K.R. MANGALAM UNIVERSITY
THE COMPLETE WORLD OF EDUCATION

School of Engineering & Technology

Design and Analysis of Algorithms Lab

ENCA351

BCA (AI & DS)

(2025-26)

Submitted By:
Student Name: Sourav Kumar.
Roll No.: 23012011871
Programme Name: BCA(AI&DS)
Semester: Vth

Submitted to:
Dr. Aarti
Assistant Professor
SoET

Table of Contents

LAB ASSIGNMENT- 1

Assignment Title: Analyzing and Visualizing Recursive Algorithm Efficiency

```
[1]
✓ 9s
```

```python
import time
import matplotlib.pyplot as plt
import sys
```

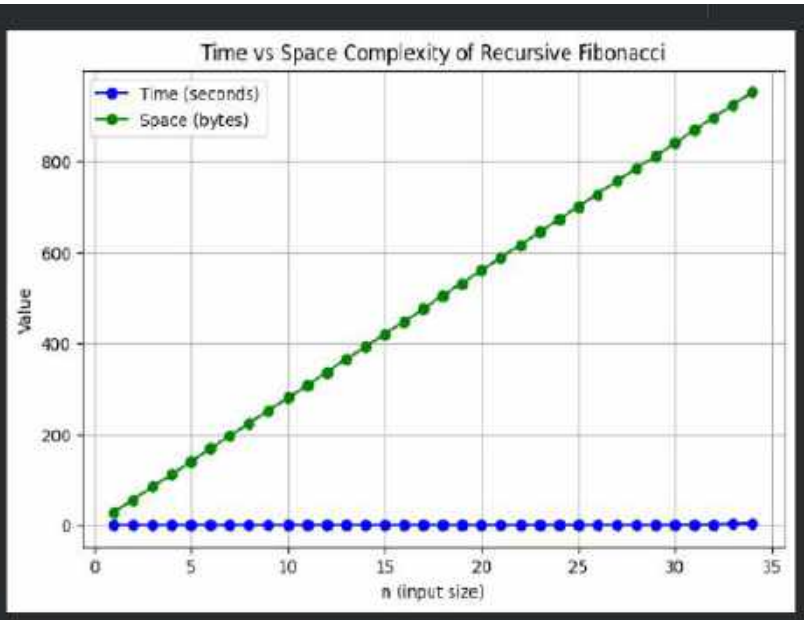
LAB DAA 1

File Edit View Insert Runtime Tools Help

mmands + Code + Text ▷ Run all

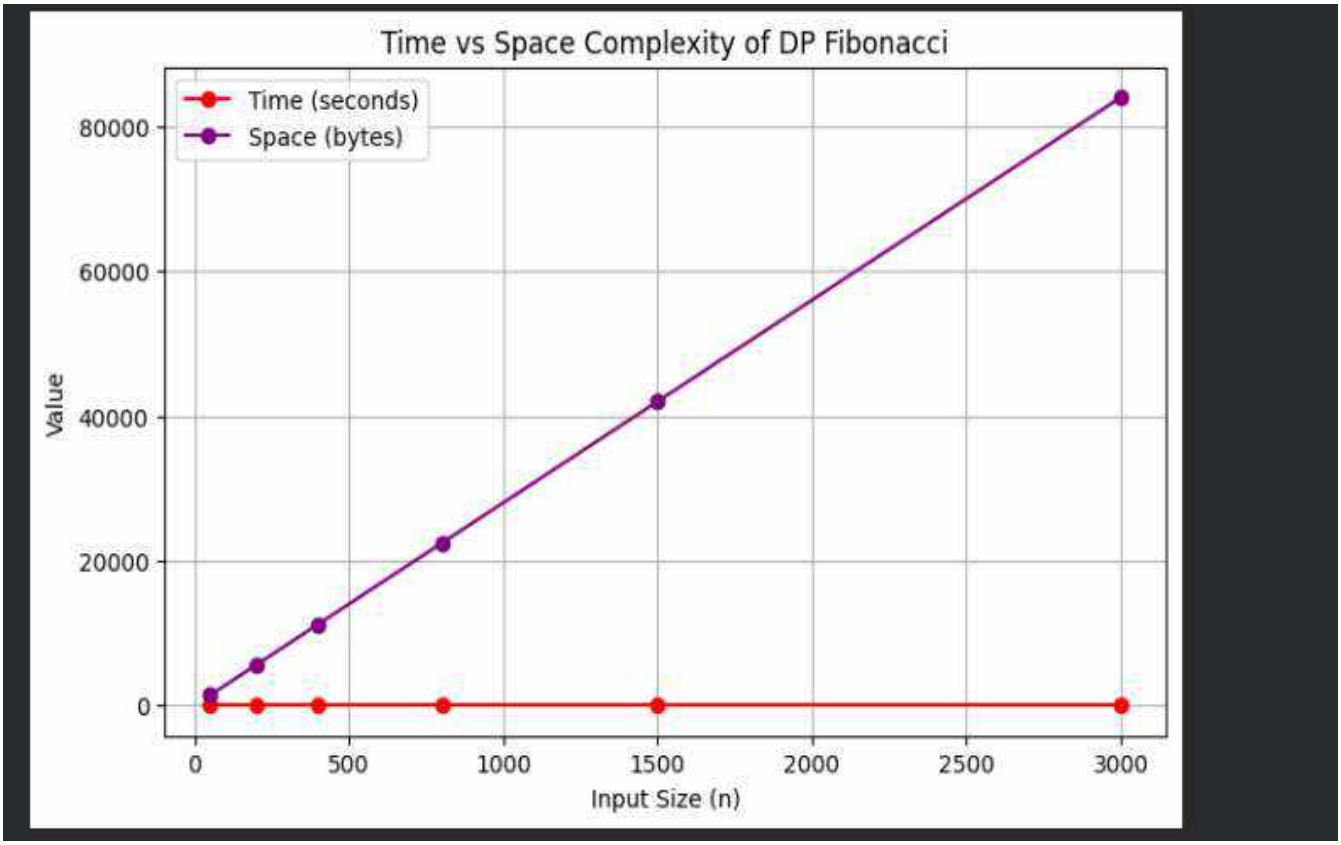
```
[1]
✓ 9s
```

```python
# Recursive Fibonacci Function

def fibonacci_recursive(n):
    if n <= 1:
        return n
    return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)

# Changed input values (different input range)
n_values = list(range(1, 35))  # ✦ Different input range
execution_times = []
space_usage = []

for n in n_values:

    start = time.time()
    fibonacci_recursive(n)
    end = time.time()
    execution_times.append(end - start)

    frame_size = sys.getsizeof(n)
    space_usage.append(n * frame_size)

plt.figure(figsize=(8, 5))
plt.plot(n_values, execution_times, marker='o', color='blue', label='Time (seconds)')
plt.plot(n_values, space_usage, marker='s', color='green', label='Space (bytes)')
plt.title('Time vs Space Complexity of Recursive Fibonacci')
plt.xlabel('n (input size)')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()

print("\n--- Recursive Fibonacci Analysis ---")
print("Input/Output:")
print("  Input: n (integer, position in Fibonacci sequence)")
print("  Output: nth Fibonacci number")
print("\nTime Complexity:")
print("  Best Case: O(1) when n = 0 or 1")
print("  Worst Case: O(2^n) due to repeated recursive calls")
print("  Average Case: O(2^n) (same as worst case)")
print("\nSpace Complexity:")
print("  O(n) because of recursion stack depth")
```



