Data Structures and Algorithms Study Guide

1. Reversing a Linked List

Using Stack Method

Concept: Push all nodes onto a stack, then pop them to reverse the order.

Example:

- Original: 1 → 2 → 3 → 4 → NULL
- Stack operations: Push 1, 2, 3, 4 → Pop 4, 3, 2, 1
- Result: 4 → 3 → 2 → 1 → NULL

Time Complexity: O(n), **Space Complexity**: O(n)

Other Methods:

1. **Iterative (Three Pointers)**: Use previous, current, and next pointers
2. **Recursive**: Reverse the rest of the list, then fix the current node
3. **Using Array**: Store values in array, rebuild list in reverse

---

2. Priority Queue

Concept: A queue where elements are served based on priority, not arrival time.

Example: Hospital emergency room

- Critical patient (Priority 1) gets treated before
- Moderate patient (Priority 2) who arrived earlier

Implementation: Usually with heaps (min-heap for smallest priority first) **Operations**: Insert O(log n), Extract-Min/Max O(log n)

---

3. Deque (Double-Ended Queue)

Concept: Can insert and delete from both ends.

Example: Browser history

- Add new page at front when navigating forward
- Add at back when going back

- Remove from either end when clearing history

Operations: Insert/Delete at front and rear all O(1)

---

4. Binary Trees Types

Full Binary Tree

Definition: Every node has either 0 or 2 children (no single child)

```
    1
   / \
  2   3
 / \
4   5
```

Complete Binary Tree

Definition: All levels filled except possibly the last, which fills left to right

```
    1
   / \
  2   3
 / \ /
4  5 6
```

Perfect Binary Tree

Definition: All internal nodes have 2 children, all leaves at same level

```
    1
   / \
  2   3
 / \ / \
4  5 6  7
```

---

5. BFS (Breadth-First Search)

Concept: Explore all nodes at current depth before moving to next depth.

Example: Finding shortest path in unweighted graph

- Start from source
- Visit all neighbors first
- Then visit neighbors of neighbors

Implementation: Uses Queue **Time Complexity**: O(V + E) where V = vertices, E = edges

---

6. DFS (Depth-First Search)

Concept: Go as deep as possible before backtracking.

Example: Maze solving

- Choose a path and follow it completely
- When stuck, backtrack and try another path

Implementation: Uses Stack (or recursion) **Time Complexity**: O(V + E)

---

7. Tree Traversals

Inorder (Left → Root → Right)

BST Example:

```
    4
   / \
  2   6
 / \ / \
1  3 5  7
```

Result: 1, 2, 3, 4, 5, 6, 7 (sorted order in BST)

Preorder (Root → Left → Right)

Result: 4, 2, 1, 3, 6, 5, 7 (useful for copying tree)

Postorder (Left → Right → Root)

Result: 1, 3, 2, 5, 7, 6, 4 (useful for deleting tree)

---

8. BST: Sum of Elements ≤ Kth Smallest

Concept: Find Kth smallest element, then sum all elements ≤ that value.

Approach:

1. Do inorder traversal (gives sorted order)
2. Stop at Kth element
3. Sum all elements encountered

Time Complexity: O(k) - only visit k nodes

Example: K=3 in BST [1,2,3,4,5,6,7]

- 3rd smallest = 3
- Sum = 1 + 2 + 3 = 6

---

9. Family Tree as Directed Graph

Concept: Represents family relationships with directed edges.

Example:

```
    John
   /   \
  Mary   Tom
 / \   / \
Sue Bob Lisa Mike
```

Properties:

- No cycles (person can't be ancestor of themselves)
- Directed edges show parent-child relationships
- Can find ancestors, descendants, common ancestors

---

10. Bipartite Graph

Concept: Graph whose vertices can be divided into two disjoint sets where no two vertices within the same set are adjacent.

Example: Job matching

- Set A: People {Alice, Bob, Charlie}
- Set B: Jobs {Engineer, Doctor, Teacher}
- Edges only between sets (people to jobs they can do)

Detection: Color graph with 2 colors using BFS/DFS

---

11. Non-linearity and Memory Allocation

Concept: Non-linear data structures (trees, graphs) don't store elements in sequential memory locations.

Example: Linked List vs Array

- Array: Elements in consecutive memory [100, 104, 108, 112]
- Linked List: Elements scattered, connected by pointers

Advantages: Dynamic size, efficient insertion/deletion **Disadvantages**: No random access, extra memory for pointers

---

12. Dijkstra's Algorithm

Purpose: Find shortest path from source to all vertices in weighted graph.

How it works:

1. Start with source distance = 0, all others = infinity
2. Pick unvisited vertex with minimum distance
3. Update distances to its neighbors
4. Repeat until all vertices visited

Example: Finding shortest route between cities with different road distances.

