, 0, float('inf'), 4],
    [float('inf'), 1, 0, float('inf')],
    [float('inf'), float('inf'), 2, 0]
]
print(floyd_warshall(graph))
```

Time Complexity: O(V^3) - Three nested loops for all pairs.
Space Complexity: O(V^2) - For the distance matrix.

9. Kruskal's Algorithm

Problem: Find the Minimum Spanning Tree (MST) of a connected, undirected graph using Kruskal's algorithm.

Example Input:

```
edges = [
    (1, 2, 1),
    (1, 3, 3),
    (2, 3, 2),
    (2, 4, 4),
    (3, 4, 5)
]
```

Expected Output: MST edges and their total weight.

Logic Behind the Algorithm: Kruskal's Algorithm finds the MST by sorting the edges and adding them one by one:

1. Sort all edges in ascending order by weight.
2. Initialize a union-find structure to track connected components.
3. Iterate through the edges, adding them to the MST if they don't create a cycle (i.e., if the two vertices are in different components).
4. Stop when you have V-1 edges in your MST.

Code:

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            self.parent[root_u] = root_v

def kruskal(vertices, edges):
    edges.sort(key=lambda x: x[2])  # Sort edges by weight
    uf = UnionFind(vertices)
    mst = []
    total_weight = 0

    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):  # No cycle
            uf.union(u, v)  # Union the two components
            mst.append((u, v, weight))
            total_weight += weight

    return mst, total_weight

# Example usage
edges = [
    (0, 1, 1),
    (0, 2, 3),
    (1, 2, 2),
    (1, 3, 4),
    (2, 3, 5)
]
mst, total_weight = kruskal(4, edges)
print(mst)  # Output: [(0, 1, 1), (1, 2, 2), (1, 3, 4)]
print(total_weight)  # Output: 7
```

Time Complexity: O(E log E) - Sorting edges takes O(E log E), and union-find operations are nearly constant.
Space Complexity: O(V) - For the union-find structure.

10. Prim's Algorithm

Problem: Find the Minimum Spanning Tree (M

ST) of a connected, undirected graph using Prim’s algorithm.

Example Input:

```
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'A': 1, 'C': 2, 'D': 4},
    'C': {'A': 3, 'B': 2, 'D': 5},
    'D': {'B': 4, 'C': 5}
}
```

Expected Output: MST edges and their total weight.

Logic Behind the Algorithm: Prim’s Algorithm builds the MST incrementally:

1. Start from any vertex and add it to the MST.
2. Repeatedly add the smallest edge that connects a vertex in the MST to a vertex outside it until all vertices are included.

Code:

```python
import heapq

def prim(graph):
    start_vertex = next(iter(graph))  # Start from an arbitrary vertex
    mst = []
    total_weight = 0
    visited = {start_vertex}
    edges = [(weight, start_vertex, to) for to, weight in graph[start_vertex].items()]
    heapq.heapify(edges)  # Create a min-heap of edges

    while edges:
        weight, frm, to = heapq.heappop(edges)  # Get the smallest edge
        if to not in visited:
            visited.add(to)  # Mark vertex as visited
            mst.append((frm, to, weight))  # Add edge to MST
            total_weight += weight  # Update total weight
            for next_to, next_weight in graph[to].items():
                if next_to not in visited:
                    heapq.heappush(edges, (next_weight, to, next_to))  # Push new edges to the
heap

    return mst, total_weight

# Example usage
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'A': 1, 'C': 2, 'D': 4},
    'C': {'A': 3, 'B': 2, 'D': 5},
    'D': {'B': 4, 'C': 5}
}
mst, total_weight = prim(graph)
print(mst)  # Output: [('B', 'A', 1), ('C', 'B', 2), ('D', 'B', 4)]
print(total_weight)  # Output: 7
```

Time Complexity: O(E log V) - Each edge is processed and added to the priority queue.
Space Complexity: O(V) - For the visited set and the edge list.

---

11. Heap Sort

Problem: Sort an array using the heap sort algorithm.

Example Input:

```
arr = [12, 11, 13, 5, 6, 7]
```

Expected Output: `[5, 6, 7, 11, 12, 13]`

Logic Behind the Algorithm: Heap Sort utilizes a binary heap data structure to sort elements:

1. Build a max heap from the input array.
2. The largest element is at the root of the max heap. Swap it with the last element and reduce the heap size by one.
3. Heapify the root of the tree to restore the heap property.
4. Repeat the process until all elements are sorted.

Code:

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    # Build a max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements from heap
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap
        heapify(arr, i, 0)

# Example usage
arr = [12, 11, 13, 5, 6, 7]
heap_sort(arr)
print(arr)  # Output: [5, 6, 7, 11, 12, 13]
```

Time Complexity: O(n log n)
Space Complexity: O(1) - In-place sorting.

---

12. Counting Sort

Problem: Sort an array of integers where the range of the input data is known and limited.

Example Input:

```
arr = [4, 2, 2, 8, 3, 3, 1]
```

Expected Output: `[1, 2, 2, 3, 3, 4, 8]`

Logic Behind the Algorithm: Counting Sort counts the occurrences of each unique value in the input array:

1. Create a count array of size equal to the range of input values.
2. Count each element's occurrences and store them in the count array.
3. Modify the count array to determine the position of each element in the output array.
4. Build the output array based on the counts.

Code:

```python
def counting_sort(arr):
    max_val = max(arr)
    count = [0] * (max_val + 1)
    output = [0] * len(arr)

    # Count occurrences
    for num in arr:
        count[num] += 1

    # Update count array to get positions
    for i in range(1, len(count)):
        count[i] += count[i - 1]

    # Build the output array
    for num in reversed(arr):
        output[count[num] - 1] = num
        count[num] -= 1

    return output

# Example usage
arr = [4, 2, 2, 8, 3, 3, 1]
sorted_arr = counting_sort(arr)
print(sorted_arr)  # Output: [1, 2, 2, 3, 3, 4, 8]
```

Time Complexity: O(n + k) - Where n is the number of elements and k is the range of the input values.
Space Complexity: O(k) - For the count array.

---

13. Radix Sort

Problem: Sort an array of non-negative integers using radix sort.

Example Input:

