DAA Unit 2

1. Disjoint Sets

A **disjoint-set** (or union-find) data structure is used to keep track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

Disjoint Set Operations:

- **Make-Set(x):** Creates a new set containing the element `x`.
- **Find(x):** Returns the representative (or leader) of the set containing `x`. This operation is crucial for determining if two elements are in the same subset.
- **Union(x, y):** Merges the sets containing `x` and `y`. This operation combines two disjoint sets into a single set.

Union-Find Algorithms:

- **Union by Rank:** This optimizes the union operation by attaching the smaller tree under the root of the larger tree.
- **Path Compression:** This optimizes the find operation by making nodes point directly to the root, flattening the structure of the tree.

Time Complexity:

- With union by rank and path compression, the amortized time complexity for each operation (Find and Union) is **O(α(n))**, where `α(n)` is the inverse Ackermann function, which grows very slowly, making the operations almost constant in practice.

---

2. Priority Queue

A **priority queue** is an abstract data type in which each element has a "priority" associated with it. Elements with higher priority are dequeued before elements with lower priority.

Heaps:

A **heap** is a specialized tree-based data structure that satisfies the **heap property**:

- **Max-Heap Property:** The key at a node is greater than or equal to the keys of its children.
- **Min-Heap Property:** The key at a node is less than or equal to the keys of its children.

Common Operations in Heaps:

- **Insert(x):** Adds element `x` to the heap.
- **Extract-Min (Min-Heap) or Extract-Max (Max-Heap):** Removes and returns the element with the minimum (or maximum) key.
- **Heapify:** Ensures that the heap property is maintained after insertion or deletion.

Heapsort:

Heapsort is a comparison-based sorting algorithm that uses a binary heap data structure. The algorithm works by first building a max heap (for descending order) or min heap (for ascending order) from the input data, then repeatedly extracting the root (largest or smallest) element from the heap and reconstructing the heap.

Steps in Heapsort:

1. Build a max heap from the input data.
2. Swap the root of the heap with the last element and reduce the heap size by 1.
3. Heapify the root again to maintain the max heap property.
4. Repeat until all elements are sorted.

Time Complexity:

- Building the heap takes **O(n)**.
- Each extraction and heapify operation takes **O(log n)**, and this is done for **n** elements.
  Thus, the overall time complexity of heapsort is **O(n log n)**.

---

3. Backtracking

Backtracking is a general algorithmic technique used for solving problems incrementally, by building candidates for solutions and abandoning candidates ("backtracking") as soon as it is determined that they cannot possibly lead to a valid solution.

General Method:

1. Define a solution space.
2. Explore the solution space by traversing nodes, which represent partial solutions.
3. At each node, check if it is a valid solution.
4. If not, backtrack by moving to the previous node and trying a different path.

Applications of Backtracking

1. n-Queens Problem

- **Problem**: Place `n` queens on an `n x n` chessboard so that no two queens threaten each other (i.e., no two queens share the same row, column, or diagonal).
- **Approach**: Place queens row by row. If placing a queen on a row leads to no valid position for the next queen, backtrack and try a different placement.

Sudo Algorithm:

```
function solveNQueens(board, row, n):
    if row == n:
        print(board)  // Solution found, print the board
        return

    for col in range(0, n):
        if isSafe(board, row, col, n):
            board[row][col] = 'Q'  // Place queen
            solveNQueens(board, row + 1, n)  // Recur for next row
            board[row][col] = '.'  // Backtrack (remove queen)

function isSafe(board, row, col, n):
    // Check if the column is safe
    for i in range(0, row):
        if board[i][col] == 'Q':
            return False

    // Check upper-left diagonal
    for i, j in range(row, col, -1, -1):
        if board[i][j] == 'Q':
            return False

    // Check upper-right diagonal
    for i, j in range(row, col, -1, +1):
        if board[i][j] == 'Q':
            return False

    return True
```

2. Sum of Subsets Problem

- **Problem**: Given a set of non-negative integers and a value `S`, find all subsets whose elements sum to `S`.
- **Approach**: Explore all subsets by including/excluding each element. Backtrack when the partial sum exceeds `S`.

Sudo Algorithm:

```
C

function subsetSum(arr, n, partial, target, index):
    if sum(partial) == target:
        print(partial)  // Solution found
        return

    if sum(partial) > target or index >= n:
        return  // Exceeded target or exhausted array

    // Include the current element
    subsetSum(arr, n, partial + [arr[index]], target, index + 1)

    // Exclude the current element (backtrack)
    subsetSum(arr, n, partial, target, index + 1)
```

---

3. Graph Coloring

- **Problem**: Assign colors to the vertices of a graph such that no two adjacent vertices have the same color using the minimum number of colors.
- **Approach**: Explore color assignments for each vertex and backtrack when two adjacent vertices have the same color.

Sudo Algorithm:

```
function graphColoring(graph, colors, vertex, n):
    if vertex == n:
        print(colors)  // Solution found
        return True

    for color in range(1, m + 1):  // m is the number of colors
        if isSafe(graph, vertex, colors, color):
            colors[vertex] = color  // Assign color
            if graphColoring(graph, colors, vertex + 1, n):
                return True
            colors[vertex] = 0  // Backtrack

    return False  // No solution

function isSafe(graph, vertex, colors, color):
    for i in range(len(graph)):
        if graph[vertex][i] == 1 and colors[i] == color:
            return False  // Adjacent vertex has the same color
    return True
```

---

4. Hamiltonian Cycle

- **Problem**: A Hamiltonian cycle is a cycle in a graph that visits each vertex exactly once and returns to the starting vertex.
- **Approach**: Explore different paths in the graph, and backtrack if a valid cycle cannot be formed.

Sudo Algorithm:

```
function hamiltonianCycle(graph, path, pos, n):
    if pos == n:
        if graph[path[pos - 1]][path[0]] == 1:
            print(path)  // Solution found
            return True
        else:
            return False

    for vertex in range(1, n):
        if isSafe(vertex, graph, path, pos):
            path[pos] = vertex  // Add vertex to path
            if hamiltonianCycle(graph, path, pos + 1, n):
                return True
            path[pos] = -1  // Backtrack

    return False  // No solution

function isSafe(vertex, graph, path, pos):
    // Check if the vertex is adjacent to the last vertex in the
path
    if graph[path[pos - 1]][vertex] == 0:
        return False

    // Check if the vertex has already been included in the path
    for i in range(pos):
        if path[i] == vertex:
            return False

    return True
```

Backtracking Algorithm for n-Queens:

PYTHON

```python
def is_safe(board, row, col, n):
    # Check this column on upper side
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check upper diagonal on right side
    for i, j in zip(range(row, -1, -1), range(col, n)):
        if board[i][j] == 1:
            return False

    return True

def solve_nqueens(board, row, n):
    if row >= n:
        return True

    for col in range(n):
        if is_safe(board, row, col, n):
            board[row][col] = 1
            if solve_nqueens(board, row + 1, n):
                return True
            board[row][col] = 0

    return False

# Initialize and solve for 4 queens
n = 4
board = [[0] * n for _ in range(n)]
solve_nqueens(board, 0, n)
```