Pros:

- Guarantees shortest path
- Works with weighted graphs

Cons:

- Doesn't work with negative weights
- Higher time complexity than BFS for unweighted graphs

Time Complexity: $O(V^2)$ with array, $O((V+E)\log V)$ with priority queue

Negative Weight Issue: Algorithm fails because it assumes once a vertex is processed, its shortest distance is finalized.

---

13. Minimum Spanning Tree (MST)

Concept: Subset of edges that connects all vertices with minimum total weight and no cycles.

Example: Connecting cities with minimum cost cables

- Cities = vertices
- Cable costs = edge weights
- MST = cheapest way to connect all cities

Properties:

- Has V-1 edges for V vertices
- No cycles
- Connects all vertices

---

14. Kruskal's MST Algorithm

Steps:

1. Sort all edges by weight
2. Pick smallest edge that doesn't create cycle
3. Repeat until V-1 edges selected

Example: Edges [(1,2,1), (2,3,2), (1,3,3)]

- Pick (1,2,1) - no cycle
- Pick (2,3,2) - no cycle
- Skip (1,3,3) - would create cycle

Time Complexity: O(E log E) due to sorting

---

15. Union-Find Algorithm

Purpose: Efficiently determine if two elements are in same set and merge sets.

Example: Social network friend groups

- Find: Are Alice and Bob in same friend group?
- Union: Merge two friend groups

Operations:

- **Find**: Determine which set element belongs to
- **Union**: Merge two sets

Optimizations:

- Path compression in Find
- Union by rank/size

Time Complexity: Nearly O(1) per operation with optimizations

---

16. Big-O Notation for Trees

Binary Search Tree (Balanced):

- Search: O(log n)
- Insert: O(log n)
- Delete: O(log n)

Binary Search Tree (Unbalanced):

- Worst case: O(n) - becomes like linked list

Complete Binary Tree:

- Height: O(log n)
- Level order traversal: O(n)

General Tree Operations:

- Traversals: O(n)
- Height calculation: O(n)

---

17. Dynamic Programming (DP)

Concept: Solve complex problems by breaking them into overlapping subproblems and storing results.

Key Properties:

1. **Optimal Substructure**: Optimal solution contains optimal solutions to subproblems
2. **Overlapping Subproblems**: Same subproblems solved multiple times

Example: Fibonacci sequence

- F(n) = F(n-1) + F(n-2)
- Without DP: Recalculates F(n-2) multiple times
- With DP: Store F(n-2) result, reuse it

---

18. Longest Common Subsequence (LCS)

Problem: Find longest sequence that appears in both strings in same order.

Example:

- String 1: "ABCDGH"
- String 2: "AEDFHR"
- LCS: "ADH" (length 3)

DP Approach:

- If characters match: 1 + LCS(i-1, j-1)
- If don't match: max(LCS(i-1, j), LCS(i, j-1))

Time Complexity: O(m × n) where m, n are string lengths

---

19. When to Use DP

Indicators:

1. **Overlapping Subproblems**: Same smaller problems solved repeatedly
2. **Optimal Substructure**: Optimal solution built from optimal solutions of subproblems
3. **Optimization Problem**: Finding minimum/maximum value

Examples:

- Fibonacci numbers
- Shortest path problems
- Knapsack problem
- Edit distance
- Matrix chain multiplication

---

20. Memoization (Top-Down)

Concept: Start with original problem, recursively solve subproblems, store results.

Example: Fibonacci with memoization

```
def fib(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

Advantages: Natural recursive structure, only computes needed subproblems

---

21. Tabulation (Bottom-Up)

Concept: Start with smallest subproblems, build up to original problem.

Example: Fibonacci with tabulation

```
def fib(n):
    if n <= 1:
        return n
    dp = [0] * (n+1)
    dp[1] = 1
    for i in range(2, n+1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Advantages: No recursion overhead, better space optimization possible

---

22. Greedy Approach

Concept: Make locally optimal choice at each step, hoping to find global optimum.

Example: Coin change with denominations [1, 5, 10, 25]

- For amount 30: Pick 25 (largest), then 5 (largest remaining)
- Result: 25 + 5 = 30 (2 coins)

When it Works: Problem has greedy choice property **When it Fails**: Local optimum ≠ global optimum

Examples:

- Huffman coding
- Fractional knapsack
- Activity selection
- Dijkstra's algorithm

---

23. Probabilistic Analysis

Concept: Analyze algorithms using probability theory, especially for randomized inputs.