```python
arr = [170, 45, 75, 90, 802, 24, 2, 66]
```

Expected Output: `[2, 24, 45, 66, 75, 90, 170, 802]`

Logic Behind the Algorithm: Radix Sort processes the integers digit by digit, starting from the least significant digit:

1. Use Counting Sort as a subroutine to sort the numbers based on each digit.

2. Start from the least significant digit to the most significant digit.
3. Repeat the process until all digits have been processed.

Code:

```python
def counting_sort_for_radix(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10  # Assuming base 10

    # Count occurrences based on the digit represented by exp
    for i in range(n):
        index = arr[i] // exp
        count[index % 10] += 1

    # Update count to reflect positions
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Build the output array
    for i in range(n - 1, -1, -1):
        index = arr[i] // exp
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1

    return output

def radix_sort(arr):
    max_val = max(arr)
    exp = 1
    while max_val // exp > 0:
        arr = counting_sort_for_radix(arr, exp)
        exp *= 10
    return arr

# Example usage
arr = [170, 45, 75, 90, 802, 24, 2, 66]
sorted_arr = radix_sort(arr)
print(sorted_arr)  # Output: [2, 24, 45, 66, 75, 90, 170, 802]
```

Time Complexity: O(nk) - Where n is the number of elements and k is the number of digits in the largest number.
Space Complexity: O(n) - For the output array.

14. Shell Sort

Problem: Sort an array using the shell sort algorithm.

Example Input:

```
arr = [12, 34, 54, 2, 3]
```

Expected Output: `[2, 3, 12, 34, 54]`

Logic Behind the Algorithm: Shell Sort is an optimization of insertion sort that allows the exchange of items far apart:

1. Start with a gap and compare elements that are that gap apart.
2. If the elements are in the wrong order, swap them.
3. Reduce the gap and repeat until the gap is 0, at which point the array is sorted.

Code:

```
def shell_sort(arr):
    n = len(arr)
    gap = n // 2  # Start with a large gap

    while gap > 0:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]  # Shift elements
                j -= gap
            arr[j] = temp  # Place temp in the right position
        gap //= 2  # Reduce the gap

# Example usage
arr = [12, 34, 54, 2, 3]
shell_sort(arr)
print(arr)  # Output: [2, 3, 12, 34, 54]
```

Time Complexity: O(n log n) to O(n^2) - Depends on the gap sequence.
Space Complexity: O(1) - In-place sorting.

---

15. Backtracking

Problem: Solve the N-Queens problem: Place N queens on an N×N chessboard such that no two queens threaten each other.

Example Input: `N = 4`

Expected Output: Possible arrangements of queens.

Logic Behind the Algorithm: Backtracking explores all possible arrangements of queens:

1. Place a queen in a row and move to the next row.
2. Check if the placement is valid (no two queens share the same column or diagonal).
3. If valid, recursively place queens in the next rows.
4. If all queens are placed, save the arrangement. If not, backtrack by removing the queen and trying the next column.

Code:

```
def is_safe(board, row, col):
    # Check the same column and both diagonals
    for i in range(row):
        if board[i] == col or \
           board[i] - i == col - row or \
           board[i] + i == col + row:
            return False
    return True

def solve_n_queens(n, row=0, board=[]):
    if row == n:
        print(board)  # Found a solution
        return
    for col in range(n):
        if is_safe(board, row, col):
            board.append(col)  # Place queen
            solve_n_queens(n, row + 1, board)
            board.pop()  # Backtrack

# Example usage
solve_n_queens(

4)  # Outputs possible arrangements of 4 queens
```

Time Complexity: O(N!) - N choices for the first queen, N-1 for the second, etc.
Space Complexity: O(N) - For the recursion stack.

16. Divide and Conquer (Merge Sort)

Problem: Sort an array using the merge sort algorithm.

Example Input:

```
arr = [38, 27, 43, 3, 9, 82, 10]
```

Expected Output: `[3, 9, 10, 27, 38, 43, 82]`

Logic Behind the Algorithm: Merge Sort divides the array into halves:

1. Recursively split the array into halves until each subarray has one element.
2. Merge the subarrays back together in sorted order.

Code:

```python
def merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid

    L = arr[left:mid + 1]
    R = arr[mid + 1:right + 1]

    i = j = 0
    k = left

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def merge_sort(arr, left, right):
    if left < right:
        mid = (left + right) // 2
        merge_sort(arr, left, mid)
        merge_sort(arr, mid + 1, right)
        merge(arr, left, mid, right)

# Example usage
arr = [38, 27, 43, 3, 9, 82, 10]
merge_sort(arr, 0, len(arr) - 1)
print(arr)  # Output: [3, 9, 10, 27, 38, 43, 82]
```

Time Complexity: O(n log n)
Space Complexity: O(n) - For the temporary arrays.

17. Quick Sort

Problem: Sort an array using the quick sort algorithm.

Example Input:

```
arr = [10, 7, 8, 9, 1, 5]
```

Expected Output: `[1, 5, 7, 8, 9, 10]`

Logic Behind the Algorithm: Quick Sort uses a pivot to partition the array:

1. Choose a pivot element from the array.
2. Partition the array into two subarrays: elements less than the pivot and elements greater than the pivot.
3. Recursively apply the above steps to the subarrays.

Code:

```
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi - 1)  # Before pi
        quick_sort(arr, pi + 1, high)  # After pi

# Example usage
arr = [10, 7, 8, 9, 1, 5]
quick_sort(arr, 0, len(arr) - 1)
print(arr)  # Output: [1, 5, 7, 8, 9, 10]
```

Time Complexity: O(n log n) on average; O(n^2) in the worst case.
Space Complexity: O(log n) - For the recursion stack.

18. Depth-First Search (DFS)

Problem: Perform a depth-first search on a graph.

Example Input:

