Introduction to Python Algorithms

- A course by Aditya Saxena

Welcome to the course: **Mastering Algorithms with Python**

In this course, we explore the most important algorithms every Python developer should know—organized by domain. This course will blend theory with hands-on Python implementations to develop both understanding and skill.

Course Objectives:

- Understand algorithmic problem-solving.
- Implement core algorithms in Python.
- Apply algorithms to real-world problems in various domains.

Covered Domains:

1. Sorting and Searching
2. Graph Algorithms
3. Dynamic Programming
4. Machine Learning Algorithms
5. String Algorithms
6. Tree Algorithms
7. Computational Geometry
8. Greedy Algorithms
9. Backtracking Algorithms
10. Cryptographic Algorithms

```python
# Common imports
import math
import random
import heapq
import hashlib
from collections import deque, defaultdict
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from typing import List, Tuple, Dict, Set

print("Environment ready for algorithmic exploration!")

# Utility: Timer for performance checking
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Executed in {end - start:.6f} seconds")
        return result
    return wrapper

# Placeholder for exploration
print("\nYou're all set! Start by opening any of the following notebooks as we go deeper:")
print("1. Sorting_Searching.ipynb")
print("2. Graph_Algorithms.ipynb")
print("3. Dynamic_Programming.ipynb")
print("4. Machine_Learning.ipynb")
print("5. String_Algorithms.ipynb")
print("6. Tree_Algorithms.ipynb")
print("7. Computational_Geometry.ipynb")
print("8. Greedy_Algorithms.ipynb")
print("9. Backtracking_Algorithms.ipynb")
print("10. Cryptographic_Algorithms.ipynb")
```

```
Environment ready for algorithmic exploration!

You're all set! Start by opening any of the following notebooks as we go deeper:
1. Sorting_Searching.ipynb
2. Graph_Algorithms.ipynb
3. Dynamic_Programming.ipynb
4. Machine_Learning.ipynb
5. String_Algorithms.ipynb
6. Tree_Algorithms.ipynb
7. Computational_Geometry.ipynb
8. Greedy_Algorithms.ipynb
9. Backtracking_Algorithms.ipynb
10. Cryptographic_Algorithms.ipynb
```

1. Sorting and Searching

Quick Sort – Efficient, divide-and-conquer sorting algorithm.

Merge Sort – Stable sort using recursion and merging.

Binary Search – Fast element lookup in sorted arrays.

📘 Quick Sort

🔧 **How It Works:** Quick Sort sorts a list by choosing a pivot element, then splitting the list into two parts—elements less than the pivot and those greater than or equal to it. It then recursively sorts each part and combines them with the pivot in between. This approach breaks the problem into smaller pieces until everything is sorted.

⚙️ **Performance:**

- Best case: O(n log n)
- Average case: O(n log n)
- Worst case: O(n²) (when pivot is poorly chosen)
- Space: O(log n) for recursion stack (in-place sort)

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = [x for x in arr[1:] if x < pivot]
    right = [x for x in arr[1:] if x >= pivot]
    return quick_sort(left) + [pivot] + quick_sort(right)
```

```python
import time

# Test Case 1
arr1 = [5, 3, 8, 4, 2]
start = time.time()
sorted_arr1 = quick_sort(arr1)
end = time.time()
print("Input:", arr1)
print("Sorted:", sorted_arr1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: [5, 3, 8, 4, 2] is sorted to [2, 3, 4, 5, 8]

# Test Case 2
arr2 = [1, 1, 1, 1]
start = time.time()
sorted_arr2 = quick_sort(arr2)
end = time.time()
print("\nInput:", arr2)
print("Sorted:", sorted_arr2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: All elements are equal, so output remains [1, 1, 1, 1]
```

```
Input: [5, 3, 8, 4, 2]
Sorted: [2, 3, 4, 5, 8]
Time taken: 0.000099 seconds

Input: [1, 1, 1, 1]
```

```
Sorted: [1, 1, 1, 1]
Time taken: 0.000055 seconds
```

📘 Merge Sort

🔧 **How It Works:**

Merge Sort is a recursive algorithm that splits a list into two halves, sorts each half, and then **merges** the sorted halves into a single sorted list. This continues until each sublist has only one element (which is already sorted), and then they are combined in sorted order. It's very stable and performs consistently well, even on large datasets.

⚙️ **Performance:**

- Best case: O(n log n)
- Average case: O(n log n)
- Worst case: O(n log n)
- Space: O(n) (extra space for merging)

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

```python
import time

# Test Case 1
arr1 = [9, 5, 1, 3, 7]
start = time.time()
sorted_arr1 = merge_sort(arr1)
end = time.time()
print("Input:", arr1)
print("Sorted:", sorted_arr1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: List is sorted to [1, 3, 5, 7, 9]

# Test Case 2
arr2 = [10]
start = time.time()
sorted_arr2 = merge_sort(arr2)
end = time.time()
print("\nInput:", arr2)
print("Sorted:", sorted_arr2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Single-element list stays the same: [10]
```

```
Input: [9, 5, 1, 3, 7]
Sorted: [1, 3, 5, 7, 9]
Time taken: 0.000073 seconds
```

```
Input: [10]
Sorted: [10]
Time taken: 0.000048 seconds
```

📘 Binary Search

🔧 **How It Works:**

Binary Search finds an element in a **sorted list** by repeatedly dividing the search space in half. It compares the target value to the middle element; if they are not equal, it discards the half where the target cannot be. This continues until the element is found or the list is empty.

⚙️ **Performance:**

- Best case: O(1)
- Average case: O(log n)
- Worst case: O(log n)
- Space: O(1) (iterative) or O(log n) (recursive)

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid  # Found the target at index mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1  # Target not found
```

```
import time

# Test Case 1
arr1 = [1, 3, 5, 7, 9]
target1 = 5
start = time.time()
result1 = binary_search(arr1, target1)
end = time.time()
print("Input array:", arr1)
print("Target:", target1)
print("Result (Index):", result1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: 5 is found at index 2

# Test Case 2
arr2 = [2, 4, 6, 8, 10]
target2 = 7
start = time.time()
result2 = binary_search(arr2, target2)
end = time.time()
print("\nInput array:", arr2)
print("Target:", target2)
print("Result (Index):", result2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: 7 is not in the array, so result is -1
```

```
Input array: [1, 3, 5, 7, 9]
Target: 5
Result (Index): 2
Time taken: 0.000060 seconds

Input array: [2, 4, 6, 8, 10]
Target: 7
Result (Index): -1
Time taken: 0.000049 seconds
```

2. Graph Algorithms

Dijkstra’s Algorithm – Shortest path in weighted graphs.

Depth-First Search (DFS) – Explore as deep as possible.

Breadth-First Search (BFS) – Level-order exploration.

📘 Dijkstra’s Algorithm

🔧 **How It Works:**
Dijkstra’s Algorithm finds the shortest path from a **starting node** to all other nodes in a graph with **non-negative weights**. It uses a priority queue to always explore the nearest unvisited node, updating the shortest known distances as it progresses, until the shortest paths to all nodes are found.

⚙️ **Performance:**

- Best case: O(V + E log V) (with min-heap)
- Average case: O((V + E) log V)
- Worst case: O(V^2) (with adjacency matrix)
- Space: O(V) for distances and visited sets

```python
import heapq

def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    priority_queue = [(0, start)]  # (distance, node)

    while priority_queue:
        current_dist, current_node = heapq.heappop(priority_queue)

        # Skip if we've already found a better path
        if current_dist > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node]:
            distance = current_dist + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances



import time

# Test Case 1
graph1 = {
    'A': [('B', 1), ('C', 4)],
    'B': [('C', 2), ('D', 5)],
    'C': [('D', 1)],
    'D': []
}
start_node1 = 'A'

start = time.time()
distances1 = dijkstra(graph1, start_node1)
end = time.time()

print("Graph:", graph1)
print("Start node:", start_node1)
print("Shortest distances:", distances1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Shortest paths from A to others are computed using edge weights

# Test Case 2
graph2 = {
    0: [(1, 2), (2, 4)],
    1: [(2, 1), (3, 7)],
    2: [(4, 3)],
    3: [(5, 1)],
    4: [(3, 2), (5, 5)],
    5: []
}
start_node2 = 0

start = time.time()
distances2 = dijkstra(graph2, start_node2)
end = time.time()

print("\nGraph:", graph2)
print("Start node:", start_node2)
print("Shortest distances:", distances2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Calculates shortest distances from node 0 to all others
```

```
Graph: {'A': [('B', 1), ('C', 4)], 'B': [('C', 2), ('D', 5)], 'C': [('D', 1)], 'D': []}
Start node: A
Shortest distances: {'A': 0, 'B': 1, 'C': 3, 'D': 4}
Time taken: 0.000092 seconds

Graph: {0: [(1, 2), (2, 4)], 1: [(2, 1), (3, 7)], 2: [(4, 3)], 3: [(5, 1)], 4: [(3, 2), (5, 5)], 5: []}
Start node: 0
Shortest distances: {0: 0, 1: 2, 2: 3, 3: 8, 4: 6, 5: 9}
Time taken: 0.000076 seconds
```

📘 Depth-First Search (DFS)

🔧 **How It Works:**
DFS explores a graph by starting at a source node and going **as deep as possible** along each branch before backtracking. It uses a stack (or recursion) to keep track of the path. DFS is useful for tasks like detecting cycles, topological sorting, and connected components.

