

Swayam

Introduction to Algorithms and Analysis

Prof. Sourav Mukhopadhyay | IIT Kharagpur

VARUN KUMAR
02-11-2025

Table of Contents

Week – 01: Sorting Problem, Time Complexity, and Asymptotic Analysis	3
Lecture 01: Insertion Sort.....	3
Lecture 02: Analysis of Insertion Sort	4
Lecture 03: Asymptotic Notation	5
Lecture 04: Recurrence for Merge Sort.....	6
Lecture 05: Substitution Method	8
Week – 02: Solving Recurrence, Divide and Conquer.....	9
Lecture 06: The Master Method	9
Week – 03: Quick Sort and Heap Sort, Decision Tree	10
Week – 04: Linear Time Sorting, Order Statistic	11
Week – 05: Hash Function, Binary Search Tree (BST) Sort.....	12
Week – 06: Randomly Build BST, Red Black Tree, Augmentation of Data Structure	13
Week – 07: Van Emde Boas, Amortized Analysis, Computational Geometry.....	14
Week – 08: Dynamic Programming, Graph, Prim's Algorithm.....	15
Week – 09: BFS & DFS, Shortest Path Problem, Dijkstra, Bellman-Ford	16
Week – 10: All Pair Shortest Path, Floyd-Warshall, Jhonson Algorithm	17
Week – 11: More Amortized Analysis, Disjoint Set Data Structure	18
Week – 12: Network Flow, Computational Complexity	19
Appendix – 01: Test of 2025.....	20
Week – 01.....	20
Appendix – 02: Important Links	21
Appendix – 03: Chat GPT and Deep Seek	22
Insertion Sort.....	22
Whimsical Diagrams	26
Sorting Technique	26
Divide and Conquer Algorithm	27
Greedy Algorithm.....	28
Dynamic Programming.....	29
Appendix – 04: Python Setup Guide	31
Appendix – 05: Step-by-Step Guide of Various Algorithm with Python Code	34
Appendix – 06: Working with Graph using NetworkX	35

Appendix – 07: Essential Problems from CLRS.....	40
--	----

Week – 01: Sorting Problem, Time Complexity, and Asymptotic Analysis

Lecture 01: Insertion Sort

Topics to be Covered: -

- Problem of Sorting, Pseudo Code,
- Insertion Sort, Loop Invariant, Runtime, Parameterise the runtime by the size of the input

The Problem of Sorting

Input:- A sequence of $\langle a_1, a_2, \dots, a_n \rangle$ of numbers

Output:- A permutation of $\langle a'_1, a'_2, \dots, a'_n \rangle$ such that

$$a'_1 \leq a'_2 \leq \dots \leq a'_n$$

Example

Input: 9 3 5 0 4 7

Output: 0 3 4 5 7 9

Pseudo Code:- Insertion Sort

Insertion Sort(A,n):

for $j \leftarrow 1$ to n :

do $key \leftarrow A[i]$

$i \leftarrow j-1$

while $i > 0$ & $A[i] > key$

do $A[i+1] \leftarrow A[i]$

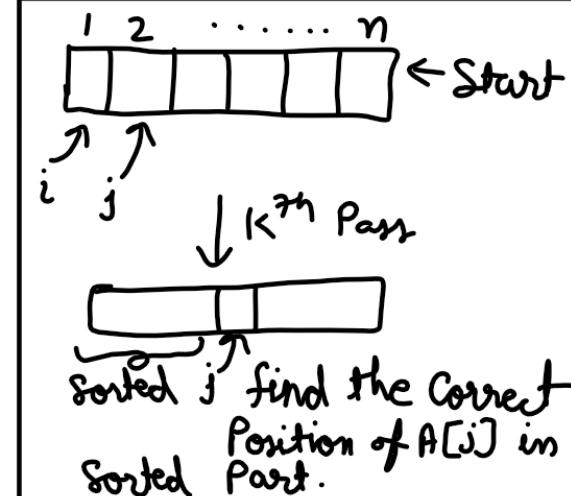
$i \leftarrow i-1$

$A[i+1] = key$

Do a Dry Run of the code.

Example of Insertion Sort

8, 2, 4, 9, 3, 6
 2, 8, 4, 9, 3, 6
 2, 4, 8, 9, 3, 6
 2, 3, 4, 8, 9, 6
 2, 3, 4, 6, 8, 9

**★ Runtime of Insertion Sort**

- The running time depends on input i.e., already sorted sequence is easier to sort
- Parameterize the running time by the size of the input, since short sequence are easier to sort than long one.
- Generally, we seek upper bounds on running time.

Lecture 02: Analysis of Insertion Sort**Topics to be Covered: -**

- Types of Analysis: Worst Case, Best Case and Average Case, Machine Independence
- Asymptotic Notation, Big-Theta Notation (θ)

★ Types of Analysis

- **Worst Case (Usually) :-**
 - $T(n)$ = Maximum time of algorithm on any input of size ' n '.
- **Average Case (Sometimes) :-**
 - $T(n)$ = Expected time of algorithm on any input of size ' n '.
- **Best Case :-**
 - Cheat with a slow algorithm that works fast on 'some' input.

★ Machine-Independent Time

What is Insertion Sort's worst-case time?

- It depends on the speed of the computer
 - o Relative Speed (on the same machine)
 - o Absolute Speed (on different machine)

★ Big Idea

Ignore machine-dependent constants.