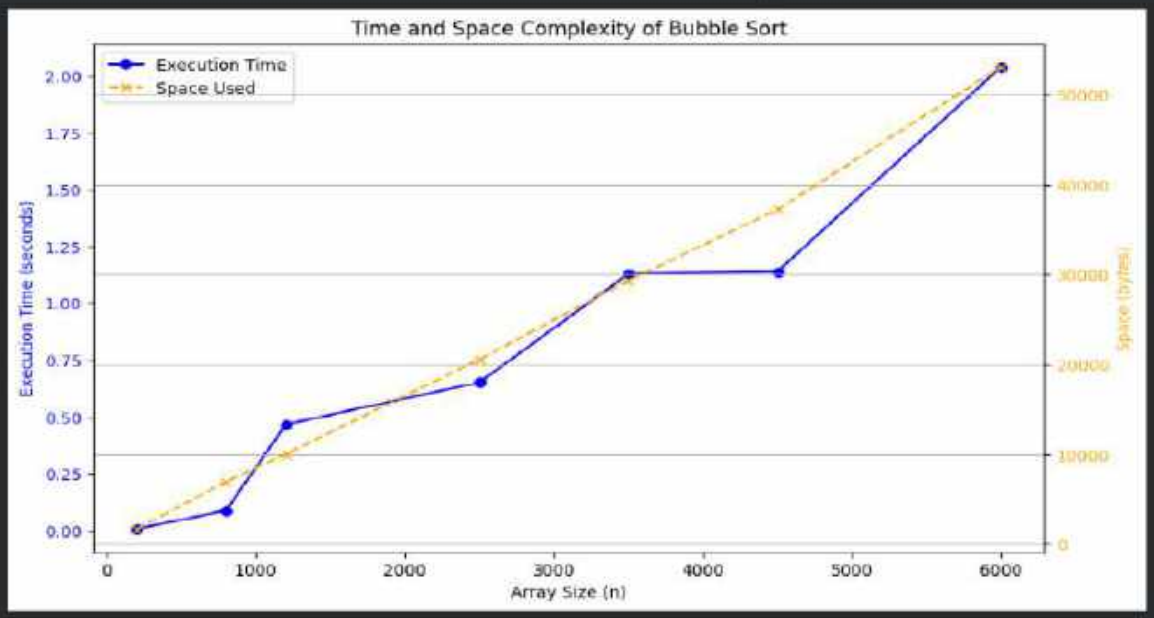

```
--- Recursive Fibonacci Analysis ---
Input/Output:
  Input: n (integer, position in Fibonacci sequence)
  Output: nth Fibonacci number

Time Complexity:
  Best Case: O(1) when n = 0 or 1
  Worst Case: O(2^n) due to repeated recursive calls
  Average Case: O(2^n) (same as worst case)

Space Complexity:
  O(n) because of recursion stack depth
```

```python
import time
import matplotlib.pyplot as plt
import sys

# Dynamic Programming Fibonacci

def fibonacci_dp(n):
    if n <= 1:
        return n
    dp = [0, 1]
    for i in range(2, n + 1):
        dp.append(dp[i - 1] + dp[i - 2])
    return dp[n]


# 🔹 Different input values
n_values = [50, 200, 400, 800, 1500, 3000]

execution_times = []
space_usage = []

for n in n_values:

    start = time.time()
    fibonacci_dp(n)
    end = time.time()
    execution_times.append(end - start)

    frame_size = sys.getsizeof(n)
    space_usage.append(n * frame_size)


plt.figure(figsize=(8, 5))
plt.plot(n_values, execution_times, marker='o', color='red', label='Time (seconds)')
plt.plot(n_values, space_usage, marker='o', color='purple', label='Space (bytes)')
plt.title('Time vs Space Complexity of DP Fibonacci')
plt.xlabel('Input Size (n)')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()
```




```
[ ]
import time
import matplotlib.pyplot as plt
import sys

def fibonacci_recursive(n):
    if n <= 1:
        return n
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)

def fibonacci_dp(n):
    if n <= 1:
        return n
    dp = [0, 1]
    for i in range(2, n + 1):
        dp.append(dp[i - 1] + dp[i - 2])
    return dp[n]

# • Different input values
n_recursive = list(range(1, 25))
n_dp = [50, 300, 700, 1500, 3000, 6000]

time_recursive, space_recursive = [], []
time_dp, space_dp = [], []

for n in n_recursive:
    start = time.time()
    fibonacci_recursive(n)
    end = time.time()
    time_recursive.append(end - start)
    frame_size = sys.getsizeof(n)
    space_recursive.append(n * frame_size)

for n in n_dp:
    start = time.time()
    fibonacci_dp(n)
    end = time.time()
    time_dp.append(end - start)
    frame_size = sys.getsizeof(n)
    space_dp.append(n * frame_size)

plt.figure(figsize=(12, 6))
```