⚙️ **Performance:**

- Best case: O(V + E)
- Average case: O(V + E)
- Worst case: O(V + E)
- Space: O(V) for visited nodes (plus recursion stack)

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

    return visited
```

```
import time

# Test Case 1
graph1 = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
}
start_node1 = 'A'

start = time.time()
visited1 = dfs(graph1, start_node1)
end = time.time()

print("Graph:", graph1)
print("Start node:", start_node1)
print("Visited nodes:", visited1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: DFS visits nodes in depth order: A → B → D → C → E

# Test Case 2
graph2 = {
    1: [2, 3],
    2: [4],
    3: [],
    4: [5],
    5: []
}
start_node2 = 1

start = time.time()
visited2 = dfs(graph2, start_node2)
end = time.time()

print("\nGraph:", graph2)
print("Start node:", start_node2)
print("Visited nodes:", visited2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: DFS visits nodes starting from 1: 1 → 2 → 4 → 5 → 3 (though order may vary)
```

```
Graph: {'A': ['B', 'C'], 'B': ['D'], 'C': ['E'], 'D': [], 'E': []}
Start node: A
Visited nodes: {'B', 'D', 'C', 'A', 'E'}
Time taken: 0.000151 seconds

Graph: {1: [2, 3], 2: [4], 3: [], 4: [5], 5: []}
Start node: 1
Visited nodes: {1, 2, 3, 4, 5}
Time taken: 0.000089 seconds
```

📘 Breadth-First Search (BFS)

🔧 **How It Works:**

BFS explores a graph **level by level**, starting from a source node. It uses a **queue** to visit all neighboring nodes before moving to the next level. BFS is ideal for finding the shortest path in **unweighted graphs** and exploring all nodes reachable from a starting point.

⚙️ **Performance:**

- Best case: O(V + E)
- Average case: O(V + E)
- Worst case: O(V + E)
- Space: O(V) for the queue and visited list

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

    return visited
```

```
import time

# Test Case 1
graph1 = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
}
start_node1 = 'A'

start = time.time()
visited1 = bfs(graph1, start_node1)
end = time.time()

print("Graph:", graph1)
print("Start node:", start_node1)
print("Visited nodes (BFS order):", visited1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: BFS visits A → B → C → D → E (level by level)

# Test Case 2
graph2 = {
    1: [2, 3],
    2: [4],
    3: [],
    4: [5],
    5: []
}
start_node2 = 1

start = time.time()
visited2 = bfs(graph2, start_node2)
end = time.time()

print("\nGraph:", graph2)
print("Start node:", start_node2)
print("Visited nodes (BFS order):", visited2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: BFS visits 1 → 2 → 3 → 4 → 5 in level-order fashion
```

```
Graph: {'A': ['B', 'C'], 'B': ['D'], 'C': ['E'], 'D': [], 'E': []}
Start node: A
Visited nodes (BFS order): {'B', 'D', 'C', 'A', 'E'}
Time taken: 0.000067 seconds

Graph: {1: [2, 3], 2: [4], 3: [], 4: [5], 5: []}
Start node: 1
Visited nodes (BFS order): {1, 2, 3, 4, 5}
Time taken: 0.000054 seconds
```

3. Dynamic Programming

Longest Common Subsequence (LCS) – Compare sequences efficiently.

0/1 Knapsack – Optimize value under weight constraints.

Fibonacci Sequence – Classic example of overlapping subproblems.

📘 Longest Common Subsequence (LCS)

🔧 **How It Works:**
LCS finds the longest sequence that appears in the **same order** (not necessarily consecutively) in two strings. It uses **dynamic programming** to build a table of subproblem results, where each entry represents the LCS length for prefixes of the two strings. It builds the solution bottom-up by comparing characters and storing the maximum result so far.

⚙️ **Performance:**

- Best case: O(m × n)
- Average case: O(m × n)
- Worst case: O(m × n)
- Space: O(m × n) (can be optimized to O(min(m, n)) with rolling arrays)

```
def lcs(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    # Fill the DP table
    for i in range(m):
        for j in range(n):
            if s1[i] == s2[j]:
                dp[i + 1][j + 1] = dp[i][j] + 1
            else:
                dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j])

    return dp[m][n]
```

```
import time

# Test Case 1
s1_1 = "ABCBDAB"
s2_1 = "BDCAB"
start = time.time()
length1 = lcs(s1_1, s2_1)
end = time.time()
print("String 1:", s1_1)
print("String 2:", s2_1)
print("LCS length:", length1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: LCS is "BCAB" or "BDAB", both length 4

# Test Case 2
s1_2 = "AGGTAB"
s2_2 = "GXTXAYB"
start = time.time()
length2 = lcs(s1_2, s2_2)
end = time.time()
print("\nString 1:", s1_2)
print("String 2:", s2_2)
print("LCS length:", length2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: LCS is "GTAB", length 4
```

```
String 1: ABCBDAB
String 2: BDCAB
LCS length: 4
Time taken: 0.000086 seconds

String 1: AGGTAB
String 2: GXTXAYB
LCS length: 4
Time taken: 0.000084 seconds
```

📘 0/1 Knapsack Problem

🔧 **How It Works:**

The 0/1 Knapsack Problem aims to select a subset of items with given **weights** and **values** to **maximize the total value** without exceeding the weight limit of the knapsack. Each item can either be included or excluded (0 or 1). It uses **dynamic programming** to build a table where each entry represents the best value for a given weight capacity and subset of items.

⚙️ **Performance:**

- Best case: O(n × W)
- Average case: O(n × W)
- Worst case: O(n × W)
- Space: O(n × W) (can be optimized to O(W) with 1D array)

```
# ✅ Python Implementation

def knapsack_01(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(
                    dp[i - 1][w],  # exclude item
                    dp[i - 1][w - weights[i - 1]] + values[i - 1]  # include item
                )
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1
weights1 = [1, 2, 3]
values1 = [10, 15, 40]
capacity1 = 6

start = time.time()
max_value1 = knapsack_01(weights1, values1, capacity1)
end = time.time()
print("Weights:", weights1)
print("Values:", values1)
print("Capacity:", capacity1)
print("Maximum value:", max_value1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Optimal to take all items; total value = 65

# Test Case 2
weights2 = [2, 3, 4, 5]
values2 = [3, 4, 5, 6]
capacity2 = 5

start = time.time()
max_value2 = knapsack_01(weights2, values2, capacity2)
end = time.time()
print("\nWeights:", weights2)
print("Values:", values2)
print("Capacity:", capacity2)
print("Maximum value:", max_value2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Best to take item with weight 2 and value 3, and item with weight 3 and value 4 → total value = 7
```

```
Weights: [1, 2, 3]
Values: [10, 15, 40]
Capacity: 6
Maximum value: 65
Time taken: 0.000074 seconds

Weights: [2, 3, 4, 5]
Values: [3, 4, 5, 6]
Capacity: 5
Maximum value: 7
Time taken: 0.000065 seconds
```

📘 Fibonacci Sequence (DP Approach)

🔧 How It Works:

The Fibonacci Sequence is a series where each number is the **sum of the two preceding ones**. Using **dynamic programming**, we avoid repeated calculations by storing already computed values (memoization or tabulation), making the solution efficient even for large inputs.

⚙️ Performance:

- Best case: O(n)
- Average case: O(n)
- Worst case: O(n)
- Space: O(n) (O(1) with space optimization)

```
# ✅ Python Implementation

def fibonacci_dp(n):
    if n <= 1:
        return n

    fib = [0] * (n + 1)
    fib[1] = 1

    for i in range(2, n + 1):
        fib[i] = fib[i - 1] + fib[i - 2]

    return fib[n]


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1
n1 = 10
start = time.time()
result1 = fibonacci_dp(n1)
end = time.time()
print("Fibonacci number at position", n1, "is:", result1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: The 10th Fibonacci number is 55

# Test Case 2
n2 = 25
start = time.time()
result2 = fibonacci_dp(n2)
end = time.time()
print("\nFibonacci number at position", n2, "is:", result2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: The 25th Fibonacci number is 75025
```

```
Fibonacci number at position 10 is: 55
Time taken: 0.000113 seconds

Fibonacci number at position 25 is: 75025
Time taken: 0.000051 seconds
```

4. Machine Learning

Linear Regression – Predict continuous values.

