📚 Detailed Notes — *Data Structures & Algorithms in Python*.

Below is a breakdown by lesson, with deeper detail, key ideas, and some additional insights.

Course Overview & Structure

From the course page:

- The course is **beginner-friendly**, covering data structures (linked lists, stacks, queues, graphs) and algorithms (search, sorting, recursion, DP). (jovian.ai)
- You watch coding-oriented video lectures, practice in Jupyter notebooks, and solve interview-like problems. (jovian.ai)
- There are lessons, corresponding assignments, and a final project. (jovian.ai)
- The lessons are:
  - Binary Search, Linked Lists, Complexity Preview
  - Binary Search Trees, Traversals, Recursion
  - Sorting Algorithms & Divide & Conquer
  - Recursion and Dynamic Programming
  - Graph Algorithms (BFS, DFS, Shortest Paths)
  - Python Interview Questions, Tips & Advice
  - A Project where you solve a problem step-by-step (jovian.ai)
- Assignments are interleaved, e.g.

- Assignment 1: Binary Search Practice
- Assignment 2: Hash Tables & Dictionaries
- Assignment 3: Divide & Conquer Practice
- And so on (jovian.ai)

- The instructor is Aakash N S — background in software engineering, open source, online education. (jovian.ai)

Detailed Lesson-by-Lesson Notes & Core Concepts

Below are deeper notes, sorted by lesson, with key ideas, sample problems, and deeper insight.

Lesson 1: Binary Search, Linked Lists & Complexity Preview

Key Topics:

- Linear Search vs. Binary Search
- Big O notation & complexity theory
- Implementation of Linked Lists using Python classes

Details & insights:

1. **Linear Search**
   - Traverse list one by one, compare each element.
   - Time complexity: `O(N)` in worst case.
   - Simple, works on unsorted data.
2. **Binary Search**

    - Works only on **sorted** arrays/lists.
    - Idea: compare target to middle element, then discard half of the search space.
    - Time complexity: `O(log N)` (base 2)
    - Must carefully handle index boundaries (start, end, mid) and off-by-one errors.

3. **Complexity & Big O**
    - Introduction to **asymptotic analysis**: focus on dominant term, ignore constants and lower-order terms.
    - Compare best, average, worst cases.
    - Space complexity (extra memory use) also matters.

4. **Linked Lists (using Python classes)**
    - Node class: `value` + `next` pointer (or reference).
    - A LinkedList wrapper class might have `head` pointer, methods like `insert`, `delete`, `search`.
    - Advantages: easy insertion/deletion at head, constant time for certain operations (if you have the node).
    - Disadvantages: no direct indexing, so random access is `O(N)`.

Lesson 2: Binary Search Trees, Traversals & Recursion

Key Topics:

- Binary Trees and Binary Search Trees (BST)
- Tree traversal methods
- Recursion fundamentals
- Balanced BSTs and optimizations

Details & insights:

1. **Binary Tree & BST definitions**
   - A **binary tree**: each node has up to two children (left, right).
   - A **BST** (Binary Search Tree): left subtree has smaller keys, right subtree has larger keys.
   - Common operations: insert, search, delete.
2. **Traversals**
   - **Inorder** (Left → Node → Right)
   - **Preorder** (Node → Left → Right)
   - **Postorder** (Left → Right → Node)
   - These can be implemented recursively, or using a stack (iterative).
3. **Recursion**
   - Recursion is when a function calls itself with a smaller/simpler subproblem.
   - Base case(s) are essential to terminate recursion.
   - Many tree algorithms are naturally recursive (e.g. traversal, depth, height).
4. **Balanced BSTs & optimizations**
   - In a degenerate BST (like a linked list), operations degrade to `O(N)`.
   - Balanced BSTs (AVL, Red-Black Trees) keep depth to `O(log N)`.
   - In the context of the course, you may get an introduction to balancing or recognizing when tree degenerates.