Look at 'growth' of $T(n)$ as $n \rightarrow \infty$

ASYMPTOTIC ANALYSIS

★ Θ Notation

Maths:-

- $\Theta(g(n)) = \{f(n) : \exists \text{ positive constant } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0\}$

Engineering:-

- Drop low order terms; ignore leading constants

Example:-

$$3n - 90n + 5n - 1024 = \Theta(n^3)$$

Lecture 03: Asymptotic Notation

Topics to be Covered: -

- Asymptotic Notation: - Big-Oh, Big-Theta, and Big-Omega
- Time Complexity of Insertion Sort: - Worst Case, Best Case, and Average Case
- Merge Sort

★ O Notation

$$O(g(n)) = \{f(n) : \exists \text{ positive constant } c \text{ and } n_0 \text{ such that } f(n) \leq c * g(n) \forall n \geq n_0\}$$

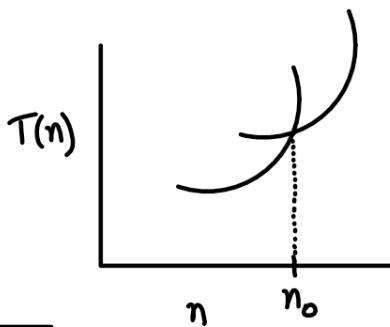
★ Ω Notation

$$\Omega(g(n)) = \{f(n) : \exists \text{ positive constant } c \text{ and } n_0 \text{ such that } f(n) \geq c * g(n) \forall n \geq n_0\}$$

★ Asymptotic Notation

When n gets large enough a $\theta(n^2)$ algorithm always beats a $O(n^3)$ algorithm.

- We shouldn't ignore asymptotically slower algorithm.
- Real world design situations often calls for a careful balancing of engineering objectives.
- It is a useful tool to help structure our thinking.



★ Insertion Sort Analysis

- **Worst Case:** Input Inversely sorted.

$$T(n) = \sum_{j=2}^n \theta(j) = \theta(n^2) \text{ [Arithmetic Series]}$$

- **Average Case:** All permutation equally likely.

$$T(n) = \sum_{j=2}^n \theta\left(\frac{j}{2}\right) = \theta(n^2)$$

- It is moderately fast for small ' n '.
- It is not at all fast for large ' n '.

Lecture 04: Recurrence for Merge Sort

Topics to be Covered: -

- Merge Sort, Run time of Merge Sort
- Recurrence and Recursive Tree

Merge Sort

MERGE-SORT A[1....n]

To sort n numbers

1. If $n=1$, done
2. Recursively Sort $A[1 \dots n/2]$ and $A[\lceil n/2 \rceil + 1 \dots n]$
3. Merge the two sorted lists

Key Sub-Routine : MERGE

↳ Time = $\Theta(n)$ for n input

★ Analysis of Merge Sort

MERGE-SORT $A[1 \dots n]$

$T(n)$	To sort n numbers
$\Theta(1)$	1. If $n=1$, done
$\Theta(n/2)$	2. Recursively Sort $A[1 \dots n/2]$ and $A[\lceil n/2 \rceil + 1 \dots n]$
$\Theta(n)$	3. Merge the two sorted lists

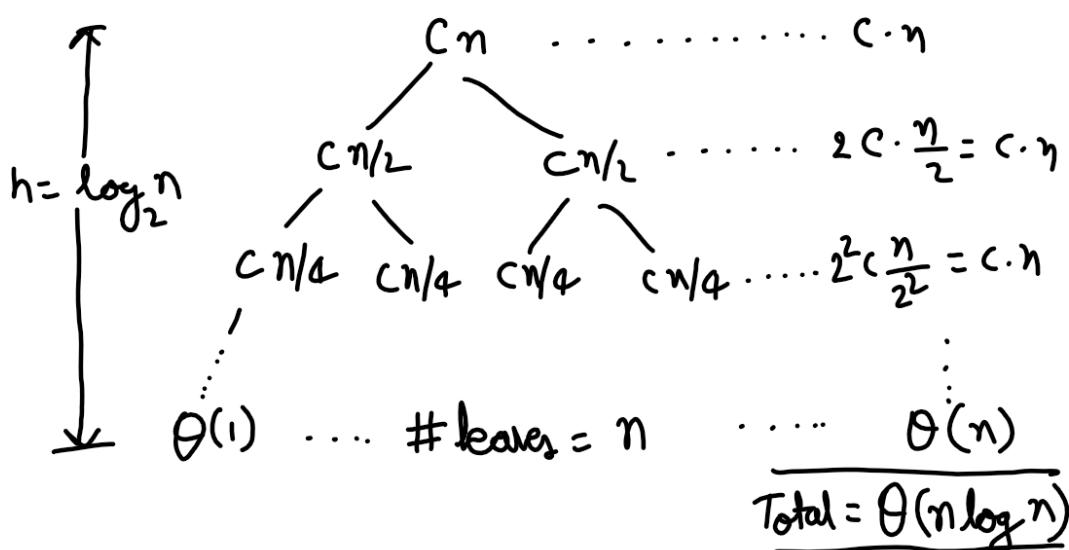
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$$

★ Recurrence for Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2) + \Theta(n) & \text{if } n>1 \end{cases}$$

We shall usually omit the base case when for sufficiently small ' n ' and when it has no effect on the solution to the recurrence

★ Recursion Tree



★ Best case of Merge Sort



Not Inplace
↳ because
of merge
Sub-Routine

$$T(n) = 2T(n/2) + \Theta(n) \quad [\text{always}]$$

$$= \Theta(n \log n)$$

Lecture 05: Substitution Method

Topics to be Covered: -

- Solving the Recurrence: Substitution Method
- Method of Induction

It is the most general method:

- Guess the form of solution
- Verify by Induction
- Solve for constants

$$\begin{aligned} T(n) &= 4T(n/2) + n \leq 4C(n/2)^3 + n \\ &= (C/2)n^3 + n = \underbrace{Cn^3 - ((C/2)n^3 - n)}_{\text{desired - residual}} \leq Cn^3 \end{aligned}$$