Logistic Regression – Classification of binary outcomes.

K-Means Clustering – Partition data into clusters.

📘 Linear Regression (Closed-Form Solution)

🔧 **How It Works:**
Linear Regression predicts a **continuous value** by fitting a straight line to data points. Using the **Normal Equation**, we compute the best-fit line by directly solving the equation:
$\theta = (X^tX)^{-1}X^ty$, where X is the feature matrix and y is the target vector. This method avoids iteration and gives an exact solution for small datasets.

⚙️ **Performance:**

- Best case: $O(n^2)$
- Average case: $O(n^2 \times m)$ (n = samples, m = features)
- Worst case: $O(n^3)$ (due to matrix inversion)
- Space: $O(n \times m)$

```
# ✅ Python Implementation

import numpy as np

def linear_regression_normal_eq(X, y):
    # Add bias term (intercept)
    X_b = np.c_[np.ones((X.shape[0], 1)), X]
    # Closed-form solution: θ = (XᵗX)^(-1) Xᵗy
    theta = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
    return theta  # Returns [intercept, slope(s)]


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1: Simple 1D data
X1 = np.array([[1], [2], [4], [3]])
y1 = np.array([1, 3, 7, 5])

start = time.time()
theta1 = linear_regression_normal_eq(X1, y1)
end = time.time()

print("Test Case 1 Coefficients:", theta1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Should fit a line close to y ≈ 2x - 1

# Test Case 2: 2D feature data
X2 = np.array([[1, 2], [2, 3], [4, 6], [3, 5]])
y2 = np.array([2, 4, 8, 6])

start = time.time()
theta2 = linear_regression_normal_eq(X2, y2)
end = time.time()

print("\nTest Case 2 Coefficients:", theta2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Multivariate regression to estimate weights for two input features
```

```
Test Case 1 Coefficients: [-1.  2.]
Time taken: 0.011103 seconds

Test Case 2 Coefficients: [4.26325641e-14 2.00000000e+00 0.00000000e+00]
Time taken: 0.000365 seconds
```

📘 Logistic Regression

🔧 **How It Works:**

Logistic Regression is a **classification algorithm** used to predict binary outcomes (e.g., yes/no, 0/1). It models the **probability** that an input belongs to a certain class using the **sigmoid function**, which maps any real value to the range [0, 1]. Model parameters (weights) are learned using **gradient descent** to minimize the log loss (cross-entropy).

⚙️ **Performance:**

- Best case: O(n × m)
- Average case: O(n × m × k) (k = iterations)
- Worst case: O(n × m × k)
- Space: O(m) (m = number of features)

```
# ✅ Python Implementation

import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def logistic_regression(X, y, lr=0.01, epochs=1000):
    m, n = X.shape
    X_b = np.c_[np.ones((m, 1)), X]  # add bias term
    theta = np.zeros(n + 1)

    for _ in range(epochs):
        z = X_b.dot(theta)
        predictions = sigmoid(z)
        gradient = (1 / m) * X_b.T.dot(predictions - y)
        theta -= lr * gradient
    return theta


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1: Simple OR logic
X1 = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y1 = np.array([0, 1, 1, 1])

start = time.time()
theta1 = logistic_regression(X1, y1, lr=0.1, epochs=1000)
end = time.time()

print("Test Case 1 Coefficients:", theta1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Should learn to classify the OR function

# Test Case 2: Linear separation
X2 = np.array([[1], [2], [3], [4]])
y2 = np.array([0, 0, 1, 1])

start = time.time()
theta2 = logistic_regression(X2, y2, lr=0.1, epochs=1000)
end = time.time()

print("\nTest Case 2 Coefficients:", theta2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Should learn a sigmoid that separates around x=2.5
```

📘 K-Means Clustering

🔧 **How It Works:**
K-Means is an **unsupervised learning** algorithm used to group data into **k clusters**. It starts by initializing k random centroids, then repeatedly performs two steps:

1. **Assign** each data point to the nearest centroid.
2. **Update** centroids as the mean of points assigned to each cluster.
   The process continues until centroids stop changing or a maximum number of iterations is reached.

⚙️ **Performance:**

- Best case: O(n × k × i)
- Average case: O(n × k × i) (n = samples, k = clusters, i = iterations)
- Worst case: O(n × k × i)
- Space: O(n + k)

```
# ✅ Python Implementation

import numpy as np

def kmeans(X, k, max_iters=100):
    n_samples, _ = X.shape
    rng = np.random.default_rng(seed=42)
    centroids = X[rng.choice(n_samples, k, replace=False)]

    for _ in range(max_iters):
        # Assign each point to the nearest centroid
        distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)
        clusters = np.argmin(distances, axis=1)

        # Update centroids
        new_centroids = np.array([X[clusters == j].mean(axis=0) for j in range(k)])

        # Check for convergence
        if np.allclose(centroids, new_centroids):
            break
        centroids = new_centroids
    return centroids, clusters


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1: Simple 2D points
X1 = np.array([[1, 2], [1, 4], [1, 0],
               [10, 2], [10, 4], [10, 0]])
k1 = 2

start = time.time()
centroids1, labels1 = kmeans(X1, k1)
end = time.time()

print("Test Case 1 - Centroids:\n", centroids1)
print("Cluster Labels:", labels1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Should separate into two clusters around x ≈ 1 and x ≈ 10

# Test Case 2: 1D clustering
X2 = np.array([[2], [4], [10], [12], [3], [20]])
k2 = 3

start = time.time()
centroids2, labels2 = kmeans(X2, k2)
end = time.time()

print("\nTest Case 2 - Centroids:\n", centroids2)
print("Cluster Labels:", labels2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Clusters should form around [2–4], [10–12], and [20]
```

```
Test Case 1 - Centroids:
 [[ 1.  2.]
 [10.  2.]]
Cluster Labels: [0 0 0 1 1 1]
Time taken: 0.026725 seconds

Test Case 2 - Centroids:
 [[20.]
 [ 3.]
 [11.]]
Cluster Labels: [1 1 2 2 1 0]
Time taken: 0.000931 seconds
```

5. String Algorithms

Rabin-Karp Algorithm – Pattern searching with hashing.

KMP (Knuth-Morris-Pratt) – Efficient string matching.

Edit Distance (Levenshtein Distance) – Measure difference between strings.

📘 Rabin-Karp Algorithm

🔧 **How It Works:**
Rabin-Karp is a **string searching** algorithm that uses **hashing** to find a pattern in a text. It computes a hash for the pattern and for each substring of the same length in the text, comparing hashes first (fast) and only checking characters if the hashes match (to avoid collisions).

⚙️ **Performance:**

- Best case: O(n + m)
- Average case: O(n + m)
- Worst case: O(n × m) (many hash collisions)
- Space: O(1) (excluding input/output)

```
# ✅ Python Implementation
```

```python
def rabin_karp(text, pattern, base=256, prime=101):
    n, m = len(text), len(pattern)
    hpattern = 0
    htext = 0
    h = 1

    # Precompute h = (base^(m-1)) % prime
    for _ in range(m - 1):
        h = (h * base) % prime

    # Compute initial hash values
    for i in range(m):
        hpattern = (base * hpattern + ord(pattern[i])) % prime
        htext = (base * htext + ord(text[i])) % prime

    matches = []
    for i in range(n - m + 1):
        if hpattern == htext:
            if text[i:i + m] == pattern:
                matches.append(i)

        if i < n - m:
            htext = (base * (htext - ord(text[i]) * h) + ord(text[i + m])) % prime
            if htext < 0:
                htext += prime

    return matches


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1
text1 = "abracadabra"
pattern1 = "abra"

start = time.time()
result1 = rabin_karp(text1, pattern1)
end = time.time()

print("Text:", text1)
print("Pattern:", pattern1)
print("Match positions:", result1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: "abra" appears at positions 0 and 7

# Test Case 2
text2 = "aaaaaa"
pattern2 = "aaa"

start = time.time()
result2 = rabin_karp(text2, pattern2)
end = time.time()

print("\nText:", text2)
print("Pattern:", pattern2)
print("Match positions:", result2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: "aaa" appears at positions 0, 1, 2, and 3 (overlapping)
```

```
Text: abracadabra
Pattern: abra
Match positions: [0, 7]
Time taken: 0.000068 seconds

Text: aaaaaa
Pattern: aaa
Match positions: [0, 1, 2, 3]
Time taken: 0.000057 seconds
```

📘 Knuth-Morris-Pratt (KMP) Algorithm

🔧 **How It Works:**

KMP is a **pattern matching** algorithm that avoids redundant comparisons by preprocessing the pattern into a **partial match table** (also called the LPS array). This table tells the algorithm how far to shift the pattern upon a mismatch, allowing it to skip unnecessary checks.

