Lecture 1: Introduction and Asymptotic Analysis

- **Factorial (Iterative)**
  - Time Complexity: O(n), Ω(n), Θ(n) (Exact steps: 2n + 2)
  - Space Complexity: O(1), Ω(1), Θ(1) (Exact units: 3 variables)
- **Linear Search (in an array of size n)**
  - Time Complexity:
    - Best Case: O(1) (Element found at the beginning)
    - Worst Case: O(n) (Element not found or found at the end)
    - Overall: O(n), Ω(1)
  - Space Complexity: O(1), Ω(1)
- **Decimal to Binary Conversion (Iterative)**
  - Time Complexity: $O(\log_2 n)$
  - Space Complexity: Not explicitly stated, but involves string concatenation which could be O(log n) depending on implementation.

Lecture 2: Recursion Analysis

- **Factorial (Recursive)**
  - Time Complexity: O(n) (Derived using backward substitution: T(n) = T(n-1) + c)
  - Space Complexity: O(n) (Due to recursion call stack depth)
- **Fibonacci (Recursive)**
  - Time Complexity: $O(2^n)$ (Derived using backward substitution, upper bounded by 2T(n-1)+c)
  - Space Complexity: O(n) (Due to recursion call stack depth)
- **Decimal to Binary Conversion (Recursive)**
  - Time Complexity: $O(\log_2 n)$ (Derived using Master Theorem: T(n) = T(n/2) + c => a=1, b=2, d=0 => Case 2)
  - Space Complexity: $O(\log_2 n)$ (Implicit due to recursion depth, though not explicitly stated in lecture)

- **Euclid's Algorithm (GCD - Iterative/Recursive, Decrease by Variable Amount)**
    - Time Complexity: Not explicitly analyzed in this lecture, but mentioned as an example of decrease-by-variable-amount.
    - Space Complexity: Iterative O(1), Recursive O(log(min(a,b))) due to stack depth (not explicitly stated).

Lecture 3: Brute Force and Greedy Algorithms

- **Traveling Salesman Problem (Brute Force)**
    - Time Complexity: O(n!) (Finds all permutations)
    - Space Complexity: O(n) (To store a permutation, potentially O(n!) to store all if needed, but typically O(n) for processing one)
- **Knapsack Problem (Brute Force)**
    - Time Complexity: O(2^n) (Finds all combinations/subsets)
    - Space Complexity: O(n) (To represent a subset)
- **Selection Sort (Brute Force)**
    - Time Complexity: O(n^2) (Compares all pairs, sum of n+(n-1)+...+1)
    - Space Complexity: O(1) (In-place sort)
- **Traveling Salesman Problem (Greedy - Nearest Neighbor)**
    - Time Complexity: O(n^2) (Selects nearest neighbor at each step)
    - Space Complexity: O(n) (To track visited nodes and the path)
    - Note: Provides a nearly-optimal solution, not guaranteed optimal.
- **Minimum Spanning Tree (MST) - Kruskal's Algorithm (Greedy)**
    - Time Complexity: O(E log E) or O(E log V) depending on implementation (Dominated by sorting edges, E is number of edges, V is number of vertices. Lecture states O(n $\log_2$n) where n=E)
    - Space Complexity: O(E + V) (To store edges and disjoint set data structure)
- **Minimum Spanning Tree (MST) - Prim's Algorithm (Greedy)**
    - Time Complexity: O(E + V log V) with Fibonacci heap, O(V^2) with adjacency matrix (Lecture states O(m*n) = O(n^2) where n=E, m=V, assuming dense graph E $\approx$ V^2)
    - Space Complexity: O(E + V) (To store graph and priority queue/tracking arrays)

Lecture 4: Divide and Conquer

- **Selection Sort (Revisited - Brute Force)**
  - Time Complexity: $O(n^2)$
  - Space Complexity: $O(1)$
- **Merge Sort (Divide and Conquer)**
  - Time Complexity: $O(n \log_2 n)$ (Best and Worst Case, derived from $T(n) = 2T(n/2) + n$ using Master Theorem)
  - Space Complexity: $O(n)$ (Requires extra space for merging)
- **Quick Sort (Divide and Conquer)**
  - Time Complexity:
    - Best Case: $O(n \log_2 n)$ (Pivot partitions evenly, $T(n) = 2T(n/2) + n$)
    - Worst Case: $O(n^2)$ (Pivot partitions unevenly, $T(n) = T(n-1) + n$)
    - Average Case: $O(n \log_2 n)$ (Mentioned as typical/high probability)
  - Space Complexity: $O(\log_2 n)$ (In-place partitioning, stack depth for recursion)

Lecture 5: Non-Comparative Sorting and Transfer & Conquer

- **Counting Sort (Non-Comparative)**
  - Time Complexity: $O(n + k)$ (Best and Worst Case, where n is number of elements, k is the range of input values)
  - Space Complexity: $O(n + k)$ (Requires auxiliary arrays C and O)
  - Note: Stable sort. Not suitable for large ranges, decimals.
- **Radix Sort (Non-Comparative)**
  - Time Complexity: $O(r * (n + k))$ or $O(r * n)$ if k (digits 0-9) is constant (where r is the number of digits/passes, n is number of elements). Lecture states $O(r*n)$.
  - Space Complexity: $O(n + k)$ or $O(n)$ if k is constant (Uses Counting Sort internally). Lecture states $O(n)$.
  - Note: Stable sort (relies on Counting Sort stability). Not effective with floating point numbers.