```
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
}
```

Starting Vertex: `A`

Expected Output: `['A', 'B', 'D', 'C', 'E']` (Order of nodes visited.)

Logic Behind the Algorithm: DFS explores as far as possible along each branch before backtracking:

1. Start from the source node, mark it as visited, and explore each unvisited neighbor recursively.
2. Continue until all nodes are visited.

Code:

```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = []
    visited.append(start)  # Mark the node as visited
    for neighbor in graph[start]:  # Explore neighbors
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
}
print(dfs(graph, 'A'))  # Output: ['A', 'B', 'D', 'C', 'E']
```

Time Complexity: O(V + E)
Space Complexity: O(V) - For the visited list.

---

19. Binary Search

Problem: Find the index of a target value in a sorted array.

Example Input:

```python
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
target = 5
```

Expected Output: `4` (Index of the target.)

Logic Behind the Algorithm: Binary Search repeatedly divides the search interval in half:

1. Start with the entire sorted array.
2. Compare the target value to the middle element of the array.
3. If the target is equal to the middle element, return its index.
4. If the target is less, repeat the process in the left half; if greater, repeat in the right half.

Code:

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2  # Prevent overflow
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  # Target not found

# Example usage
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
target = 5
print(binary_search(arr, target))  # Output: 4
```

Time Complexity: O(log n)
Space Complexity: O(1) - Iterative approach.

---

20. Breadth-First Search (BFS) for Trees

Problem: Perform a level-order traversal on a binary tree.

Example Input:

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
```

Expected Output: `[1, 2, 3, 4, 5]` (Values in level order.)

Logic Behind the Algorithm: BFS for trees visits nodes level by level:

1. Use a queue to track nodes to visit.

2. Start from the root, enqueue it, and mark it as visited.
3. While the queue is not empty, dequeue a node, process it, and enqueue its children.

Code:

```
from collections import deque

def bfs_tree(root):
    if not root:
        return []

    result = []
    queue = deque([root])  # Initialize the queue

    while queue:
        node = queue.popleft()  # Dequeue the next node
        result.append(node.value)  # Process the node

        if node.left:
            queue.append(node.left)  # Enqueue left child
        if node.right:
            queue.append(node.right)  # Enqueue right child

    return result

# Example usage
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print(bfs_tree(root))  # Output: [1, 2, 3, 4, 5]
```

Time Complexity: O(n) - Every node is visited once.
Space Complexity: O(n) - For the queue in the worst case.

---

21. Dijkstra's Algorithm

Problem: Find the shortest path from a starting vertex to all other vertices in a weighted graph.

Example Input:

```
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
```

Starting Vertex: `A`

Expected Output: Shortest paths from A to all other vertices.

Logic Behind the Algorithm: Dijkstra's Algorithm maintains a priority queue of vertices to explore:

1. Initialize distances to all vertices as infinite, except for the starting vertex (set to 0).
2. Use a priority queue to extract the vertex with the smallest distance.
3. Update distances for its neighbors and add them to the queue.
4. Repeat until all vertices have been processed.

Code:

```python
import heapq

def dijkstra(graph, start):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    priority_queue = [(0, start)]  # (distance, vertex)

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example usage
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
shortest_paths = dijkstra(graph, 'A')
print(shortest_paths)  # Output: {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

Time Complexity: O(E log V) - Where E is the number of edges and V is the number of vertices.
Space Complexity: O(V) - For the distances dictionary.

---

22. Floyd-Warshall Algorithm

Problem: Find the shortest paths between all pairs of vertices in a weighted graph.

Example Input:

```python
graph = [
    [0, 5, float('inf'), 10],
    [float('inf'), 0, 3, float('inf')],
    [float('inf'), float('inf'), 0, 1],
    [float('inf'), float('inf'), float('inf'), 0]
]
```

Expected Output: Shortest path matrix.

Logic Behind the Algorithm: Floyd-Warshall checks all pairs of vertices:

1. Initialize a distance matrix with the graph weights.
2. For each vertex, update the distance matrix considering if going through that vertex yields a shorter path.
3. Repeat for all vertices.

Code:

```python
def floyd_warshall(graph):
    num_vertices = len(graph)
    distance = [[graph[i][j] for j in range(num_vertices)] for i in range(num_vertices)]

    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                if distance[i][j] > distance[i][k] + distance[k][j]:
                    distance[i][j] = distance[i][k] + distance[k][j]

    return distance

# Example usage
graph = [
    [0, 5, float('inf'), 10],
    [float('inf'), 0, 3, float('inf')],
    [float('inf'), float('inf'), 0, 1],
    [float('inf'), float('inf'), float('inf'), 0]
]
shortest_paths = floyd_warshall(graph)
for row in shortest_paths:
    print(row)  # Output: Shortest path matrix
```

Time Complexity: O(V^3) - Where V is the number of vertices.
Space Complexity: O(V^2) - For the distance matrix.

23. Kruskal's Algorithm

Problem: Find the Minimum Spanning Tree (MST) of a connected, undirected graph.

Example Input:

```
edges = [
    (1, 2, 1),
    (1, 3, 3),
    (2, 3, 2),
    (2, 4, 4),
    (3, 4, 5)
]
```

Expected Output: MST edges and their total weight.

Logic Behind the Algorithm: Kruskal’s Algorithm adds edges in increasing order of weight:

1. Sort all edges by weight.
2. Use a disjoint set to detect cycles while adding edges.
3. Add edges to the MST until it has V-1 edges.

Code:

```python
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])  # Path compression
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            self.parent[root_v] = root_u  # Union

def kruskal(n, edges):
    edges.sort(key=lambda x: x[2])  # Sort by weight
    ds = DisjointSet(n)
    mst = []
    total_weight = 0

    for u, v, weight in edges:
        if ds.find(u) != ds.find(v):
            ds.union(u, v)
            mst.append((u, v, weight))
            total_weight += weight

    return mst, total_weight

# Example usage
edges = [
    (1, 2, 1),
    (1, 3, 3),
    (2, 3, 2),
    (2, 4, 4),
    (3, 4, 5)
]
mst, total_weight = kruskal(5, edges)
print(mst)  # Output: [(1, 2, 1), (2, 3, 2), (2, 4, 4)]
print(total_weight)  # Output: 7
```

Time Complexity: O(E log E) - Sorting edges.
Space Complexity: O(V) - For the disjoint set.