⚙️ **Performance:**

- Best case: O(n)
- Average case: O(n)
- Worst case: O(n) (n = length of text)
- Space: O(m) (m = length of pattern for LPS array)

```
# ✅ Python Implementation

def compute_lps(pattern):
    lps = [0] * len(pattern)
    length = 0  # length of the previous longest prefix suffix

    i = 1
    while i < len(pattern):
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
    return lps

def kmp_search(text, pattern):
    n, m = len(text), len(pattern)
    lps = compute_lps(pattern)
    i = j = 0
    matches = []

    while i < n:
        if text[i] == pattern[j]:
            i += 1
            j += 1
        if j == m:
            matches.append(i - j)
            j = lps[j - 1]
        elif i < n and text[i] != pattern[j]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return matches


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1
text1 = "abxabcabcaby"
pattern1 = "abcaby"

start = time.time()
result1 = kmp_search(text1, pattern1)
end = time.time()

print("Text:", text1)
print("Pattern:", pattern1)
print("Match positions:", result1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Pattern "abcaby" starts at position 6

# Test Case 2
text2 = "aaaaabaaaaab"
pattern2 = "aaaaab"
```

```
start = time.time()
result2 = kmp_search(text2, pattern2)
end = time.time()

print("\nText:", text2)
print("Pattern:", pattern2)
print("Match positions:", result2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Pattern "aaaaab" matches at positions 0 and 6
```

📘 Edit Distance (Levenshtein Distance)

🔧 **How It Works:**

Edit Distance measures how many **insertions, deletions, or substitutions** are needed to convert one string into another. It uses **dynamic programming** to build a matrix where each cell represents the minimum number of edits required for substrings up to that point.

⚙️ **Performance:**

- Best case: O(m × n)
- Average case: O(m × n)
- Worst case: O(m × n) (m and n are string lengths)
- Space: O(m × n) (can be optimized to O(min(m, n)))

```
# ✅ Python Implementation

def edit_distance(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j  # insert all characters of s2
            elif j == 0:
                dp[i][j] = i  # remove all characters of s1
            elif s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]  # characters match
            else:
                dp[i][j] = 1 + min(
                    dp[i - 1][j],    # delete
                    dp[i][j - 1],    # insert
                    dp[i - 1][j - 1] # substitute
                )
    return dp[m][n]


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1
s1_1 = "kitten"
s2_1 = "sitting"

start = time.time()
dist1 = edit_distance(s1_1, s2_1)
end = time.time()

print("String 1:", s1_1)
print("String 2:", s2_1)
print("Edit distance:", dist1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Minimum operations = 3 (kitten → sitten → sittin → sitting)

# Test Case 2
s1_2 = "flaw"
s2_2 = "lawn"

start = time.time()
```

```
dist2 = edit_distance(s1_2, s2_2)
end = time.time()

print("\nString 1:", s1_2)
print("String 2:", s2_2)
print("Edit distance:", dist2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Minimum operations = 2 (flaw → law → lawn)
```

```
String 1: kitten
String 2: sitting
Edit distance: 3
Time taken: 0.000133 seconds

String 1: flaw
String 2: lawn
Edit distance: 2
Time taken: 0.000101 seconds
```

6. Tree Algorithms

Binary Tree Traversals (Inorder, Preorder, Postorder) – Navigate tree structures.

Lowest Common Ancestor (LCA) – Find shared parent node.

Trie Insertion and Search – Efficient prefix lookups.

📘 Binary Tree Traversals (Inorder, Preorder, Postorder)

🔧 **How It Works:**
Tree traversal algorithms visit all nodes of a **binary tree** in a specific order.

- **Inorder:** Left → Root → Right
- **Preorder:** Root → Left → Right
- **Postorder:** Left → Right → Root

  Each traversal can be done **recursively or iteratively** and is used in parsing, evaluation, and tree reconstruction tasks.

⚙️ **Performance:**

- Best case: O(n)
- Average case: O(n)
- Worst case: O(n) (n = number of nodes)
- Space: O(h) (h = height of the tree, for recursion stack)

```
# ✅ Python Implementation

class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def inorder(root):
    return inorder(root.left) + [root.val] + inorder(root.right) if root else []

def preorder(root):
    return [root.val] + preorder(root.left) + preorder(root.right) if root else []

def postorder(root):
    return postorder(root.left) + postorder(root.right) + [root.val] if root else []
```

```
# 🧪 Test Cases with Timer and Explanation

import time

# Construct the tree:
#       1
#      / \
#     2   3
#    / \
#   4   5

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

start = time.time()
print("Inorder:", inorder(root))      # Expected: [4, 2, 5, 1, 3]
print("Preorder:", preorder(root))    # Expected: [1, 2, 4, 5, 3]
print("Postorder:", postorder(root))  # Expected: [4, 5, 2, 3, 1]
end = time.time()

print(f"Time taken: {end - start:.6f} seconds")
# Explanation:
# Inorder visits Left → Root → Right → [4, 2, 5, 1, 3]
# Preorder visits Root → Left → Right → [1, 2, 4, 5, 3]
# Postorder visits Left → Right → Root → [4, 5, 2, 3, 1]
```

```
Inorder: [4, 2, 5, 1, 3]
Preorder: [1, 2, 4, 5, 3]
Postorder: [4, 5, 2, 3, 1]
Time taken: 0.001306 seconds
```

📘 Lowest Common Ancestor (LCA)

🔧 How It Works:

The LCA of two nodes in a tree is the **deepest node** that is an ancestor of both. In a **binary tree**, it can be found by recursively searching for the two nodes: if one is found in the left subtree and the other in the right, the current node is their LCA. Optimized versions use preprocessing for fast queries in rooted trees.

⚙️ Performance:

- Best case: O(log n) (in balanced trees)
- Average case: O(n)
- Worst case: O(n) (in skewed trees)
- Space: O(h) (h = height of the tree)

```
# ✅ Python Implementation

class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def find_lca(root, p, q):
    if root is None or root == p or root == q:
        return root

    left = find_lca(root.left, p, q)
    right = find_lca(root.right, p, q)

    if left and right:
        return root
    return left if left else right



# 🧪 Test Cases with Timer and Explanation

import time

# Build the tree:
#       3
#      / \
#     5   1
```

```python
#    / \ / \
#   6  2 0  8
#     / \
#    7   4

root = TreeNode(3)
root.left = TreeNode(5)
root.right = TreeNode(1)
root.left.left = TreeNode(6)
root.left.right = TreeNode(2)
root.right.left = TreeNode(0)
root.right.right = TreeNode(8)
root.left.right.left = TreeNode(7)
root.left.right.right = TreeNode(4)

p = root.left        # Node 5
q = root.left.right.right  # Node 4

start = time.time()
lca_node = find_lca(root, p, q)
end = time.time()

print("LCA of nodes", p.val, "and", q.val, "is:", lca_node.val)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Node 5 is an ancestor of node 4, so LCA is 5
```

```
LCA of nodes 5 and 4 is: 5
Time taken: 0.000105 seconds
```

📘 Trie Insertion and Search

🔧 **How It Works:**

A Trie (prefix tree) is a tree-based data structure used to **store strings efficiently**. Each node represents a character, and words are formed by paths from the root. Insertion and search follow the characters of a word one by one, creating nodes as needed. Tries are ideal for **prefix matching** and **autocomplete**.

⚙️ **Performance:**

- Best case: O(L)
- Average case: O(L) (L = length of word)
- Worst case: O(L)
- Space: O(AL × N) (A = alphabet size, N = number of words)

```python
# ✅ Python Implementation

class TrieNode:
    def __init__(self):
        self.children = {}
        self.end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
        node.end_of_word = True

    def search(self, word):
        node = self.root
        for ch in word:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return node.end_of_word
```

```
# 🧪 Test Cases with Timer and Explanation

import time

trie = Trie()

# Test Case 1: Insert and search for multiple words
start = time.time()
words = ["apple", "app", "ape"]
for word in words:
    trie.insert(word)

print("Search 'apple':", trie.search("apple"))  # True
print("Search 'app':", trie.search("app"))      # True
print("Search 'apex':", trie.search("apex"))    # False
end = time.time()

print(f"Time taken: {end - start:.6f} seconds")
# Explanation:
# - "apple" and "app" were inserted → should return True
# - "apex" was not inserted → should return False
```

```
Search 'apple': True
Search 'app': True
Search 'apex': False
Time taken: 0.000329 seconds
```

7. Computational Geometry

Convex Hull (Graham Scan / Andrew's Algorithm) – Smallest convex polygon enclosing points.

Line Segment Intersection – Detect overlaps in 2D space.

Sweep Line Algorithm – For processing geometric events in order.