↑ ↑
desired residual desired

Whenever $((C/2)n^3 - n) \geq 0$ if $C \geq 2, n \geq 1$

↑
Residual

Week – 02: Solving Recurrence, Divide and Conquer

Lecture 06: The Master Method

Week – 03: Quick Sort and Heap Sort, Decision Tree

Week – 04: Linear Time Sorting, Order Statistic

Week – 05: Hash Function, Binary Search Tree (BST) Sort

Week – 06: Randomly Build BST, Red Black Tree, Augmentation of Data Structure

Week – 07: Van Emde Boas, Amortized Analysis, Computational Geometry

Week – 08: Dynamic Programming, Graph, Prim's Algorithm

Week – 09: BFS & DFS, Shortest Path Problem, Dijkstra, Bellman-Ford

Week – 10: All Pair Shortest Path, Floyd-Warshall, Jhonson Algorithm

Week – 11: More Amortized Analysis, Disjoint Set Data Structure

Week – 12: Network Flow, Computational Complexity

Appendix – 01: Test of 2025

Week – 01

Appendix – 02: Important Links

Current: - https://onlinecourses.nptel.ac.in/noc25_cs150/preview

2023: - https://onlinecourses.nptel.ac.in/noc23_cs88/preview

NPTEL: - <https://nptel.ac.in/courses/106105164>

Appendix – 03: Chat GPT and Deep Seek

Insertion Sort

◆ Insertion Sort Explained

Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time. It's similar to how you might sort playing cards in your hands: you pick one card at a time and place it in the correct position among the already sorted cards.

💡 How It Works (Step-by-Step)

1. Start from the second element (index 1), because a single element is already "sorted".
2. Compare the current element with the elements before it.
3. Shift the larger elements one position to the right.
4. Insert the current element into the correct position.
5. Repeat for all elements.

💡 Example

Let's sort the array:

[5, 3, 4, 1, 2]

Step-by-step:

Step	Array State	Explanation
1	[5, 3, 4, 1, 2]	Start from index 1 (3)
2	[3, 5, 4, 1, 2]	Insert 3 before 5
3	[3, 4, 5, 1, 2]	Insert 4 between 3 and 5
4	[1, 3, 4, 5, 2]	Insert 1 at the beginning
5	[1, 2, 3, 4, 5]	Insert 2 between 1 and 3

✓ Final Sorted Array: [1, 2, 3, 4, 5]

Time Complexity

Case	Time Complexity
Best Case	$O(n)$ (already sorted)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$ (reverse sorted)

```
# Python Code for Insertion Sort
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Move elements greater than key to one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key

# Example
arr = [5, 3, 4, 1, 2]
insertion_sort(arr)
print("Sorted array:", arr)
```

Walk through the **step-by-step logic to code Insertion Sort**, assuming you understand loops, conditionals, and lists.

Step-by-Step Plan to Code Insertion Sort

Let's say we want to sort the list `arr = [5, 3, 4, 1, 2]`.

Step 1: Understand the Goal

We want to sort `arr` in ascending order using the insertion sort technique.

Step 2: Loop Through the Array

Start the loop from the second element (index 1), because the first element is trivially sorted.

```
for i in range(1, len(arr)):
```

Step 3: Store the Current Value

Pick the current value (to be inserted into the sorted part on the left).

```
key = arr[i]
```

Step 4: Set Pointer for Comparison

Set a pointer j to the element just before i. We'll use this to compare with the elements on the left side.

```
j = i - 1
```

Step 5: Shift Elements Greater Than key

While j is not out of bounds and the current left-side element is greater than key, shift it to the right.

```
while j >= 0 and arr[j] > key:  
    arr[j + 1] = arr[j] # shift right  
    j -= 1 # move left
```

Step 6: Insert the key at the Correct Position

Now insert the key at $j + 1$ because the last checked position (j) was either out of bounds or smaller than key.

```
arr[j + 1] = key
```

✓ Step 7: Full Code Together

See above

✓ Step 8: Test It

```
arr = [5, 3, 4, 1, 2]
insertion_sort(arr)
print("Sorted array:", arr)
```

Output:

Sorted array: [1, 2, 3, 4, 5]