24. Topological Sort

Problem: Find a topological ordering of a directed acyclic graph (DAG).

Example Input:

```
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['D'],
    'D': []
}
```

Expected Output: A valid topological order, e.g., `['A', 'B', 'C', 'D']`.

Logic Behind the Algorithm: Topological Sort orders vertices such that for every directed edge u → v, vertex u comes before v:

1. Count in-degrees of all vertices.
2. Initialize a queue with vertices of in-degree 0.
3. While the queue is not empty, dequeue a vertex, add it to the order, and decrease the in-degree of its neighbors. If any neighbor's in-degree becomes 0, enqueue it.

Code:

```python
from collections import deque

def topological_sort(graph):
    in_degree = {u: 0 for u in graph}  # Initialize in-degrees
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1

    queue = deque([u for u in in_degree if in_degree[u] == 0])
    top_order = []

    while queue:
        u = queue.popleft()
        top_order.append(u)

        for v in graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)

    return top_order

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['D'],
    'D': []
}
print(topological_sort(graph))  # Output: ['A', 'B', 'C', 'D'] or similar
```

Time Complexity: O(V + E)
Space Complexity: O(V) - For in-degrees and the queue.

---

25. Hash Table (Open Addressing)

Problem: Implement a simple hash table with open addressing.

Example Input:

```python
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def hash_function(self, key):
        return key % self.size

    def insert(self, key, value):
        index = self.hash_function(key)
        while self.table[index] is not None:
            index = (index + 1) % self.size  # Linear probing
        self.table[index] = (key, value)

    def get(self, key):


        index = self.hash_function(key)
        while self.table[index] is not None:
            if self.table[index][0] == key:
                return self.table[index][1]
            index = (index + 1) % self.size
        return None  # Not found
```

Expected Output: Ability to insert and retrieve values based on keys.

Logic Behind the Algorithm: A hash table stores key-value pairs:

1. Use a hash function to compute the index for the key.
2. If a collision occurs, use open addressing (e.g., linear probing) to find the next available slot.

Code:

```python
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def hash_function(self, key):
        return key % self.size

    def insert(self, key, value):
        index = self.hash_function(key)
        while self.table[index] is not None:
            index = (index + 1) % self.size  # Linear probing
        self.table[index] = (key, value)

    def get(self, key):
        index = self.hash_function(key)
        while self.table[index] is not None:
            if self.table[index][0] == key:
                return self.table[index][1]
            index = (index + 1) % self.size
        return None  # Not found

# Example usage
hash_table = HashTable(10)
hash_table.insert(1, "Value1")
hash_table.insert(2, "Value2")
print(hash_table.get(1))  # Output: "Value1"
print(hash_table.get(2))  # Output: "Value2"
print(hash_table.get(3))  # Output: None
```

Time Complexity: O(1) on average for insertions and lookups.
Space Complexity: O(n) - For storing the hash table entries.

---

26. Depth-First Search (DFS) for Graphs

Problem: Perform a depth-first search on a graph.

Example Input:

```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': [],
    'F': []
}
```

Starting Vertex: `A`

Expected Output: `['A', 'B', 'D', 'E', 'C', 'F']`

Logic Behind the Algorithm: DFS for graphs visits each vertex recursively:

1. Start from the source node, mark it as visited.
2. Recursively visit each unvisited neighbor.

Code:

```
def dfs_graph(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs_graph(graph, neighbor, visited)

    return visited

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': [],
    'F': []
}
print(dfs_graph(graph, 'A'))  # Output: {'A', 'B', 'C', 'D', 'E', 'F'}
```

Time Complexity: O(V + E)
Space Complexity: O(V) - For the visited set.

27. Breadth-First Search (BFS)

Problem: Perform a breadth-first search on a graph.

Example Input:

```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': [],
    'F': []
}
```

Starting Vertex: `A`

Expected Output: `['A', 'B', 'C', 'D', 'E', 'F']`

Logic Behind the Algorithm: BFS explores nodes level by level:

1. Use a queue to track nodes to visit.
2. Start from the source, enqueue it, and mark it as visited.
3. While the queue is not empty, dequeue a node, process it, and enqueue its neighbors.

Code:

```python
from collections import deque

def bfs_graph(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)
    result = []

    while queue:
        vertex = queue.popleft()
        result.append(vertex)

        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    return result

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': [],
    'F': []
}
print(bfs_graph(graph, 'A'))  # Output: ['A', 'B', 'C', 'D', 'E', 'F']
```

Time Complexity: O(V + E)
Space Complexity: O(V) - For the visited set and the queue.

---

28. String Matching (KMP Algorithm)

Problem: Find all occurrences of a pattern in a text using the Knuth-Morris-Pratt (KMP) algorithm.

Example Input:

```python
text = "ababcababcabc"
pattern = "abc"
```

Expected Output: Indices where the pattern occurs: `[2, 7, 12]`

Logic Behind the Algorithm: KMP preprocesses the pattern to create a longest prefix-suffix (LPS) array:

1. Use the LPS array to skip unnecessary comparisons in the text when a mismatch occurs.
2. Continue searching until the end of the text is reached.