Lesson 3: Sorting Algorithms & Divide & Conquer

Key Topics:

- Basic sorting: bubble sort, insertion sort
- Divide & Conquer paradigm
- Merge Sort
- Quick Sort

Details & insights:

1. **Bubble Sort & Insertion Sort**
   - Bubble: repeatedly swap adjacent elements if out of order → `O(N^2)`.
   - Insertion: build sorted portion one by one by inserting into correct spot → also `O(N^2)` in worst case.
2. **Divide & Conquer**
   - A broad algorithmic strategy:
     - Divide the problem into subproblems,
     - Solve them recursively,
     - Combine results.
   - Merge sort is a canonical example.
3. **Merge Sort**
   - Divide the list roughly in half, sort each half, then merge two sorted halves.
   - Time complexity: `O(N log N)` (best, average, worst).
   - Space complexity: `O(N)` extra space (for merging).
4. **Quick Sort**
   - Pick a pivot, partition elements < pivot to one side, > pivot to other side, then recursively sort partitions.

- Average-case time complexity: `O(N log N)`
- Worst-case: `O(N^2)` (e.g. if pivot is always worst)
- In practice, good with randomization or choosing pivots well.
- In-place partitioning helps reduce space overhead.

5. **Other Sorting Considerations**
    - Stable vs. unstable sorting
    - In-place vs. not in-place
    - Hybrid algorithms (e.g. switching to insertion sort for small subarrays)

Lesson 4: Recursion & Dynamic Programming

Key Topics:

- Deep dive on recursion & memoization
- Subsequence problems
- 0/1 Knapsack
- Backtracking & pruning

Details & insights:

1. **Recursion & Memoization**
    - Pure recursion may recompute the same subproblem multiple times.
    - Memoization (caching intermediate results) saves time.
    - Top-down (recursive + memo) vs bottom-up (DP table) approaches.
2. **Subsequence Problems**

- Example: *Longest Common Subsequence (LCS)*
- If the last characters match, you reduce both indices. Otherwise, consider skipping one character from one string.
- Use DP to avoid repeated work.

3. **0/1 Knapsack**
   - You have items (weight, value) and a capacity.
   - Recurrence: for each item, either take it (if fits) or skip it, and pick better of two options.
   - Use DP (2D table) so that the complexity becomes `O(n * capacity)`.
4. **Backtracking & Pruning**
   - Backtracking: try all possibilities (e.g. for combinatorial problems) with recursion.
   - Pruning: stop exploring a branch when you know it can't yield a better solution (e.g. bounding).
   - Useful for subset sum, permutations, combinations, etc.

Lesson 5: Graph Algorithms (BFS, DFS & Shortest Paths)

Key Topics:

- Graph representation (adjacency list, adjacency matrix)
- BFS & DFS
- Shortest path algorithms (especially Dijkstra)
- Directed graphs & weights

Details & insights:

1. **Graph Representations**

- **Adjacency List**: for each node, a list (or dict) of neighbors → efficient for sparse graphs.
- **Adjacency Matrix**: 2D matrix, `graph[u][v] = cost` or boolean → good for dense graphs or fast edge lookups.

2. **BFS (Breadth-First Search)**
   - Uses a **queue**
   - Visits nodes in “layers” from a starting node
   - Good for shortest path (in unweighted graphs)
   - Time complexity: `O(V + E)` where `V` = nodes, `E` = edges
3. **DFS (Depth-First Search)**
   - Uses recursion or explicit stack
   - Goes deep, then backtracks
   - Useful for exploring connectivity, detecting cycles, topological sort
   - Time complexity: also `O(V + E)`
4. **Shortest Paths in Weighted Graphs**
   - For non-negative weights, **Dijkstra’s algorithm** is classical.
   - Using a **priority queue** (min-heap) yields a complexity like `O((V + E) log V)`.
   - Key idea: maintain tentative distance to each node, repeatedly pick the undiscovered node with smallest distance, relax its edges.
5. **Directed Graphs & Variations**
   - Graphs may have directed edges (u → v but not v → u).
   - Problems like detecting cycles, strongly connected components, topological sort may be introduced (or at least touched on).