📘 Convex Hull (Graham Scan)

🔧 **How It Works:**
The Convex Hull algorithm finds the **smallest convex polygon** that encloses a set of points. Graham Scan first sorts the points, then uses a **stack** to build the hull by checking the **turn direction** (left or right) between points. Points that make a right turn are removed to maintain convexity.

⚙️ **Performance:**

- Best case: O(n log n)
- Average case: O(n log n)
- Worst case: O(n log n)
- Space: O(n) (for sorting and stack)

```
# ✅ Python Implementation

def cross(o, a, b):
    # Cross product of vectors OA and OB (O = origin, A & B = points)
    return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

def convex_hull(points):
    points = sorted(set(points))  # Sort and remove duplicates
    if len(points) <= 1:
        return points

    lower = []
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)

    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)

    # Remove last point of each half (repeats start/end of other half)
    return lower[:-1] + upper[:-1]


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1
points1 = [(0, 0), (1, 1), (2, 2), (2, 0), (2, 4), (3, 3), (0, 3)]

start = time.time()
hull1 = convex_hull(points1)
end = time.time()

print("Input points:", points1)
print("Convex Hull:", hull1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Hull includes outermost points forming the convex boundary

# Test Case 2
points2 = [(1, 2), (2, 2), (2, 1), (1, 1), (1.5, 1.5)]

start = time.time()
hull2 = convex_hull(points2)
end = time.time()

print("\nInput points:", points2)
print("Convex Hull:", hull2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Convex hull excludes the center point (1.5, 1.5) and keeps corners
```

```
Input points: [(0, 0), (1, 1), (2, 2), (2, 0), (2, 4), (3, 3), (0, 3)]
Convex Hull: [(0, 0), (2, 0), (3, 3), (2, 4), (0, 3)]
Time taken: 0.000081 seconds

Input points: [(1, 2), (2, 2), (2, 1), (1, 1), (1.5, 1.5)]
Convex Hull: [(1, 1), (2, 1), (2, 2), (1, 2)]
Time taken: 0.000064 seconds
```

📘 Line Segment Intersection

🔧 How It Works:

This algorithm checks whether any two **line segments intersect** in a 2D plane. The common approach is to use the **orientation method** to determine the relative direction of points and apply the **general case and special case rules**. For many segments, a **sweep line** technique with an event queue is used for efficiency.

⚙️ Performance:

- Best case: O(n log n) (with sweep line algorithm)
- Average case: O(n log n + k) (k = number of intersections)
- Worst case: O(n²) (brute-force comparison)
- Space: O(n) (event queue and active set)

```
# ✅ Python Implementation

def orientation(p, q, r):
    val = (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] - q[1])
    if val == 0:
        return 0  # collinear
    return 1 if val > 0 else 2  # 1 = clockwise, 2 = counterclockwise

def on_segment(p, q, r):
    return min(p[0], r[0]) <= q[0] <= max(p[0], r[0]) and min(p[1], r[1]) <= q[1] <= max(p[1], r[1])

def segments_intersect(p1, q1, p2, q2):
    o1 = orientation(p1, q1, p2)
    o2 = orientation(p1, q1, q2)
    o3 = orientation(p2, q2, p1)
    o4 = orientation(p2, q2, q1)

    # General case
    if o1 != o2 and o3 != o4:
        return True

    # Special cases (collinear and overlapping)
    if o1 == 0 and on_segment(p1, p2, q1): return True
    if o2 == 0 and on_segment(p1, q2, q1): return True
    if o3 == 0 and on_segment(p2, p1, q2): return True
    if o4 == 0 and on_segment(p2, q1, q2): return True

    return False


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1: Intersecting segments
p1, q1 = (1, 1), (4, 4)
p2, q2 = (1, 4), (4, 1)

start = time.time()
result1 = segments_intersect(p1, q1, p2, q2)
end = time.time()

print("Segments 1 and 2 intersect:", result1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Segments cross each other forming an 'X'

# Test Case 2: Non-intersecting segments
p3, q3 = (0, 0), (1, 1)
p4, q4 = (2, 2), (3, 3)

start = time.time()
result2 = segments_intersect(p3, q3, p4, q4)
end = time.time()

print("\nSegments 3 and 4 intersect:", result2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Segments are collinear but non-overlapping
```

```
Segments 1 and 2 intersect: True
Time taken: 0.000091 seconds

Segments 3 and 4 intersect: False
Time taken: 0.000096 seconds
```

📘 Sweep Line Algorithm

🔧 How It Works:

The Sweep Line algorithm processes a set of geometric events (like points or segments) by moving a **virtual vertical line** from left to right. It maintains an **active set** of objects the line is currently intersecting. As events occur (like segment start/end), the structure updates to detect **intersections** or enforce **ordering** efficiently.

⚙️ Performance:

- Best case: O(n log n)
- Average case: O(n log n + k) (k = number of output events)
- Worst case: $O(n^2)$ (if all elements interact)
- Space: O(n) (event queue and active data structure)

```
# ✅ Python Implementation (Bentley-Ottmann simplified version for vertical/horizontal lines)

import heapq
```

```
def sweep_line_intersections(segments):
    events = []
    for i, ((x1, y1), (x2, y2)) in enumerate(segments):
        left = (x1, y1) if x1 < x2 or (x1 == x2 and y1 < y2) else (x2, y2)
        right = (x2, y2) if left == (x1, y1) else (x1, y1)
        events.append((left[0], 0, left[1], i))   # segment start
        events.append((right[0], 1, right[1], i)) # segment end

    events.sort()
    active = set()
    intersections = []

    for x, typ, y, idx in events:
        if typ == 0:
            for other in active:
                if segments_intersect(segments[idx][0], segments[idx][1],
                                      segments[other][0], segments[other][1]):
                    intersections.append((idx, other))
            active.add(idx)
        else:
            active.discard(idx)

    return intersections



# 🧪 Test Cases with Timer and Explanation

import time

# Required helper from previous algorithm
def orientation(p, q, r):
    val = (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] - q[1])
    if val == 0:
        return 0
    return 1 if val > 0 else 2

def on_segment(p, q, r):
    return min(p[0], r[0]) <= q[0] <= max(p[0], r[0]) and min(p[1], r[1]) <= q[1] <= max(p[1], r[1])

def segments_intersect(p1, q1, p2, q2):
    o1 = orientation(p1, q1, p2)
    o2 = orientation(p1, q1, q2)
    o3 = orientation(p2, q2, p1)
    o4 = orientation(p2, q2, q1)

    if o1 != o2 and o3 != o4:
        return True
    if o1 == 0 and on_segment(p1, p2, q1): return True
    if o2 == 0 and on_segment(p1, q2, q1): return True
    if o3 == 0 and on_segment(p2, p1, q2): return True
    if o4 == 0 and on_segment(p2, q1, q2): return True
    return False

# Test Case
segments = [
    ((1, 1), (5, 5)),
    ((1, 5), (5, 1)),
    ((2, 2), (6, 2)),
    ((3, 0), (3, 6)),
]

start = time.time()
intersections = sweep_line_intersections(segments)
end = time.time()

print("Segments:", segments)
print("Intersecting pairs (by indices):", intersections)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Reports index pairs of segments that intersect (e.g., (0, 1), (2, 3))
```

```
Segments: [((1, 1), (5, 5)), ((1, 5), (5, 1)), ((2, 2), (6, 2)), ((3, 0), (3, 6))]
Intersecting pairs (by indices): [(1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2)]
Time taken: 0.000090 seconds
```

8. Greedy Algorithms

Activity Selection Problem – Choose max compatible activities.

Huffman Coding – Optimal prefix coding for data compression.

Prim’s Algorithm – Minimum Spanning Tree construction.

📘 Activity Selection Problem

🔧 **How It Works:**
The Activity Selection Problem aims to **select the maximum number of non-overlapping activities**, each with a start and end time. The greedy strategy is to **sort activities by end time** and always pick the next one that finishes earliest and doesn't overlap with the previous selection.