Code:

```python
def KMP_search(text, pattern):
    def compute_lps(pattern):
        lps = [0] * len(pattern)
        length = 0
        i = 1

        while i < len(pattern):
            if pattern[i] == pattern[length]:
                length += 1
                lps[i] = length
                i += 1
            else:
                if length != 0:
                    length = lps[length - 1]
                else:
                    lps[i] = 0
                    i += 1
        return lps

    lps = compute_lps(pattern)
    i = j = 0
    occurrences = []

    while i < len(text):
        if pattern[j] == text[i]:
            i += 1
            j += 1

        if j == len(pattern):
            occurrences.append(i - j)
            j = lps[j - 1]
        elif i < len(text) and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return occurrences

# Example usage
text = "ababcababcabc"
pattern = "abc"
print(KMP_search(text, pattern))  # Output: [2, 7, 12]
```

Time Complexity: O(n + m) - n is the length of the text, m is the length of the pattern.
Space Complexity: O(m) - For the LPS array.

29. Sliding Window Maximum

Problem: Given an array and a number k, find the maximum for each subarray of size k.

Example Input:

```
nums = [1, 3, -1, -3, 5, 3, 6, 7]
k = 3
```

Expected Output: `[3, 3, 5, 5, 6, 7]`

Logic Behind the Algorithm: Use a deque to maintain indices of the maximum elements:

1. Traverse the array, maintaining the deque such that the maximum element's index is at the front.
2. Remove elements that fall out of the current window from the front.
3. Append the current element index and add the maximum to the result.

Code:

```python
from collections import deque

def sliding_window_maximum(nums, k):
    if not nums:
        return []

    result = []
    dq = deque()  # Stores indices

    for i in range(len(nums)):
        # Remove indices that are out of the current window
        if dq and dq[0] < i - k + 1:
            dq.popleft()

        # Remove indices of elements that are less than the current element
        while dq and nums[dq[-1]] < nums[i]:
            dq.pop()

        dq.append(i)  # Add the current element's index

        # Append the maximum for the current window
        if i >= k - 1:
            result.append(nums[dq[0]])

    return result

# Example usage
nums = [1, 3, -1, -3, 5, 3, 6, 7]
k = 3
print(sliding_window_maximum(nums, k))  # Output: [3, 3, 5, 5, 6, 7]
```

Time Complexity: O(n) - Each element is added and removed from the deque at most once.
Space Complexity: O(k) - For the deque.

---

30. Longest Increasing Subsequence

Problem: Find the length of the longest increasing subsequence in an array.

Example Input:

```python
nums = [10, 9, 2, 5, 3, 7,

 101, 18]
```

Expected Output: `4` (The longest increasing subsequence is `[2, 3, 7, 101]`)

Logic Behind the Algorithm: Use dynamic programming to build a solution:

1. Create an array `dp` where `dp[i]` stores the length of the longest increasing subsequence ending at index `i`.
2. For each element, check all previous elements. If the current element is greater, update `dp[i]`.

Code:

```python
def longest_increasing_subsequence(nums):
    if not nums:
        return 0

    dp = [1] * len(nums)

    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)

# Example usage
nums = [10, 9, 2, 5, 3, 7, 101, 18]
print(longest_increasing_subsequence(nums))  # Output: 4
```

Time Complexity: O(n^2) - Nested loops.
Space Complexity: O(n) - For the dp array.

---

31. Merge Intervals

Problem: Given a collection of intervals, merge all overlapping intervals.

Example Input:

```python
intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
```

Expected Output: `[[1, 6], [8, 10], [15, 18]]`

Logic Behind the Algorithm:

1. Sort the intervals by the start time.
2. Initialize a list to hold merged intervals.
3. Iterate through the sorted intervals and merge them if they overlap.

Code:

```
def merge_intervals(intervals):
    if not intervals:
        return []

    intervals.sort(key=lambda x: x[0])  # Sort by start time
    merged = [intervals[0]]

    for current in intervals[1:]:
        last_merged = merged[-1]
        if current[0] <= last_merged[1]:  # Overlap
            last_merged[1] = max(last_merged[1], current[1])
        else:
            merged.append(current)

    return merged

# Example usage
intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
print(merge_intervals(intervals))  # Output: [[1, 6], [8, 10], [15, 18]]
```

Time Complexity: O(n log n) - For sorting.
Space Complexity: O(n) - For the merged list.

---

32. Unique Paths

Problem: Given a m x n grid, find the number of unique paths from the top-left corner to the bottom-right corner.

Example Input:

```
m = 3
n = 7
```

Expected Output: `28`

Logic Behind the Algorithm: Use dynamic programming:

1. Create a 2D array to store the number of unique paths to each cell.
2. Each cell can be reached from the cell above or the cell to the left.

Code:

```python
def unique_paths(m, n):
    dp = [[1] * n for _ in range(m)]  # Initialize with 1s

    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

    return dp[-1][-1]

# Example usage
m = 3
n = 7
print(unique_paths(m, n))  # Output: 28
```

Time Complexity: O(m * n)
Space Complexity: O(m * n) - For the dp array.

---

33. Coin Change

Problem: Given an amount and a list of coin denominations, find the fewest number of coins that make up that amount.

Example Input:

```python
coins = [1, 2, 5]
amount = 11
```

Expected Output: `3` (11 = 5 + 5 + 1)

Logic Behind the Algorithm: Use dynamic programming:

1. Create a dp array where `dp[i]` represents the minimum coins needed for amount `i`.
2. For each coin, update the dp array.

Code:

```python
def coin_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0  # Base case

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1

# Example usage
coins = [1, 2, 5]
amount = 11
print(coin_change(coins, amount))  # Output: 3
```

Time Complexity: O(n * amount)
Space Complexity: O(amount) - For the dp array.

---

34. Valid Parentheses

Problem: Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

Example Input:

```python
s = "({[]})"
```

Expected Output: `True`

Logic Behind the Algorithm: Use a stack:

1. Traverse the string and push opening brackets onto the stack.
2. For closing brackets, check if the stack is not empty and if the top of the stack matches the closing bracket.

Code:

```python
def valid_parentheses(s):
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}

    for char in s:
        if char in mapping:
            top_element = stack.pop() if stack else '#'
            if mapping[char] != top_element:
                return False
        else:
            stack.append(char)

    return not stack

# Example usage
s = "({[]})"
print(valid_parentheses(s))  # Output: True
```

Time Complexity: O(n)
Space Complexity: O(n) - For the stack.