Lesson 6: Python Interview Questions, Tips & Advice

Key Topics:

- Real interview-style problems with solutions
- Coding challenge strategies
- Time management & thinking process
- Tips on writing clean, testable code

Details & insights:

- Emphasis likely on practicing common patterns: sliding window, two pointers, hash-based counting, recursion, DP
- Advice on debugging, writing test cases, reading the problem carefully
- Presenting your solution, articulating complexity, edge cases

---

Project & Assignments

- After lessons and assignments, a **project** pushes you to pick a non-trivial problem, break it down, and document step-by-step solution. (jovian.ai)
- Assignments reinforce each module (e.g. implementing hash tables from scratch, divide & conquer problems, graph problems). (jovian.ai)
- You earn a certificate by completing weekly assignments. (jovian.ai)

Additional Depth & Extended Insights

Here are some deeper points and expansions (beyond what might be strictly in the course) that tie into the course's topics, which may help you understand more fully.

On Complexity — Amortized Analysis & Constant Factors

- Sometimes operations average out over many calls (e.g. dynamic array resizing).
- Big-O ignores constant multipliers, but in real settings, constant factors and lower-order terms can matter in practice.
- Also consider worst-case vs average-case performance.

On Hash Tables & Collisions

- When designing a hash table from scratch, collision resolution strategies include:
  - **Chaining** (each bucket holds a list of entries)
  - **Open addressing** (linear probing, quadratic probing, double hashing)
- Resizing (rehashing) is important: when load factor exceeds a threshold, expand the table and rehash all entries.

On Balanced Trees & Alternatives

- Beyond basic BST, balanced trees like **AVL**, **Red-Black Trees**, **Treaps**, **Splay Trees** ensure the height remains `O(log N)` under insertions & deletions.
- Sometimes **heaps** (binary heap, Fibonacci heap) are used for priority queue tasks, particularly in graph algorithms or scheduling.

On Advanced Graph Algorithms (beyond Dijkstra)

- **Bellman-Ford** handles negative weights (no negative cycles).
- **Floyd-Warshall** computes all-pairs shortest paths.
- **Johnson's algorithm** (for sparse graphs)
- **Minimum Spanning Tree (MST)**: Prim's and Kruskal's algorithms.
- **Topological sort**, **strongly connected components** (Kosaraju's, Tarjan's)

- **Flow algorithms** (e.g. Edmonds-Karp) etc.

On More Dynamic Programming Patterns

- Common pattern types:
    - **Knapsack / subsets / combinatorial optimization**
    - **Longest increasing subsequence (LIS)**
    - **Edit distance / string alignment**
    - **Matrix chain multiplication**
    - **DP on trees / graphs**
- Recognizing overlapping subproblems and optimal substructure is key.

On Space Optimizations & Trade-offs

- In DP, sometimes only a few previous rows are needed (so we can reduce 2D table to 1D).
- In recursion, careful about stack depth; tail recursion vs iteration.
- Some “optimized” algorithms trade time for space or vice versa.

On Coding Best Practices

- Write modular functions (small pieces)
- Use assertions / unit tests
- Document time / space complexity in comments
- Consider edge cases: empty input, single element, maximum input, invalid input
- Use version control (Git) to track changes & experiment safely
- Use a test suite (like Jovian’s) to automate verification

Suggested Additional Examples & Exercises

To cement understanding, here are extra ideas to try (on your own):

1. **Implement a Hash Map from scratch**
   - Use open addressing or chaining
   - Support `put(key, value)`, `get(key)`, `delete(key)`
2. **Find k-th smallest element in an unsorted array**
   - Using sorting vs. using a selection algorithm (like Quickselect)
3. **Longest Palindromic Subsequence / Substring**
   - Use DP or expand-around-center techniques
4. **Graph problem**: shortest path on a grid with obstacles (e.g. 0/1 weights)
   - Use BFS or Dijkstra depending on weights
5. **Interview-style problem**: "given a string, find the minimum window substring that contains all characters of another string" — typical sliding window + hash table problem.
6. **Compare different sorting algorithms in practice**
   - Measure runtimes on random arrays of increasing size
   - Compare merge sort, quicksort, insertion sort on various data distributions (random, sorted, reversed)