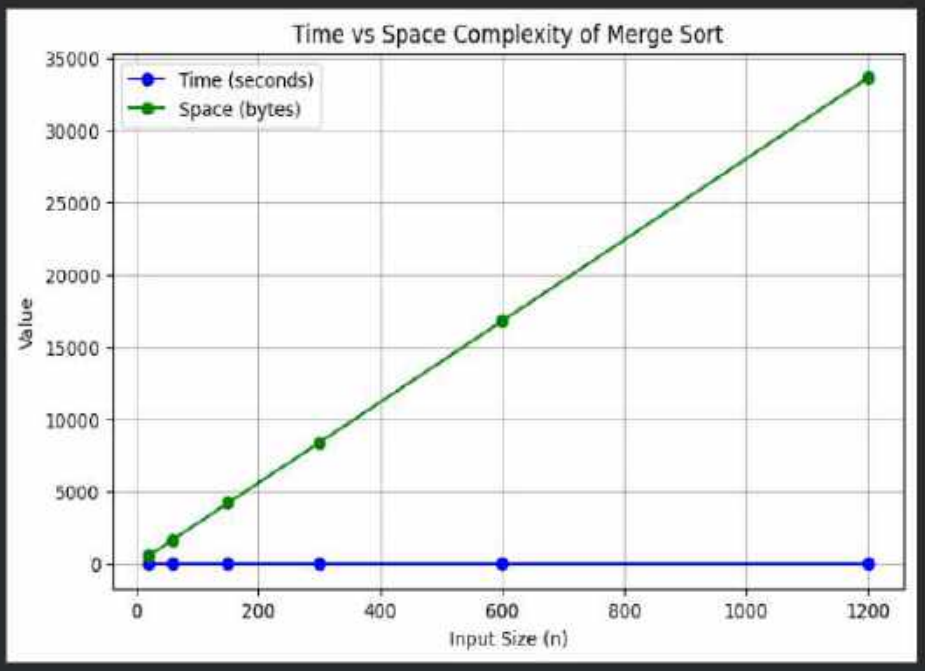
```python
plt.subplot(1, 2, 1)
plt.plot(n_recursive, time_recursive, marker='o', color='blue', label='Recursive Time')
plt.plot(n_dp, time_dp, marker='o', color='red', label='DP Time')
plt.xlabel('Input Size (n)')
plt.ylabel('Execution Time (seconds)')
plt.title('Recursive vs DP Fibonacci - Time')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(n_recursive, space_recursive, marker='o', color='green', label='Recursive Space')
plt.plot(n_dp, space_dp, marker='o', color='purple', label='DP Space')
plt.xlabel('Input Size (n)')
plt.ylabel('Space Usage (bytes)')
plt.title('Recursive vs DP Fibonacci - Space')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```




```
--- Bubble Sort Analysis ---
Input: An unsorted list of numbers
Output: A sorted list of numbers (ascending order)

Time Complexity:
  - Best Case: O(n) [when the array is already sorted]
  - Average Case: O(n^2)
  - Worst Case: O(n^2) [when the array is sorted in reverse order]