⚙️ **Performance:**

- Best case: O(n log n) (for sorting)
- Average case: O(n log n)
- Worst case: O(n log n)
- Space: O(1) (in-place selection)

```
# ✅ Python Implementation

def activity_selection(activities):
    # Sort activities based on end time
    sorted_acts = sorted(activities, key=lambda x: x[1])
    selected = [sorted_acts[0]]

    for i in range(1, len(sorted_acts)):
        if sorted_acts[i][0] >= selected[-1][1]:
            selected.append(sorted_acts[i])

    return selected


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1
activities1 = [(1, 4), (3, 5), (0, 6), (5, 7), (8, 9), (5, 9)]

start = time.time()
selected1 = activity_selection(activities1)
end = time.time()

print("Input activities:", activities1)
print("Selected activities:", selected1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Optimal selection is [(1, 4), (5, 7), (8, 9)]

# Test Case 2
activities2 = [(2, 3), (3, 4), (0, 1), (5, 9), (6, 10)]

start = time.time()
selected2 = activity_selection(activities2)
end = time.time()

print("\nInput activities:", activities2)
print("Selected activities:", selected2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Optimal selection is [(0, 1), (2, 3), (3, 4), (5, 9)]
```

```
Input activities: [(1, 4), (3, 5), (0, 6), (5, 7), (8, 9), (5, 9)]
Selected activities: [(1, 4), (5, 7), (8, 9)]
Time taken: 0.000063 seconds

Input activities: [(2, 3), (3, 4), (0, 1), (5, 9), (6, 10)]
```

```
Selected activities: [(0, 1), (2, 3), (3, 4), (5, 9)]
Time taken: 0.000056 seconds
```

📘 Huffman Coding

🔧 **How It Works:**

Huffman Coding is a **greedy algorithm** used for **lossless data compression**. It builds a binary tree where **frequent characters have shorter codes**. The process starts by placing all characters in a priority queue based on frequency and then repeatedly merges the two least frequent nodes until one tree remains.

⚙️ **Performance:**

- Best case: O(n log n) (n = number of characters)
- Average case: O(n log n)
- Worst case: O(n log n)
- Space: O(n) (for tree and code table)

```python
# ✅ Python Implementation

import heapq
from collections import defaultdict, Counter

class HuffmanNode:
    def __init__(self, char=None, freq=0):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):  # For priority queue
        return self.freq < other.freq

def build_huffman_tree(text):
    freq = Counter(text)
    heap = [HuffmanNode(ch, fr) for ch, fr in freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        n1 = heapq.heappop(heap)
        n2 = heapq.heappop(heap)
        merged = HuffmanNode(freq=n1.freq + n2.freq)
        merged.left = n1
        merged.right = n2
        heapq.heappush(heap, merged)

    return heap[0]

def generate_codes(node, current_code="", codes=None):
    if codes is None:
        codes = {}
    if node:
        if node.char is not None:
            codes[node.char] = current_code
        generate_codes(node.left, current_code + "0", codes)
        generate_codes(node.right, current_code + "1", codes)
    return codes

def huffman_encode(text):
    root = build_huffman_tree(text)
    code_map = generate_codes(root)
    encoded = ''.join(code_map[ch] for ch in text)
    return encoded, code_map


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1
text1 = "aabacabad"
```

```
start = time.time()
encoded1, codes1 = huffman_encode(text1)
end = time.time()

print("Original text:", text1)
print("Huffman Codes:", codes1)
print("Encoded binary string:", encoded1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Characters with higher frequency get shorter codes (e.g., 'a')

# Test Case 2
text2 = "this is an example for huffman encoding"

start = time.time()
encoded2, codes2 = huffman_encode(text2)
end = time.time()

print("\nOriginal text:", text2)
print("Huffman Codes:", codes2)
print("Encoded binary string:", encoded2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Common letters like ' ' (space) and 'e' will have shorter codes
```

```
Original text: aabacabad
Huffman Codes: {'b': '00', 'c': '010', 'd': '011', 'a': '1'}
Encoded binary string: 110010101001011
Time taken: 0.000232 seconds

Original text: this is an example for huffman encoding
Huffman Codes: {'n': '000', 's': '0010', 'm': '0011', 'h': '0100', 't': '01010', 'd': '01011', 'r': '01100', 'l': '01101', 'x': '01110',
Encoded binary string: 010100100100100101011001001010111110001011110011101111001110000011011110101110111001011001010100110001101110100 11
Time taken: 0.000160 seconds
```

📘 Prim's Algorithm

🔧 **How It Works:**

Prim's Algorithm finds a **minimum spanning tree (MST)** for a weighted, connected graph. It starts from any node and grows the MST by always choosing the **smallest weight edge** that connects a new node to the tree. A **priority queue (min-heap)** helps efficiently select the next edge.

⚙️ **Performance:**

- Best case: O(E + log V) (with Fibonacci heap)
- Average case: O(E log V) (with binary heap and adjacency list)
- Worst case: O(V²) (with adjacency matrix)
- Space: O(V + E)

```
# ✅ Python Implementation

import heapq
from collections import defaultdict

def prim_mst(graph, start):
    mst = []
    visited = set([start])
    edges = [(weight, start, to) for to, weight in graph[start]]
    heapq.heapify(edges)

    while edges:
        weight, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst.append((frm, to, weight))
            for neighbor, w in graph[to]:
                if neighbor not in visited:
                    heapq.heappush(edges, (w, to, neighbor))
    return mst


# 🧪 Test Cases with Timer and Explanation
```

```
import time

# Test Case 1
graph1 = {
    'A': [('B', 2), ('D', 6)],
    'B': [('A', 2), ('C', 3), ('D', 8)],
    'C': [('B', 3), ('D', 5)],
    'D': [('A', 6), ('B', 8), ('C', 5)]
}

start = time.time()
mst1 = prim_mst(graph1, 'A')
end = time.time()

print("Graph:", graph1)
print("MST edges:", mst1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: The MST includes the smallest edges connecting all nodes without cycles

# Test Case 2
graph2 = {
    0: [(1, 4), (2, 3)],
    1: [(0, 4), (2, 1), (3, 2)],
    2: [(0, 3), (1, 1), (3, 4)],
    3: [(1, 2), (2, 4)]
}

start = time.time()
mst2 = prim_mst(graph2, 0)
end = time.time()

print("\nGraph:", graph2)
print("MST edges:", mst2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Outputs the set of edges that form the minimum spanning tree
```

```
Graph: {'A': [('B', 2), ('D', 6)], 'B': [('A', 2), ('C', 3), ('D', 8)], 'C': [('B', 3), ('D', 5)], 'D': [('A', 6), ('B', 8), ('C', 5)]}
MST edges: [('A', 'B', 2), ('B', 'C', 3), ('C', 'D', 5)]
Time taken: 0.000075 seconds

Graph: {0: [(1, 4), (2, 3)], 1: [(0, 4), (2, 1), (3, 2)], 2: [(0, 3), (1, 1), (3, 4)], 3: [(1, 2), (2, 4)]}
MST edges: [(0, 2, 3), (2, 1, 1), (1, 3, 2)]
Time taken: 0.000076 seconds
```

9. Backtracking

N-Queens Problem – Place queens without attacks.

Sudoku Solver – Fill the board using constraint satisfaction.

Subset Sum Problem – Decide if subset sums to target.

📘 N-Queens Problem

🔧 **How It Works:**
The N-Queens Problem places **N queens on an N×N chessboard** so that no two queens threaten each other. It uses **backtracking** to try placing queens row by row, and if a conflict occurs, it backtracks to try a different position. It ensures no two queens share the same row, column, or diagonal.

⚙️ **Performance:**

- Best case: O(N!)
- Average case: O(N!)
- Worst case: O(N!)
- Space: O(N) (for storing queen positions)

```
# ✅ Python Implementation
```

```python
def solve_n_queens(n):
    solutions = []
    board = [-1] * n  # board[i] = column of queen in row i

    def is_safe(row, col):
        for i in range(row):
            if board[i] == col or \
               abs(board[i] - col) == abs(i - row):
                return False
        return True

    def backtrack(row):
        if row == n:
            solutions.append(board[:])
            return
        for col in range(n):
            if is_safe(row, col):
                board[row] = col
                backtrack(row + 1)
                board[row] = -1

    backtrack(0)
    return solutions


# 🧪 Test Cases with Timer and Explanation

import time

def print_board(solution):
    n = len(solution)
    for row in solution:
        line = ['.'] * n
        line[row] = 'Q'
        print("".join(line))
    print()

# Test Case 1
n1 = 4
start = time.time()
sol1 = solve_n_queens(n1)
end = time.time()

print(f"Total solutions for N = {n1}:", len(sol1))
print("One solution:")
print_board(sol1[0])
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: 4-Queens has 2 solutions; shows one with safe placement

# Test Case 2
n2 = 5
start = time.time()
sol2 = solve_n_queens(n2)
end = time.time()

print(f"\nTotal solutions for N = {n2}:", len(sol2))
print("One solution:")
print_board(sol2[0])
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: 5-Queens has 10 solutions; shows one valid configuration
```

```
Total solutions for N = 4: 2
One solution:
.Q..
...Q
Q...
..Q.

Time taken: 0.000145 seconds

Total solutions for N = 5: 10
One solution:
Q....
..Q..
....Q
.Q...
...Q.
```

```
Time taken: 0.000269 seconds
```

📘 Sudoku Solver

🔧 **How It Works:**
The Sudoku Solver fills a 9×9 grid so that every row, column, and 3×3 box contains all digits from 1 to 9. It uses **backtracking** by trying digits in empty cells one by one. If a digit leads to a valid state, it proceeds; otherwise, it backtracks and tries the next digit.