⌚ Optional Tip for You

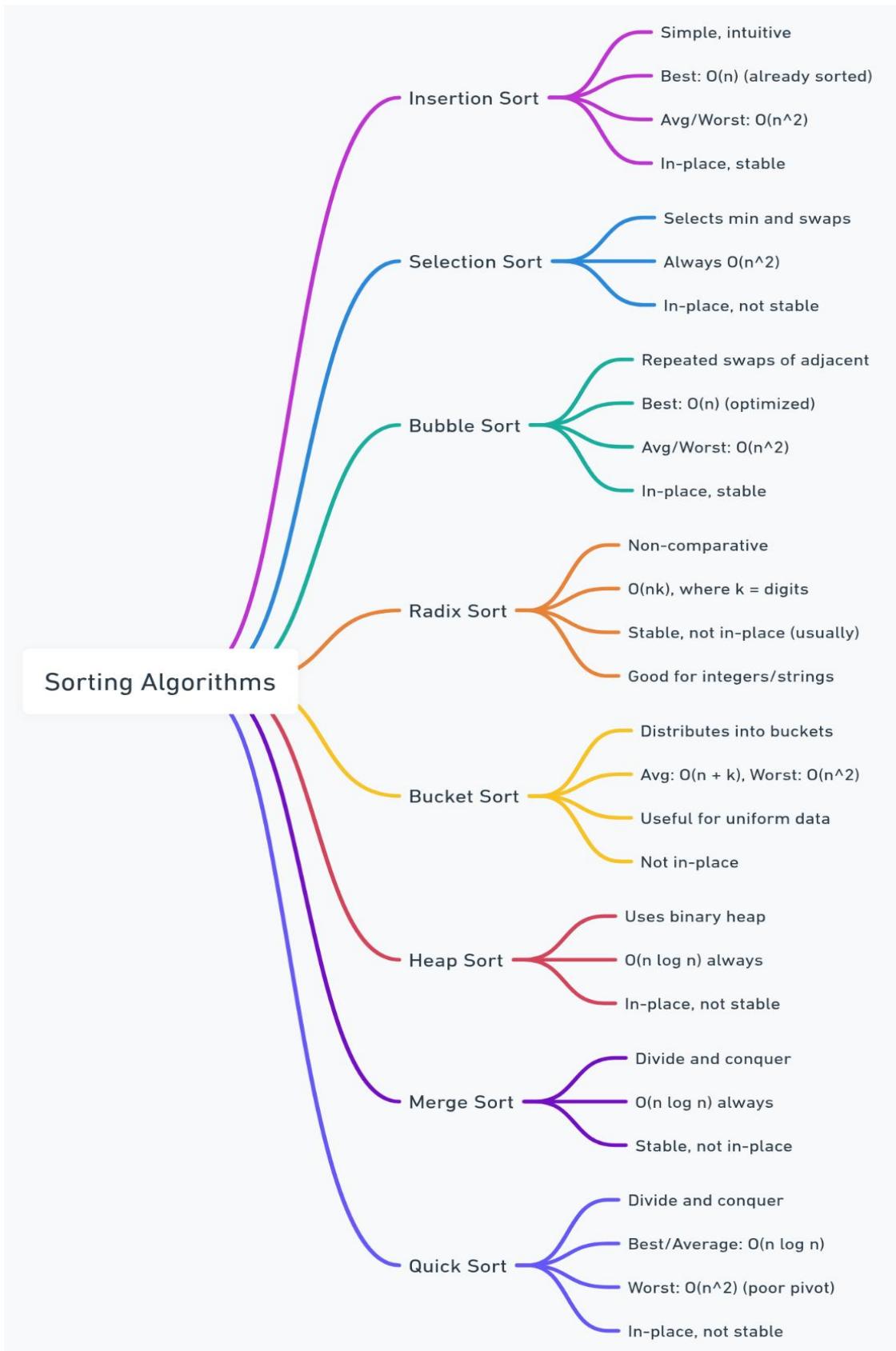
Add `print()` statements to see what's happening at each step:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        print(f"\nInserting {key}")
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
            print(f"Moved {arr[j + 1]} to position {j + 2}")
        arr[j + 1] = key