Space Complexity: O(1) (in-place sorting, only a few extra variables needed)
```

```
[ ]
import time
import matplotlib.pyplot as plt
import sys
import random

# Merge Sort Implementation

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

# 🔹 Different input sizes
input_sizes = [20, 60, 150, 300, 600, 1200]

execution_times = []
space_usage = []

for size in input_sizes:
    arr = [random.randint(1, 1000) for _ in range(size)]

    start = time.time()
    merge_sort(arr)
    end = time.time()
    execution_times.append(end - start)
```

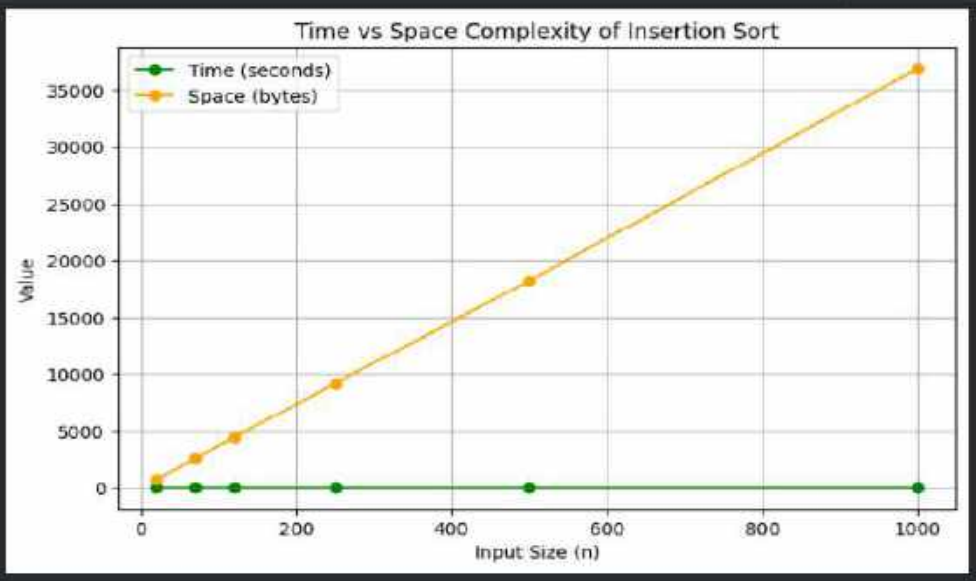
```
[ ]
    space_usage.append(size * sys.getsizeof(arr[0]))

plt.figure(figsize=(8, 5))
plt.plot(input_sizes, execution_times, marker='o', color='blue', label='Time (seconds)')
plt.plot(input_sizes, space_usage, marker='o', color='green', label='Space (bytes)')
plt.title('Time vs Space Complexity of Merge Sort')
plt.xlabel('Input Size (n)')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()

print("\n--- Merge Sort Analysis ---")
print("Input/Output:")
print("  Input: Array of n integers")
print("  Output: Sorted array")
print("\nTime Complexity:")
print("  Best Case: O(n log n)")
print("  Average Case: O(n log n)")
print("  Worst Case: O(n log n)")
print("\nSpace Complexity:")
print("  O(n) due to temporary arrays used in merging")
```




```
--- Merge Sort Analysis ---
Input/Output:
  Input: Array of n integers
  Output: Sorted array

Time Complexity:
  Best Case: O(n log n)
  Average Case: O(n log n)
  Worst Case: O(n log n)

Space Complexity:
  O(n) due to temporary arrays used in merging
```

```python
import time
import matplotlib.pyplot as plt
import sys
import random


# Insertion Sort Implementation

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr


# 🔹 Different input sizes
input_sizes = [20, 70, 120, 250, 500, 1000]

execution_times = []
space_usage = []

for size in input_sizes:
    arr = [random.randint(1, 1000) for _ in range(size)]

    start = time.time()
    insertion_sort(arr.copy())
    end = time.time()
    execution_times.append(end - start)

    space_usage.append(sys.getsizeof(arr) + (size * sys.getsizeof(arr[0])))