⚙️ **Performance:**

- Best case: O(1) (if almost solved)
- Average case: $O(9^k)$ (k = number of empty cells)
- Worst case: $O(9^{81})$ (completely empty board)
- Space: O(k) (depth of recursion stack)

```python
# ✅ Python Implementation

def is_valid(board, row, col, num):
    for i in range(9):
        if board[row][i] == num or board[i][col] == num:
            return False

    box_start_row = row - row % 3
    box_start_col = col - col % 3

    for i in range(3):
        for j in range(3):
            if board[box_start_row + i][box_start_col + j] == num:
                return False

    return True

def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                for num in range(1, 10):
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = 0
                return False
    return True


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1: Easy Sudoku with a unique solution
board1 = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]

start = time.time()
solved = solve_sudoku(board1)
end = time.time()

print("Solved Sudoku Board:")
for row in board1:
    print(row)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Uses backtracking to fill in all empty cells with valid digits
```

```
Solved Sudoku Board:
[5, 3, 4, 6, 7, 8, 9, 1, 2]
[6, 7, 2, 1, 9, 5, 3, 4, 8]
[1, 9, 8, 3, 4, 2, 5, 6, 7]
[8, 5, 9, 7, 6, 1, 4, 2, 3]
[4, 2, 6, 8, 5, 3, 7, 9, 1]
[7, 1, 3, 9, 2, 4, 8, 5, 6]
[9, 6, 1, 5, 3, 7, 2, 8, 4]
[2, 8, 7, 4, 1, 9, 6, 3, 5]
[3, 4, 5, 2, 8, 6, 1, 7, 9]
Time taken: 0.075706 seconds
```

📘 Subset Sum Problem

🔧 How It Works:

The Subset Sum Problem checks if there exists a **subset of numbers** in a given list that **adds up to a target sum**. It uses **backtracking** to explore possible combinations, and can also be solved using **dynamic programming** for improved efficiency in some cases.

⚙️ Performance:

- Best case: O(1) (early solution found)
- Average case: O(2^n)
- Worst case: O(2^n) (n = number of elements)
- Space: O(n) (recursion stack)

```
# ✅ Python Implementation (Backtracking Approach)

def subset_sum(nums, target):
    def backtrack(index, current_sum):
        if current_sum == target:
            return True
        if index == len(nums) or current_sum > target:
            return False
        # Include current element
        if backtrack(index + 1, current_sum + nums[index]):
            return True
        # Exclude current element
        return backtrack(index + 1, current_sum)

    return backtrack(0, 0)


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1
nums1 = [3, 34, 4, 12, 5, 2]
target1 = 9

start = time.time()
result1 = subset_sum(nums1, target1)
end = time.time()

print("Input:", nums1)
print("Target sum:", target1)
print("Subset exists:", result1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: A subset [4, 5] or [3, 4, 2] sums to 9

# Test Case 2
nums2 = [1, 2, 5]
target2 = 4

start = time.time()
result2 = subset_sum(nums2, target2)
end = time.time()
```

```
print("\nInput:", nums2)
print("Target sum:", target2)
print("Subset exists:", result2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: No subset adds up to 4
```

```
Input: [3, 34, 4, 12, 5, 2]
Target sum: 9
Subset exists: True
Time taken: 0.000093 seconds

Input: [1, 2, 5]
Target sum: 4
Subset exists: False
Time taken: 0.000052 seconds
```

10. Cryptographic Algorithms

RSA Encryption/Decryption – Public key cryptography.

Caesar Cipher – Basic substitution cipher.

SHA-256 Hashing – Secure hashing for integrity checking.

📘 RSA Encryption/Decryption

🔧 **How It Works:**
RSA is a **public-key cryptographic algorithm** that secures data by using a pair of keys: a **public key** for encryption and a **private key** for decryption. It relies on the **mathematical difficulty of factoring large prime numbers**. Messages are encrypted as modular exponentiations and can only be decrypted using the private key.

⚙️ **Performance:**

- Best case: O(log n) (modular exponentiation)
- Average case: $O(n^3)$ (key generation and encryption/decryption)
- Worst case: $O(n^3)$
- Space: O(n) (size of keys and intermediate values)

```
# ✅ Python Implementation (Simplified for demonstration)

def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

def mod_inverse(e, phi):
    def extended_gcd(a, b):
        if a == 0:
            return (b, 0, 1)
        g, x1, y1 = extended_gcd(b % a, a)
        return (g, y1 - (b // a) * x1, x1)
    g, x, _ = extended_gcd(e, phi)
    if g != 1:
        raise Exception("Modular inverse doesn't exist.")
    return x % phi

def rsa_keygen(p, q):
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 3
    while gcd(e, phi) != 1:
        e += 2
    d = mod_inverse(e, phi)
    return (e, n), (d, n)  # public, private

def encrypt_rsa(message, pub_key):
    e, n = pub_key
    return pow(message, e, n)
```

```
def decrypt_rsa(cipher, priv_key):
    d, n = priv_key
    return pow(cipher, d, n)


# 🧪 Test Cases with Timer and Explanation

import time

# Using small primes for demonstration
p, q = 61, 53

start = time.time()
public_key, private_key = rsa_keygen(p, q)
message = 42

cipher = encrypt_rsa(message, public_key)
decrypted = decrypt_rsa(cipher, private_key)
end = time.time()

print("Original message:", message)
print("Encrypted message:", cipher)
print("Decrypted message:", decrypted)
print("Public key:", public_key)
print("Private key:", private_key)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Message 42 is encrypted and decrypted successfully using RSA
```

```
Original message: 42
Encrypted message: 240
Decrypted message: 42
Public key: (7, 3233)
Private key: (1783, 3233)
Time taken: 0.000236 seconds
```

📘 Caesar Cipher

🔧 How It Works:

Caesar Cipher is a **substitution cipher** where each letter in the plaintext is shifted by a fixed number of positions in the alphabet. For example, with a shift of 3, A becomes D, B becomes E, and so on. It is simple but easily breakable due to its limited number of possible shifts.

⚙️ Performance:

- Best case: O(n)
- Average case: O(n)
- Worst case: O(n) (n = length of message)
- Space: O(n) (to store output)

```
# ✅ Python Implementation

def caesar_encrypt(text, shift):
    result = ""
    for char in text:
        if char.isalpha():
            offset = ord('A') if char.isupper() else ord('a')
            result += chr((ord(char) - offset + shift) % 26 + offset)
        else:
            result += char
    return result

def caesar_decrypt(ciphertext, shift):
    return caesar_encrypt(ciphertext, -shift)


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1
message1 = "Hello World!"
```

```
shift1 = 3

start = time.time()
encrypted1 = caesar_encrypt(message1, shift1)
decrypted1 = caesar_decrypt(encrypted1, shift1)
end = time.time()

print("Original message:", message1)
print("Encrypted message:", encrypted1)
print("Decrypted message:", decrypted1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Each letter is shifted by 3 → 'H' → 'K', 'E' → 'H', etc.

# Test Case 2
message2 = "Python 3.8"
shift2 = 5

start = time.time()
encrypted2 = caesar_encrypt(message2, shift2)
decrypted2 = caesar_decrypt(encrypted2, shift2)
end = time.time()

print("\nOriginal message:", message2)
print("Encrypted message:", encrypted2)
print("Decrypted message:", decrypted2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Only alphabetic characters are shifted; digits and symbols stay the same
```

```
Original message: Hello World!
Encrypted message: Khoor Zruog!
Decrypted message: Hello World!
Time taken: 0.000152 seconds

Original message: Python 3.8
Encrypted message: Udymts 3.8
Decrypted message: Python 3.8
Time taken: 0.000161 seconds
```

📘 SHA-256 Hashing

🔧 **How It Works:**

SHA-256 is a **cryptographic hash function** that takes any input and produces a **fixed 256-bit (32-byte)** output. It works by applying a series of **bitwise operations, logical functions, and modular additions** over multiple rounds. It's used to ensure data integrity and is designed to be **one-way and collision-resistant**.