- **Bucket Sort (Non-Comparative)**
    - Time Complexity:
        - Best Case: O(n + k) (Elements uniformly distributed, k is number of buckets)
        - Worst Case: $O(n^2)$ (All elements fall into one bucket, uses Insertion Sort within buckets)
        - Average Case: O(n) (Assuming uniform distribution)
    - Space Complexity: O(n + k) (For buckets)
- **Binary Search (Used in Transfer & Conquer)**
    - Time Complexity: $O(\log_2 n)$ (Requires sorted array)
    - Space Complexity: O(1) (Iterative), $O(\log_2 n)$ (Recursive stack depth)
- **Uniqueness Problem (Brute Force)**
    - Time Complexity: $O(n^2)$
    - Space Complexity: O(1)
- **Uniqueness Problem (Transfer & Conquer)**
    - Time Complexity: $O(n \log_2 n)$ (Dominated by initial sort - Merge Sort), or $O(n^2)$ if using Insertion Sort (as mentioned in lecture slide, but Merge Sort is stated earlier). Checking adjacent elements is O(n).
    - Space Complexity: O(n) (If using Merge Sort), O(1) (If using in-place sort like HeapSort, or if considering only the checking phase after sorting).
- **Mode Finding (Brute Force)**
    - Time Complexity: $O(n^2)$
    - Space Complexity: O(1)
- **Mode Finding (Transfer & Conquer)**
    - Time Complexity: $O(n \log_2 n)$ (Dominated by initial sort - Merge Sort). Scanning sorted array is O(n).
    - Space Complexity: O(n) (If using Merge Sort), O(1) (If considering only the scanning phase after sorting).

Lecture 6: Backtracking and Complexity Classes

- **Traveling Salesman Problem (Backtracking)**
  - Time Complexity: $O(n!)$ or $O(n^2 * 2^n)$ depending on implementation (Still exponential, but prunes search space compared to pure Brute Force). Lecture notes it's still exponential.
  - Space Complexity: $O(n^2)$ or $O(n)$ depending on implementation (To store state, recursion stack).
- **Sudoku Solver (Brute Force)**
  - Time Complexity: $O(k^n)$ (where k is the number of choices per cell, usually 9, and n is the number of empty cells).
  - Space Complexity: $O(n)$ (To store the board state).
- **Sudoku Solver (Backtracking)**
  - Time Complexity: $O(k^n)$ (Worst case, same as Brute Force, but faster in practice due to pruning).
  - Space Complexity: $O(n)$ (Due to recursion stack depth, n = number of empty cells).
- **Note on Complexity Classes (P, NP, NP-Complete, NP-Hard):** While not algorithms themselves, the lecture discusses these classes in relation to problem solvability and verification times. Problems like Sudoku (decision version) and TSP (decision version) are NP. Optimization versions (find solution/shortest path) are NP-Complete (Sudoku) or NP-Hard (TSP).

Lecture 7: Dynamic Programming

- **Longest Common Subsequence (LCS - Dynamic Programming)**
  - Time Complexity: $O(n*m)$ (where n and m are the lengths of the two sequences)
  - Space Complexity: $O(n*m)$ (To store the DP table)
- **Knapsack Problem (0/1 - Dynamic Programming)**
  - Time Complexity: $O(n*W)$ (where n is the number of items, W is the knapsack capacity)
  - Space Complexity: $O(n*W)$ (To store the DP table)
  - Finding items in the optimal set: $O(n)$ (Backtracking through the DP table)

- **Shortest Path from Single Source - Dijkstra's Algorithm (Dynamic Programming / Greedy)**
    - Time Complexity: $O(V^2)$ with basic implementation/adjacency matrix, $O(E + V \log V)$ with min-priority queue (e.g., Fibonacci heap). (V = vertices, E = edges). Lecture implies V^2 complexity by iterating through U (V nodes) and selecting min distance (V operations) and updating neighbors.
    - Space Complexity: O(V) (To store distances and predecessors in the routing table)

Lecture 8: Evolutionary Algorithms

- **Genetic Algorithm (Evolutionary Strategy - applied to Knapsack)**
    - Time Complexity: Not typically expressed in Big-O notation. Performance depends on parameters like population size, generations, fitness function complexity, and convergence criteria. It's a heuristic approach aiming for near-optimal solutions.
    - Space Complexity: $O(P * n)$ (where P is population size, n is the number of items/genes in a chromosome) to store the population.