---

35. Maximum Subarray

Problem: Find the contiguous subarray within a one-dimensional array of numbers which has the largest sum.

Example Input:

```
nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
```

Expected Output: `6` (The subarray is `[4, -1, 2, 1]`)

Logic Behind the Algorithm: Use Kadane's Algorithm:

1. Initialize two variables: `current_sum` and `max_sum`.
2. Traverse the array, updating `current_sum` and `max_sum`.

Code:

```
def max_subarray(nums):
    current_sum = max_sum = nums[0]

    for num in nums[1:]:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)

    return max_sum

# Example usage
nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_subarray(nums))  # Output: 6
```

Time Complexity: O(n)
Space Complexity: O(1) - Only constant space used.

---

36. Subarray Sum Equals K

Problem: Given an array of integers and an integer k, return the total number of continuous subarrays whose sum equals to k.

Example Input:

```
nums = [1, 1, 1]
k = 2
```

Expected Output: `2`

Logic Behind the Algorithm: Use a hash map to store cumulative sums:

1. Initialize a cumulative sum and a hash map.
2. Traverse the array, updating the cumulative sum and counting the occurrences.

Code:

```python
def subarray_sum(nums, k):
    count = 0
    cumulative_sum = 0
    sum_map = {0: 1}

    for num in nums:
        cumulative_sum += num
        if cumulative_sum - k in sum_map:
            count += sum_map[cumulative_sum - k]
        sum_map[cumulative_sum] = sum_map.get(cumulative_sum, 0) + 1

    return count

# Example usage
nums = [1, 1, 1]
k = 2
print(subarray_sum(nums, k))  # Output: 2
```

Time Complexity: O(n)
Space Complexity: O(n) - For the hash map.

---

37. Combination Sum

Problem: Given an array of distinct integers and a target, return all unique combinations of numbers that sum up to the target.

Example Input:

```python
candidates = [2, 3, 6, 7]
target = 7
```

Expected Output: `[[7], [2, 2, 3]]`

Logic Behind the Algorithm: Use backtracking to explore all combinations:

1. Iterate through the candidates.
2. For each candidate, recursively call the function with the updated target.

Code:

```python
def combination_sum(candidates, target):
    def backtrack(remaining, combo, start):
        if remaining == 0:
            result.append(list(combo))
            return
        if remaining < 0:
            return

        for i in range(start, len(candidates)):
            combo.append(candidates[i])
            backtrack(remaining - candidates[i], combo, i)  # Not i + 1 because we can reuse
same elements
            combo.pop()

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

# Example usage
candidates = [2, 3, 6, 7]
target = 7
print(combination_sum(candidates, target))  # Output: [[7], [2, 2, 3]]
```

Time Complexity: O(2^n) - In the worst case, all elements can be included.
Space Complexity: O(n) - For the recursion stack.

---

38. Permutations

Problem: Given an array of distinct integers, return all possible permutations.

Example Input:

```python
nums = [1, 2, 3]
```

Expected Output: `[[1, 2,

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

Logic Behind the Algorithm: Use backtracking to generate permutations:

1. Swap each element and recursively generate permutations for the remaining elements.
2. Backtrack by swapping back.

Code:

```
def permute(nums):
    def backtrack(start):
        if start == len(nums):
            result.append(nums[:])  # Make a copy of nums
            return

        for i in range(start, len(nums)):
            nums[start], nums[i] = nums[i], nums[start]  # Swap
            backtrack(start + 1)
            nums[start], nums[i] = nums[i], nums[start]  # Backtrack

    result = []
    backtrack(0)
    return result

# Example usage
nums = [1, 2, 3]
print(permute(nums))  # Output: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2,
1]]
```

Time Complexity: O(n!) - For generating permutations.
Space Complexity: O(n) - For the recursion stack.

---

39. Rotate Image

Problem: Given an n x n 2D matrix representing an image, rotate the image by 90 degrees (clockwise).

Example Input:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Expected Output: `[[7, 4, 1], [8, 5, 2], [9, 6, 3]]`

Logic Behind the Algorithm:

1. Transpose the matrix (swap rows with columns).
2. Reverse each row.

Code:

```python
def rotate(matrix):
    n = len(matrix)

    # Transpose the matrix
    for i in range(n):
        for j in range(i, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]

    # Reverse each row
    for row in matrix:
        row.reverse()

# Example usage
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
rotate(matrix)
print(matrix)  # Output: [[7, 4, 1], [8, 5, 2], [9, 6, 3]]
```

Time Complexity: O(n²) - For transposing and reversing.
Space Complexity: O(1) - In-place rotation.

---

40. Valid Sudoku

Problem: Determine if a 9 x 9 Sudoku is valid, according to the rules of Sudoku.

Example Input:

```python
board = [
    ["5", "3", ".", ".", "7", ".", ".", ".", "."],
    ["6", ".", ".", "1", "9", "5", ".", ".", "."],
    [".", "9", "8", ".", ".", ".", ".", "6", "."],
    ["8", ".", ".", ".", "6", ".", ".", ".", "3"],
    ["4", ".", ".", "8", ".", "3", ".", ".", "1"],
    ["7", ".", ".", ".", "2", ".", ".", ".", "6"],
    [".", "6", ".", ".", ".", ".", "2", "8", "."],
    [".", ".", ".", "4", "1", "9", ".", ".", "5"],
    [".", ".", ".", ".", "8", ".", ".", "7", "9"],
]
```

Expected Output: `True`

Logic Behind the Algorithm:

1. Use sets to track the presence of numbers in each row, column, and 3x3 sub-box.

2. Traverse the board, checking constraints.

Code:

```
def is_valid_sudoku(board):
    rows, cols, boxes = set(), set(), set()

    for i in range(9):
        for j in range(9):
            num = board[i][j]
            if num != '.':
                box_index = (i // 3) * 3 + (j // 3)
                if (i, num) in rows or (j, num) in cols or (box_index, num) in boxes:
                    return False
                rows.add((i, num))
                cols.add((j, num))
                boxes.add((box_index, num))

    return True

# Example usage
board = [
    ["5", "3", ".", ".", "7", ".", ".", ".", "."],
    ["6", ".", ".", "1", "9", "5", ".", ".", "."],
    [".", "9", "8", ".", ".", ".", ".", "6", "."],
    ["8", ".", ".", ".", "6", ".", ".", ".", "3"],
    ["4", ".", ".", "8", ".", "3", ".", ".", "1"],
    ["7", ".", ".", ".", "2", ".", ".", ".", "6"],
    [".", "6", ".", ".", ".", ".", "2", "8", "."],
    [".", ".", ".", "4", "1", "9", ".", ".", "5"],
    [".", ".", ".", ".", "8", ".", ".", "7", "9"],
]
print(is_valid_sudoku(board))  # Output: True
```

Time Complexity: O(1) - Always checking a fixed-size board.
Space Complexity: O(1) - For the sets used.

---

41. Word Search

Problem: Given a 2D board and a word, find if the word exists in the grid.

Example Input:

```
board = [
    ["A", "B", "C", "E"],
    ["S", "F", "C", "S"],
    ["A", "D", "E", "E"]
]
word = "ABCCED"
```

Expected Output: `True`

Logic Behind the Algorithm: Use backtracking to explore all possible paths:

1. For each cell, if it matches the first letter of the word, recursively check adjacent cells for the rest of the word.
2. Mark cells as visited to avoid cycles.

Code:

```python
def exist(board, word):
    def backtrack(x, y, index):
        if index == len(word):
            return True
        if x < 0 or x >= len(board) or y < 0 or y >= len(board[0]) or board[x][y] !=
word[index]:
            return False

        temp = board[x][y]
        board[x][y] = '#'  # Mark as visited

        found = (backtrack(x + 1, y, index + 1) or
                 backtrack(x - 1, y, index + 1) or
                 backtrack(x, y + 1, index + 1) or
                 backtrack(x, y - 1, index + 1))

        board[x][y] = temp  # Restore original value
        return found

    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == word[0] and backtrack(i, j, 0):
                return True

    return False

# Example usage
board = [
    ["A", "B", "C", "E"],
    ["S", "F", "C", "S"],
    ["A", "D", "E", "E"]
]
word = "ABCCED"
print(exist(board, word))  # Output: True
```

Time Complexity: O(m * n * 4^l) - where l is the length of the word, m is the number of rows, and n is the number of columns.
Space Complexity: O(l) - For the recursion stack.

---

42. Lowest Common Ancestor of a Binary Search Tree

Problem: Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes.

Example Input:

```
root = [6, 2, 8, 0, 4, 7, 9, None, None, 3, 5]
p = 2
q = 8
```

Expected Output: `6`

Logic Behind the Algorithm: Utilize the properties of BST:

1. If both `p` and `q` are less than the current node, search in the left subtree.
2. If both are greater, search in the right subtree.
3. If they lie on opposite sides, the current node is the LCA.

Code:

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def lowest_common_ancestor(root, p, q):
    while root:
        if p < root.val and q < root.val:
            root = root.left
        elif p > root.val and q > root.val:
            root = root.right
        else:
            return root
    return None

# Example usage
root = TreeNode(6)
root.left = TreeNode(2)
root.right = TreeNode(8)
root.left.left = TreeNode(0)
root.left.right = TreeNode(4)
root.right.left = TreeNode(7)
root.right.right = TreeNode(9)
root.left.right.left = TreeNode(3)
root.left.right.right = TreeNode(5)

p = 2
q = 8
lca = lowest_common_ancestor(root, p, q)
print(lca.val)  # Output: 6
```

Time Complexity: O(h) - Where h is the height of the tree.
Space Complexity: O(1) - No extra space is used.

---

43. Find Peak Element

Problem: A peak element is an element that is greater than its neighbors. Given an array, find one peak element.

Example Input:

```python
nums = [1, 2, 3, 1]
```

Expected Output: `3` (3 is a peak)

Logic Behind the Algorithm: Use binary search:

1. Check the middle element.
2. If it is greater than its neighbors, return it.
3. If the left neighbor is greater, search the left half.
4. If the right neighbor is greater, search the right half.

Code:

```python
def find_peak_element(nums):
    left, right = 0, len(nums) - 1

    while left < right:
        mid = (left + right) // 2
        if nums[mid] < nums[mid + 1]:
            left = mid + 1
        else:
            right = mid

    return left

# Example usage
nums = [1, 2, 3, 1]
print(find_peak_element(nums))  # Output: 2 (index of peak element 3)
```

Time Complexity: O(log n)
Space Complexity: O(1)

---

44. Number of Islands

Problem: Given a 2D grid of '1's (land) and '0's (water), count the number of islands.

Example Input:

```python
grid = [
    ["1", "1", "0", "0", "0"],
    ["1", "1", "0", "0", "0"],
    ["0", "0", "1", "0", "0"],
    ["0", "0", "0", "1", "1"]
]
```

Expected Output: `3`

Logic Behind the Algorithm: Use DFS:

1. Traverse the grid.
2. When a '1' is found, increment the island count and perform DFS to mark all connected land as visited.

Code:

```
def num_islands(grid):
    if not grid:
        return 0

    def dfs(i, j):
        if i < 0 or i >= len(grid) or j < 0 or j >= len(grid[0]) or grid[i][j] == '0':
            return
        grid[i][j] = '0'  # Mark as visited
        dfs(i + 1, j)
        dfs(i - 1, j)
        dfs(i, j + 1)
        dfs(i, j - 1)

    count = 0
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == '1':
                count += 1
                dfs(i, j)

    return count

# Example usage
grid = [
    ["1", "1", "0", "0", "0"],
    ["1", "1", "0", "0", "0"],
    ["0", "0", "1", "0", "0"],
    ["0", "0", "0", "1", "1"]
]
print(num_islands(grid))  # Output: 3
```

Time Complexity: O(m * n) - Where m is the number of rows and n is the number of columns.
Space Complexity: O(m * n) - In the worst case for the DFS stack.

45. Binary Tree Level Order Traversal

Problem: Given a binary tree, return the level order traversal of its nodes' values.

Example Input:

```
root = [3, 9, 20, None, None, 15, 7]
```

Expected Output: `[[3], [9, 20], [15, 7]]`

Logic Behind the Algorithm: Use a queue to perform BFS:

1. Start with the root node and enqueue it.
2. Process each level, enqueueing child nodes as you go.

Code:

```
from collections import deque

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def level_order(root):
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        level_nodes = []

        for _ in range(level_size):
            node = queue.popleft()
            level_nodes.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(level_nodes)

    return result

# Example usage
root = TreeNode(3)
root.left = TreeNode(9)
root.right = TreeNode(20)
root.right.left = TreeNode(15)
root.right.right = TreeNode(7)

print(level_order(root))  # Output: [[3], [9, 20], [15, 7]]
```

Time Complexity: O(n) - Where n is the number of nodes.
Space Complexity: O(n) - For

the queue.

46. Invert Binary Tree

Problem: Invert a binary tree.

Example Input:

```
root = [4, 2, 7, 1, 3, 6, 9]
```

Expected Output: `[[4], [7, 2], [9, 6, 3, 1]]`

Logic Behind the Algorithm: Use DFS or BFS to swap left and right children at each node.

Code:

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def invert_tree(root):
    if root is None:
        return None

    root.left, root.right = root.right, root.left  # Swap
    invert_tree(root.left)  # Recur on left subtree
    invert_tree(root.right)  # Recur on right subtree

    return root

# Example usage
root = TreeNode(4)
root.left = TreeNode(2)
root.right = TreeNode(7)
root.left.left = TreeNode(1)
root.left.right = TreeNode(3)
root.right.left = TreeNode(6)
root.right.right = TreeNode(9)

invert_tree(root)
# This modifies the tree in place; to see the structure, you'd need to traverse it.
```

Time Complexity: O(n) - Where n is the number of nodes.
Space Complexity: O(h) - For the recursion stack, where h is the height of the tree.

47. Serialize and Deserialize Binary Tree

Problem: Design an algorithm to serialize and deserialize a binary tree.

Example Input:

```
root = [1, 2, 3, None, None, 4, 5]
```

Expected Output: `The tree can be serialized and deserialized correctly.`

Logic Behind the Algorithm: Use pre-order traversal for serialization:

1. Traverse the tree and append values to a list, using 'None' for null nodes.
2. For deserialization, use the list to reconstruct the tree.

Code:

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def serialize(root):
    def recur(node):
        if not node:
            return ['None']
        return [str(node.val)] + recur(node.left) + recur(node.right)

    return ','.join(recur(root))

def deserialize(data):
    def recur(data_list):
        val = data_list.pop(0)
        if val == 'None':
            return None
        node = TreeNode(int(val))
        node.left = recur(data_list)
        node.right = recur(data_list)
        return node

    data_list = data.split(',')
    return recur(data_list)

# Example usage
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.right.left = TreeNode(4)
root.right.right = TreeNode(5)

serialized = serialize(root)
print(serialized)  # Output: "1,2,None,None,3,4,None,None,5,None,None"
deserialized = deserialize(serialized)
```

Time Complexity: O(n) - For both serialization and deserialization.
Space Complexity: O(n) - For the serialized string.

---

48. Group Anagrams

Problem: Given an array of strings, group the anagrams together.

Example Input:

```
strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
```

Expected Output: `[["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]`

Logic Behind the Algorithm: Use a hash map to group strings with the same sorted characters:

1. For each string, sort the characters and use it as a key.
2. Append the original string to the corresponding list in the map.

Code:

```
from collections import defaultdict

def group_anagrams(strs):
    anagrams = defaultdict(list)

    for s in strs:
        key = ''.join(sorted(s))
        anagrams[key].append(s)

    return list(anagrams.values())

# Example usage
strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
print(group_anagrams(strs))  # Output: [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]
```

Time Complexity: O(n * k log k) - Where n is the number of strings and k is the maximum length of a string.
Space Complexity: O(n * k) - For the hash map.

---

49. Kth Largest Element in an Array

Problem: Find the kth largest element in an unsorted array.

Example Input:

```
nums = [3, 2, 1, 5, 6, 4]
k = 2
```

Expected Output: `5`

Logic Behind the Algorithm: Use a min-heap:

1. Maintain a heap of size k.
2. Push elements into the heap; if the size exceeds k, pop the smallest.

Code:

```python
import heapq

def find_kth_largest(nums, k):
    return heapq.nlargest(k, nums)[-1]

# Example usage
nums = [3, 2, 1, 5, 6, 4]
k = 2
print(find_kth_largest(nums, k))  # Output: 5
```

Time Complexity: O(n log k) - For building the heap.
Space Complexity: O(k) - For the heap.

---

50. Pow(x, n)

Problem: Implement pow(x, n), which calculates x raised to the power n.

Example Input:

```
x = 2.00000
n = 10
```

Expected Output: `1024.00000`

Logic Behind the Algorithm: Use exponentiation by squaring:

1. If n is negative, convert the problem to `1/pow(x, -n)`.
2. If n is even, recursively compute `pow(x * x, n // 2)`.
3. If n is odd, compute `x * pow(x, n - 1)`.

Code:

```python
def my_pow(x, n):
    if n < 0:
        x = 1 / x
        n = -n

    def helper(x, n):
        if n == 0:
            return 1
        half = helper(x, n // 2)
        return half * half if n % 2 == 0 else half * half * x

    return helper(x, n)

# Example usage
x = 2.00000
n = 10
print(my_pow(x, n))  # Output: 1024.00000
```

Time Complexity: O(log n)
Space Complexity: O(log n) - For the recursion stack.