        print(f"Inserted {key} at position {j + 1}")
    print("Current array state:", arr)
```

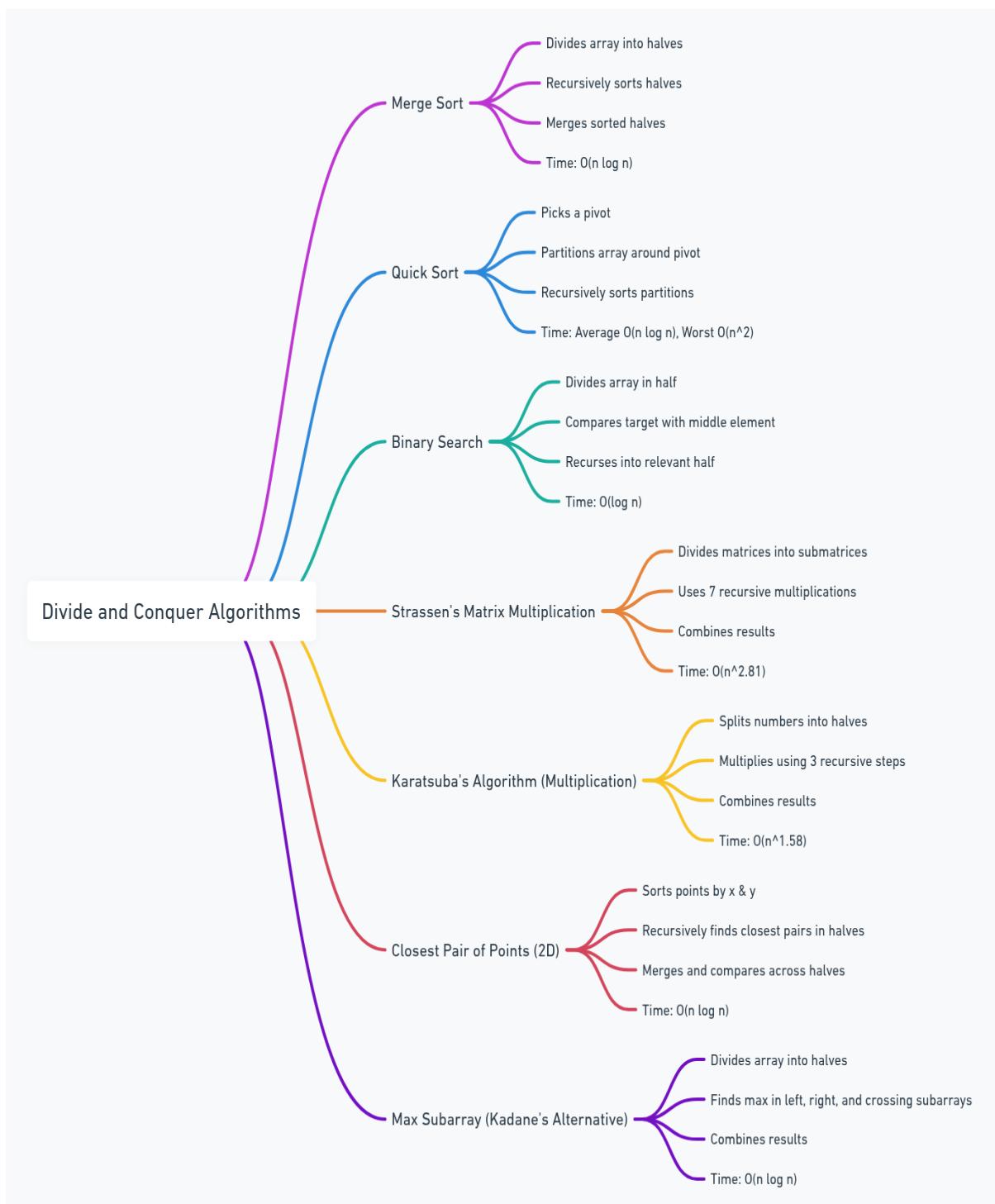
Create Insertion Sort in Decreasing order.

Whimsical Diagrams

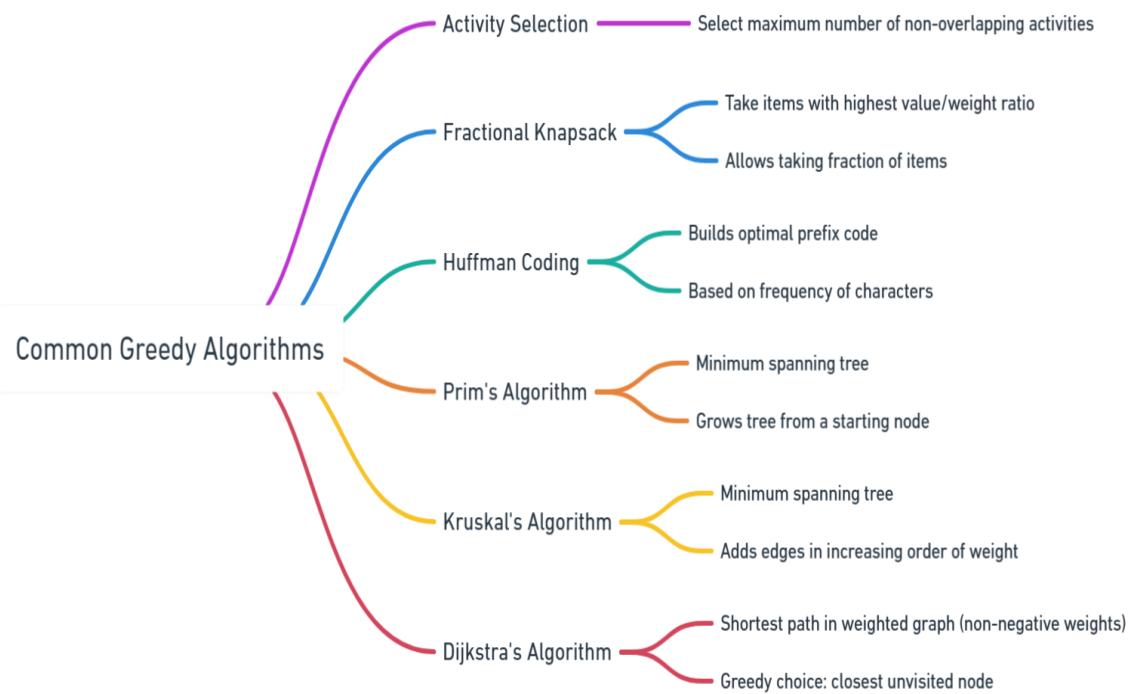
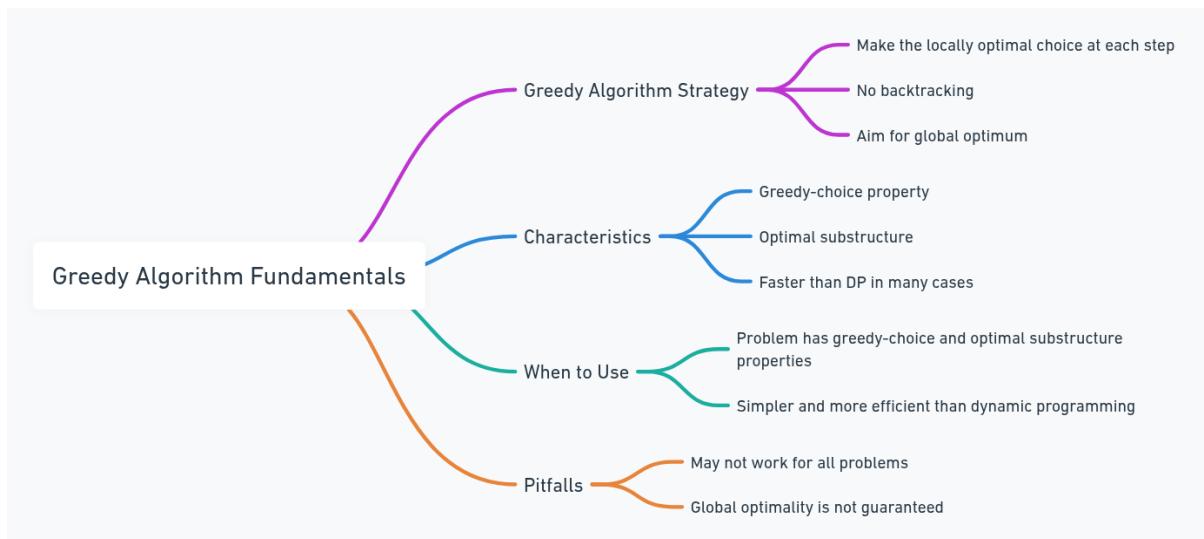
Sorting Technique



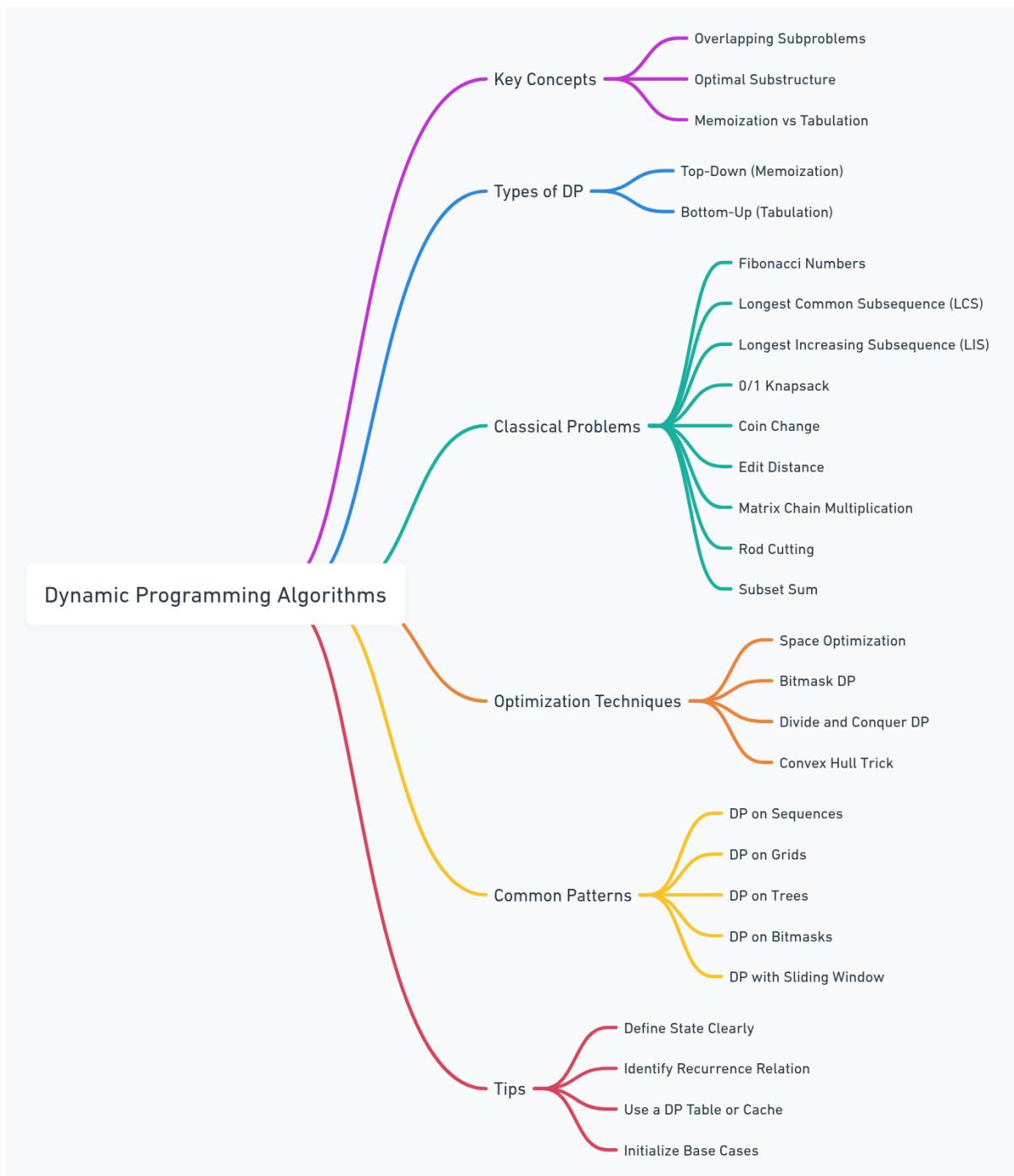