Example: Quicksort analysis

- Best case: $O(n \log n)$ when pivot divides array evenly
- Worst case: $O(n^2)$ when pivot is always smallest/largest
- Average case: $O(n \log n)$ assuming random pivot selection

Uses:

- Average-case analysis
- Expected running time
- Probability of good/bad performance

---

24. Randomized Algorithms

Concept: Algorithm makes random choices during execution.

Types:

1. **Las Vegas**: Always correct, random running time
2. **Monte Carlo**: Random result, fixed running time

Example: Randomized Quicksort

- Randomly choose pivot
- Reduces probability of worst-case $O(n^2)$
- Expected time: $O(n \log n)$

---

25. Hiring Problem

Problem: Hire candidates, pay cost each time you hire someone better than current best.

Example:

- 10 candidates with random skill levels
- Cost incurred when hiring someone better than all previous hires
- Question: What's expected cost?

Analysis: Uses probabilistic analysis

- Expected number of hires: O(log n)
- Each hire has probability 1/i of being better than first i-1 candidates

---

26. Divide and Conquer

Concept:

1. **Divide**: Break problem into smaller subproblems
2. **Conquer**: Solve subproblems recursively
3. **Combine**: Merge solutions to get final answer

Example: Finding maximum in array

- Divide: Split array into two halves
- Conquer: Find max in each half
- Combine: Return larger of the two maxima

Time Complexity: Often O(n log n)

---

27. Merge Sort

How it works:

1. Divide array into halves
2. Recursively sort each half
3. Merge sorted halves

Example: [64, 34, 25, 12, 22, 11, 90]

- Divide: [64,34,25,12] and [22,11,90]
- Keep dividing until single elements
- Merge back: [11,12,22,25,34,64,90]

Time Complexity: O(n log n) in all cases **Space Complexity**: O(n) **Stable**: Yes (maintains relative order of equal elements)

---

28. Quick Sort

How it works:

1. Choose pivot element
2. Partition array: elements < pivot on left, > pivot on right
3. Recursively sort left and right subarrays

Example: [64, 34, 25, 12, 22, 11, 90], pivot = 25

- Partition: [12, 22, 11] 25 [64, 34, 90]
- Recursively sort left and right parts

Time Complexity:

- Best/Average: O(n log n)
- Worst: $O(n^2)$ **Space Complexity**: O(log n) for recursion **Stable**: No

---

29. Maximum Subarray Problem

Problem: Find contiguous subarray with largest sum.

Example: [-2, 1, -3, 4, -1, 2, 1, -5, 4]

- Maximum subarray: [4, -1, 2, 1] with sum 6

Approaches:

1. **Brute Force**: $O(n^3)$ - try all subarrays
2. **Kadane's Algorithm**: O(n) - greedy approach
3. **Divide and Conquer**: O(n log n)

Kadane's Algorithm Idea:

- Keep track of maximum sum ending at current position
- Update global maximum

---

30. Master's Method

Purpose: Analyze time complexity of divide-and-conquer algorithms.

Form: $T(n) = aT(n/b) + f(n)$

- a: number of subproblems
- n/b: size of each subproblem
- f(n): cost of dividing and combining

Three Cases:

1. If $f(n) = O(n^c)$ where $c < \log_b(a)$: $T(n) = O(n^{\log_b(a)})$
2. If $f(n) = O(n^c)$ where $c = \log_b(a)$: $T(n) = O(n^c \log n)$
3. If $f(n) = O(n^c)$ where $c > \log_b(a)$: $T(n) = O(f(n))$

Examples:

- Merge Sort: $T(n) = 2T(n/2) + O(n) \rightarrow O(n \log n)$
- Binary Search: $T(n) = T(n/2) + O(1) \rightarrow O(\log n)$

---

31. Strassen's Algorithm

Purpose: Matrix multiplication faster than standard $O(n^3)$ approach.

Standard Method: Multiply two n×n matrices in $O(n^3)$ time

Strassen's Approach:

- Divide matrices into 2×2 blocks
- Use 7 multiplications instead of 8
- Recursively apply to submatrices

Time Complexity: $O(n^{\log_2 7}) \approx O(n^{2.807})$

Example: For 2×2 matrices

- Standard: 8 multiplications
- Strassen: 7 multiplications + some additions

Trade-off: Faster for large matrices, but higher constant factors and more complex implementation

---

Key Exam Tips

1. **Understand the core concepts** rather than memorizing code
2. **Know when to use each algorithm/data structure**
3. **Remember time and space complexities**
4. **Practice tracing through examples**
5. **Understand the trade-offs** between different approaches
6. **Know the conditions** under which algorithms work best/worst

Focus on understanding the **why** behind each algorithm and data structure - this will help you answer conceptual questions and choose the right approach for given problems.