⚙️ **Performance:**

- Best case: O(n)
- Average case: O(n)
- Worst case: O(n) (n = length of input in bits)
- Space: O(1) (fixed-size output, no additional memory required)

```
# ✅ Python Implementation (using hashlib)

import hashlib

def sha256_hash(text):
    return hashlib.sha256(text.encode()).hexdigest()


# 🧪 Test Cases with Timer and Explanation

import time

# Test Case 1
text1 = "hello world"

start = time.time()
hash1 = sha256_hash(text1)
end = time.time()
```

```
print("Input text:", text1)
print("SHA-256 hash:", hash1)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Hash digest is fixed-length and changes completely with minor text changes

# Test Case 2
text2 = "hello world!"

start = time.time()
hash2 = sha256_hash(text2)
end = time.time()

print("\nInput text:", text2)
print("SHA-256 hash:", hash2)
print(f"Time taken: {end - start:.6f} seconds")
# Explanation: Adding just one character completely changes the output hash
```

```
Input text: hello world
SHA-256 hash: b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
Time taken: 0.000145 seconds

Input text: hello world!
SHA-256 hash: 7509e5bda0c762d2bac7f90d758b5b2263fa01ccbc542ab5e3df163be08e6ca9
Time taken: 0.000103 seconds
```

MCQ Type Questions

Question 1

```
# Q1. Quick Sort
# What will be the output of the quick_sort function on the input list below?

arr = [4, 1, 3]

# Options:
# A. [1, 3, 4]
# B. [4, 3, 1]
# C. [3, 1, 4]
# D. [1, 4, 3]


# ✅ Correct Answer: A
# Explanation: Pivot = 4, partitioned into [1, 3] and [], recursively sorted: [1, 3, 4]
```

Question 2

```
# Q2. Merge Sort
# What will be the output of the merge_sort function on the input list below?

arr = [5, 2, 1]

# Options:
# A. [1, 2, 5]
# B. [2, 1, 5]
# C. [5, 2, 1]
# D. [1, 5, 2]


# ✅ Correct Answer: A
# Explanation: Splits into [5] and [2, 1], sorts to [1, 2], merges with [5] → [1, 2, 5]
```

Question 3

```
# Q3. Binary Search
# What index will binary_search return for the value 4 in the sorted list below?
```

```
arr = [1, 2, 4, 6, 7]
target = 4

# Options:
# A. 1
# B. 2
# C. 3
# D. 4


# ✅ Correct Answer: B
# Explanation: 4 is found at index 2 using binary search
```

Question 4

```
# Q4. Dijkstra’s Algorithm
# What is the shortest distance from node 'A' to node 'D' in the graph below?

graph = {
    'A': [('B', 1)],
    'B': [('C', 2)],
    'C': [('D', 1)],
    'D': []
}
start = 'A'

# Options:
# A. 1
# B. 2
# C. 3
# D. 4


# ✅ Correct Answer: D
# Explanation: A → B (1) → C (2) → D (1) = 4
```

Question 5

```
# Q5. Depth-First Search (DFS)
# What is the order of visited nodes starting from node 1 in the graph below?

graph = {
    1: [2, 3],
    2: [4],
    3: [],
    4: []
}

# Options:
# A. [1, 2, 4, 3]
# B. [1, 3, 2, 4]
# C. [1, 2, 3, 4]
# D. [1, 4, 2, 3]


# ✅ Correct Answer: A
# Explanation: DFS visits 1 → 2 → 4 → then backtracks and visits 3
```

Question 6

```
# Q6. Breadth-First Search (BFS)
# What is the order of visited nodes starting from node 'A'?

graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
}

# Options:
# A. ['A', 'C', 'B', 'D', 'E']
# B. ['A', 'B', 'C', 'D', 'E']
# C. ['A', 'B', 'D', 'C', 'E']
# D. ['A', 'D', 'E', 'B', 'C']


# ✅ Correct Answer: B
# Explanation: BFS visits level-by-level: A → B, C → D, E
```

Question 7

```
# Q7. Longest Common Subsequence (LCS)
# What is the LCS length between s1 and s2?

s1 = "ABC"
s2 = "AC"

# Options:
# A. 3
# B. 1
# C. 2
# D. 0


# ✅ Correct Answer: C
# Explanation: LCS is "AC", length 2
```

Question 8

```
# Q8. 0/1 Knapsack
# What is the maximum value that can be obtained with capacity = 5?

weights = [2, 3]
values = [20, 30]
capacity = 5

# Options:
# A. 20
# B. 30
# C. 50
# D. 0


# ✅ Correct Answer: C
# Explanation: Include both items (2 + 3 ≤ 5) → total value = 20 + 30 = 50
```

Question 9

```
# Q9. Fibonacci (DP)
# What is the 6th Fibonacci number using bottom-up DP?

n = 6

# Options:
# A. 8
# B. 5
# C. 13
# D. 21


# ✅ Correct Answer: A
# Explanation: Fibonacci(6) = 8 (0-based: [0, 1, 1, 2, 3, 5, 8])
```

Question 10

```
# Q10. Linear Regression (Closed Form)
# Given X = [[1], [2]] and y = [3, 5], what is the predicted slope (ignoring intercept)?

# Options:
# A. 1
# B. 2
# C. 3
# D. 4


# ✅ Correct Answer: B
# Explanation: Two points: (1,3) and (2,5) → slope = (5-3)/(2-1) = 2
```

Word Problems

Problem 1: Route Planning with Traffic and Budget Constraints

Context:

You are designing a navigation system that helps users travel from one city to another. Each road segment has a travel time and a toll fee. A user wants to minimize travel time but also needs to stay within a given toll budget.

Algorithms to Use:

Dijkstra's Algorithm – to find the shortest path (based on time)

0/1 Knapsack – to choose a subset of toll roads within budget constraints

```
Start coding or generate with AI.
```

Problem 2: Secure File Compression for Cloud Backup

Context:

A software company wants to compress files before uploading them to a cloud server. To ensure data integrity, they also hash the compressed files before upload. During retrieval, the files are re-hashed and verified.

Algorithms to Use:

Huffman Coding – for lossless compression of file data

SHA-256 Hashing – for verifying integrity using cryptographic hash values

```
Start coding or generate with AI.
```

Problem 3: Smart Exam Scheduling System

Context:

A university needs to generate an exam schedule where no student has overlapping exams. The system must also find groups of students with overlapping course enrollments to ensure spacing in their exam schedule.

Algorithms to Use:

Activity Selection Problem – to schedule exams in non-overlapping slots

Depth-First Search (DFS) – to traverse student-course graphs to detect scheduling conflicts

```
Start coding or generate with AI.
```

Problem 4: Online Game – Safe Zone Placement

Context:

In a strategy-based online game, players must place safe zones on a battlefield where they do not threaten each other. Once placed, the system must determine whether a safe route exists between two zones without crossing hostile regions.

Algorithms to Use:

N-Queens Problem – to ensure zones are placed without conflict

Breadth-First Search (BFS) – to verify if a path exists between zones avoiding restricted tiles

```
Start coding or generate with AI.
```

Problem 5: Digital Signature Verification System

Context:

A secure messaging platform wants to allow senders to sign messages and receivers to verify that the messages haven't been tampered with. The signature is created using the sender's private key, and the content is hashed before and after transmission.

Algorithms to Use:

RSA Encryption/Decryption – for digital signing and verifying messages

SHA-256 Hashing – to ensure message content hasn’t changed during transmission

```
Start coding or generate with AI.
```

Problem 6: Investor Matching in a Business Directory App

Context:

You're building a social networking-style business directory. Users submit profiles of investors and startups, and the system needs to:

1. Search for relevant investor profiles by keyword.
2. Rank them based on the frequency of matched terms.

Algorithms to Use:

Trie – for fast prefix-based keyword lookup

KMP Algorithm – for substring pattern matching within descriptions

```
Start coding or generate with AI.
```

Problem 7: Emergency Response Drone Routing

Context:

A city dispatch system controls drones that must reach emergency sites as fast as possible, but with limited battery life. The system must:

1. Choose the fastest route to the emergency site.

2. Ensure the drone has enough energy for the round trip.

Algorithms to Use:

Dijkstra's Algorithm – to find the shortest route to the site

Subset Sum – to verify if battery can support a specific distance

```
Start coding or generate with AI.
```

Problem 8: AI Interview Bot – Answer Ranking

Context:

An AI bot is designed to conduct preliminary interviews. It compares a candidate's answers with a model answer and assigns a match score. It also checks for key phrases to confirm understanding.

Algorithms to Use:

Edit Distance (Levenshtein Distance) – to compute similarity between candidate and model answers

Rabin-Karp Algorithm – to efficiently detect key phrases in long text

```
Start coding or generate with AI.
```

Problem 9: Compiler – Parenthesis Checker and Expression Optimizer

Context:

You are building a compiler module that parses mathematical expressions. It needs to:

1. Check if brackets and structure are valid.
2. Reorder expression using optimal evaluation sequence.

Algorithms to Use:

DFS (on expression tree) – to validate syntax and nesting

Knapsack / LCS – to compute optimal subexpression reordering or alignment

```
Start coding or generate with AI.
```

Problem 10: Visual Map Boundary Detection

Context:

A geospatial analytics platform processes map tiles and needs to:

1. Identify the **outer boundary** of a cluster of points.
2. Detect and report **intersections** between different map layers.

Algorithms to Use:

Convex Hull (Graham Scan) – to extract the outer boundary of a point cluster

Line Segment Intersection – to find overlaps between map boundaries

```
Start coding or generate with AI.
```