Divide and Conquer Algorithm

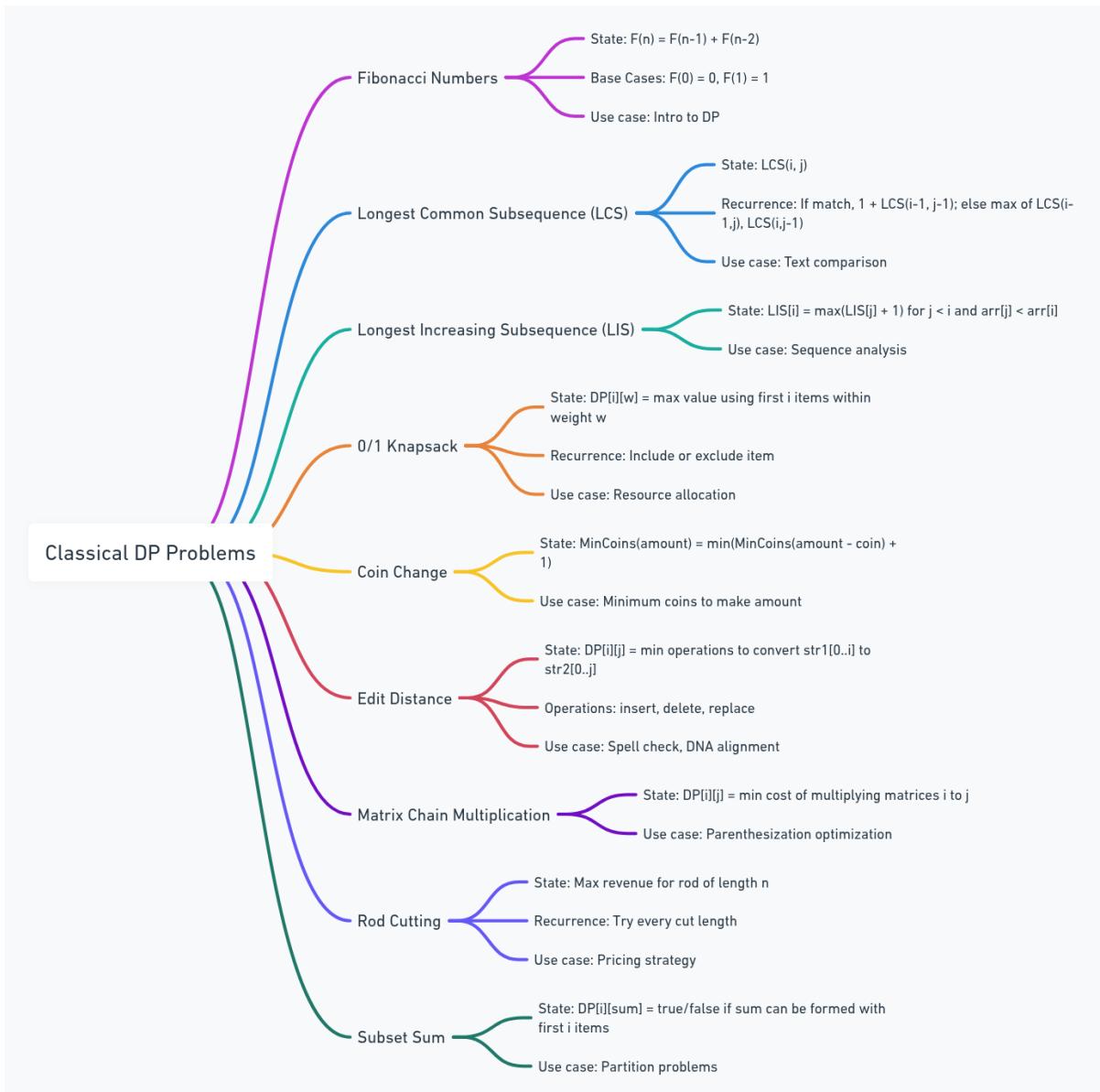


Greedy Algorithm



Dynamic Programming





Appendix – 04: Python Setup Guide

1. Installing Python

◆ Windows:

1. Download the installer from: <https://www.python.org/downloads>
2. Run the installer:
 - Check “Add Python to PATH”
 - Click **Install Now**
3. Verify installation:

```
python -version
```

◆ Linux (Ubuntu/Debian):

```
sudo apt update  
sudo apt install python3 python3-pip python3-venv  
  
# Verify Installation  
python -version
```

2. Create a Virtual Environment

◆ Windows:

```
python -m venv algo-env  
.algo-env\Scripts\activate
```

◆ Linux:

Bash

```
python3 -m venv algo-env  
source algo-env/bin/activate
```

Deactivate with:

```
deactivate
```

 **3. Install Essential Packages**

Once the virtual environment is activated, install these recommended packages:

```
pip install jupyter matplotlib numpy pandas networkx rich
```

 **Purpose of These Packages:**

<u>Package</u>	<u>Purpose</u>
<i>Jupyter</i>	Run notebooks for interactive coding
<i>matplotlib</i>	Visualization and plotting
<i>numpy</i>	Numerical computing
<i>pandas</i>	Data structures and manipulation
<i>networkx</i>	Graph theory & algorithm practice
<i>rich</i>	Beautiful CLI output (optional)

 **4. Launching Jupyter Notebook**

In your project folder:

```
jupyter notebook
```

A browser will open. You can create ‘.ipynb’ files and run code cells interactively — great for learning and testing algorithms.

You can create a ‘requirements.txt’ to share your setup:

```
pip freeze > requirements.txt
```

 **6. Optional Tools (Highly Recommended)**

Tool	Use Case	Install Command
<i>black</i>	Auto-code formatter	<code>pip install black</code>
<i>pytest</i>	Testing algorithms	<code>pip install pytest</code>
<i>ipython</i>	Enhanced interactive shell	<code>pip install ipython</code>
<i>pygraphviz</i>	Advanced graph visualizations	see note below

 **!** *pygraphviz* may require additional system libraries:

On Ubuntu:

```
sudo apt install graphviz libgraphviz-dev
```

Appendix – 05: Step-by-Step Guide of Various Algorithm with Python Code

Appendix – 06: Working with Graph using NetworkX

(<https://chatgpt.com/c/68490260-0394-800c-a581-9d6389235c43>)

Introduction

Graphs are fundamental in computer science and algorithm design. They model relationships between entities—like cities on a map, web pages, social networks, and more.

This appendix introduces [NetworkX](https://networkx.org/)(<https://networkx.org/>)—a Python library for creating, manipulating, and visualizing complex networks—to help you **experiment interactively** with graph algorithms while learning them.

Why NetworkX is Useful

- Easy to create and visualize graphs
- Supports directed, undirected, weighted, and multigraphs
- Built-in implementations of many classic graph algorithms
- Useful for both **learning concepts** and **experimenting interactively**

Here are some examples, please develop some more by yourself.

1. Creating Graphs

```
python
import networkx as nx

# Undirected Graph
G = nx.Graph()
G.add_edges_from([('A', 'B'), ('B', 'C'), ('C', 'D'), ('A', 'D')])

# Directed Graph
DG = nx.DiGraph()
DG.add_weighted_edges_from([('A', 'B', 5), ('B', 'C', 2), ('A', 'C', 9)])
```

You can also use `MultiGraph()` or `MultiDiGraph()` for multigraphs (with parallel edges).

2. Breadth-First Search (BFS)

```
python

import networkx as nx

G = nx.Graph()
G.add_edges_from([(0, 1), (0, 2), (1, 3), (2, 4)])

# BFS traversal from node 0
bfs_edges = list(nx.bfs_edges(G, source=0))
bfs_nodes = list(nx.bfs_tree(G, source=0))

print("BFS traversal:", bfs_edges)
```

 Copy  Edit

Use:

Explore all reachable nodes from a starting point in layers.

3. Depth-First Search (DFS)

```
python

dfs_edges = list(nx.dfs_edges(G, source=0))
print("DFS traversal:", dfs_edges)
```

 Copy  Edit

Use:

Used in solving puzzles, topological sorting, cycle detection, etc.

4. Dijkstra's Shortest Path

```
python

DG = nx.DiGraph()
DG.add_weighted_edges_from([
    ('A', 'B', 2),
    ('A', 'C', 5),
    ('B', 'C', 1),
    ('C', 'D', 3)
])

path = nx.dijkstra_path(DG, source='A', target='D')
length = nx.dijkstra_path_length(DG, source='A', target='D')

print("Shortest path:", path)
print("Path length:", length)
```

 Copy  Edit

 **Use:**

Finds the shortest path in weighted graphs with non-negative weights.

 **5. Cycle Detection**

python

[Copy](#) [Edit](#)

```
try:
    cycle = nx.find_cycle(G)
    print("Cycle found:", cycle)
except nx.exception.NetworkXNoCycle:
    print("No cycle found")
```

 **Use:**

Useful in detecting infinite loops or deadlocks.

 **6. Topological Sorting**

python

[Copy](#) [Edit](#)

```
DAG = nx.DiGraph()
DAG.add_edges_from([('cook', 'eat'), ('shop', 'cook'), ('study', 'pass')])

order = list(nx.topological_sort(DAG))
print("Topological Order:", order)
```

 **Note:**

Only works on Directed Acyclic Graphs (DAGs).

7. Graph Visualization

python

[Copy](#) [Edit](#)

```
import matplotlib.pyplot as plt

G = nx.Graph()
G.add_weighted_edges_from([('A', 'B', 4), ('A', 'C', 1), ('C', 'D', 2)])

pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=2000, font_size=14)
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.show()
```

Use:

Visual intuition is critical for understanding how graphs behave.

Summary Table

Task	Function(s)
Create Graph	<code>Graph()</code> , <code>DiGraph()</code> , <code>add_edge()</code>
BFS	<code>bfs_edges()</code> , <code>bfs_tree()</code>
DFS	<code>dfs_edges()</code>
Shortest Path	<code>dijkstra_path()</code> , <code>dijkstra_path_length()</code>
Cycle Detection	<code>find_cycle()</code>
Topological Sorting	<code>topological_sort()</code>
Visualization	<code>draw()</code> , <code>draw_networkx_edge_labels()</code>

Other Use Cases:

Algorithm / Problem	networkx Feature Used	
Cycle detection	<code>nx.find_cycle()</code>	
DFS/BFS	<code>nx.dfs_edges()</code> / <code>nx.bfs_edges()</code>	
PageRank	<code>nx.pagerank()</code>	
Topological Sort	<code>nx.topological_sort()</code>	
Graph Coloring	Custom coloring or external libs	
Minimum Spanning Tree	<code>nx.minimum_spanning_tree()</code>	

Example Use Case: **Shortest Path using Dijkstra's Algorithm**

Let's say you want to compute the shortest path between two cities on a road map.

```
import networkx as nx
import matplotlib.pyplot as plt

# Step 1: Create a directed weighted graph
G = nx.DiGraph()

# Step 2: Add edges (node1, node2, weight)
G.add_weighted_edges_from([
    ('A', 'B', 4),
    ('A', 'C', 2),
    ('B', 'C', 5),
    ('B', 'D', 10),
    ('C', 'D', 3)
])

# Step 3: Compute shortest path from A to D
path = nx.dijkstra_path(G, source='A', target='D')
length = nx.dijkstra_path_length(G, source='A', target='D')

print("Shortest path:", path)
print("Path length:", length)

# Step 4: Visualize the graph
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='skyblue',
        node_size=2000, font_size=16)
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
plt.show()
```

Appendix – 07: Essential Problems from CLRS

Essential Problems from CLRS (by Chapter)

Chapter 2: Getting Started

- Insertion Sort (2.1)
 - Merge Sort – including loop invariants (2.3)
 - Binary Search (Exercise 2.3-5)
 - Inversions in an array (Problem 2-4)
-

Chapter 3: Growth of Functions

- Asymptotic notation comparison problems
 - Exercise 3.1-1 to 3.1-6 – proving O , Θ , and Ω relationships
 - Exercise 3.2-3 – use of limits in asymptotic behaviour
-

Chapter 4: Divide-and-Conquer

- Maximum Subarray Problem (4.1)
 - Recurrence Tree Method and Master Theorem (4.3)
 - Strassen's Matrix Multiplication (4.2)
-

Chapter 6: Heapsort

- Build-Max-Heap and Max-Heapify (6.3)
 - Implement Priority Queue with Heap
 - Median maintenance with two heaps (advanced)
-

Chapter 7–8: Quicksort & Sorting Lower Bounds

- Randomized Quicksort (7.3)
- Worst-case for Quicksort (Problem 7-1)
- Counting Sort and Radix Sort (8.2, 8.3)

- Lower bounds for comparison sorts (8.1)
-

Chapter 9: Medians and Order Statistics

- Randomized-Select algorithm (9.2)
 - Deterministic Select – Median of Medians (9.3)
-

Chapter 10–11: Elementary Data Structures & Hashing

- Stack, Queue, Linked List operations (10.1–10.3)
 - Hash Table with chaining and open addressing (11.2–11.4)
 - Universal hashing (11.3)
-

Chapter 12–13: Binary Search Trees & Red-Black Trees

- In-order Traversal
 - Search, Min, Max, Successor, Predecessor (12.2)
 - Insert and Delete in BST
 - Red-Black Tree Insertion & Deletion (13.3)
-

Chapter 15: Dynamic Programming

- Matrix Chain Multiplication (15.2)
 - Longest Common Subsequence (15.4)
 - Rod Cutting (15.1)
 - Optimal BST (15.5, advanced)
-

Chapter 16: Greedy Algorithms

- Activity Selection Problem (16.1)
 - Huffman Coding (16.3)
 - Fractional Knapsack (Problem 16-1)
-

Chapter 22–24: Graph Algorithms

- BFS and DFS (22.2, 22.3)
 - Topological Sort (22.4)
 - Strongly Connected Components (22.5)
 - Dijkstra's Algorithm (24.3)
 - Bellman-Ford Algorithm (24.1)
 - Floyd-Warshall Algorithm (25.2)
 - Minimum Spanning Trees: Prim's and Kruskal's (23.1, 23.2)
-

Chapter 26–27: Max Flow

- Ford-Fulkerson Algorithm (26.2)
- Bipartite Matching using flow (26.1, 26.3)
- Push-Relabel Algorithm (27.2)