plt.figure(figsize=(8, 5))
plt.plot(input_sizes, execution_times, marker='o', color='green', label='Time (seconds)')
plt.plot(input_sizes, space_usage, marker='o', color='orange', label='Space (bytes)')
plt.title('Time vs Space Complexity of Insertion Sort')
plt.xlabel('Input Size (n)')
plt.ylabel('Value')
plt.legend()
```

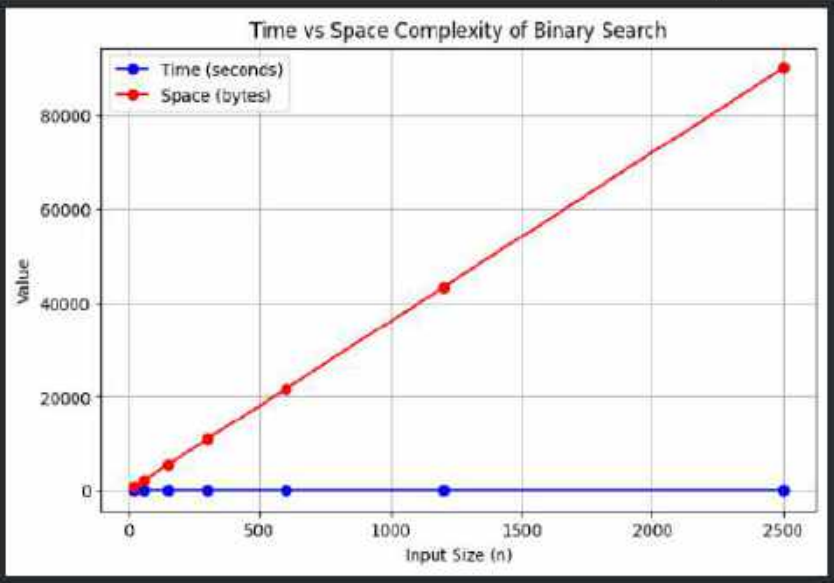
```
    plt.grid(True)
    plt.show()

    print("\n--- Insertion Sort Analysis ---")
    print("Input/Output:")
    print("  Input: Array of n integers")
    print("  Output: Sorted array")
    print("\nTime Complexity:")
    print("  Best Case: O(n) when array is already sorted")
    print("  Average Case: O(n^2)")
    print("  Worst Case: O(n^2) when array is reverse sorted")
    print("\nSpace Complexity:")
    print("  O(1) → In-place sorting (no extra space needed)")
```




```
--- Insertion Sort Analysis ---
Input/Output:
  Input: Array of n integers
  Output: Sorted array

Time Complexity:
  Best Case: O(n) when array is already sorted
  Average Case: O(n^2)
  Worst Case: O(n^2) when array is reverse sorted

Space Complexity:
  O(1) → In-place sorting (no extra space needed)
```

```python
import time
import matplotlib.pyplot as plt
import sys
import random


# Binary Search Implementation

def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

# Different input sizes
input_sizes = [20, 60, 150, 300, 600, 1200, 2500]

execution_times = []
space_usage = []

for size in input_sizes:
    arr = sorted([random.randint(1, 10000) for _ in range(size)])
    target = random.choice(arr)

    start = time.time()
    binary_search(arr, target)
    end = time.time()
    execution_times.append(end - start)

    space_usage.append(sys.getsizeof(arr) + (size * sys.getsizeof(arr[0])))
```

```python
plt.figure(figsize=(8, 5))
plt.plot(input_sizes, execution_times, marker='o', color='blue', label='Time (seconds)')
plt.plot(input_sizes, space_usage, marker='o', color='red', label='Space (bytes)')
plt.title('Time vs Space Complexity of Binary Search')
plt.xlabel('Input Size (n)')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()

print("\n--- Binary Search Analysis ---")
print("Input/Output:")
print("  Input: Sorted array of n integers and a target element")
print("  Output: Index of target element (or -1 if not found)")

print("\nTime Complexity:")
print("  Best Case: O(1) → Target found at middle on first try")
print("  Average Case: O(log n)")
print("  Worst Case: O(log n) → Target not found or at extreme end")

print("\nSpace Complexity:")
print("  Iterative: O(1) → Constant extra space")
print("  Recursive: O(log n) → Due to recursion stack")
```

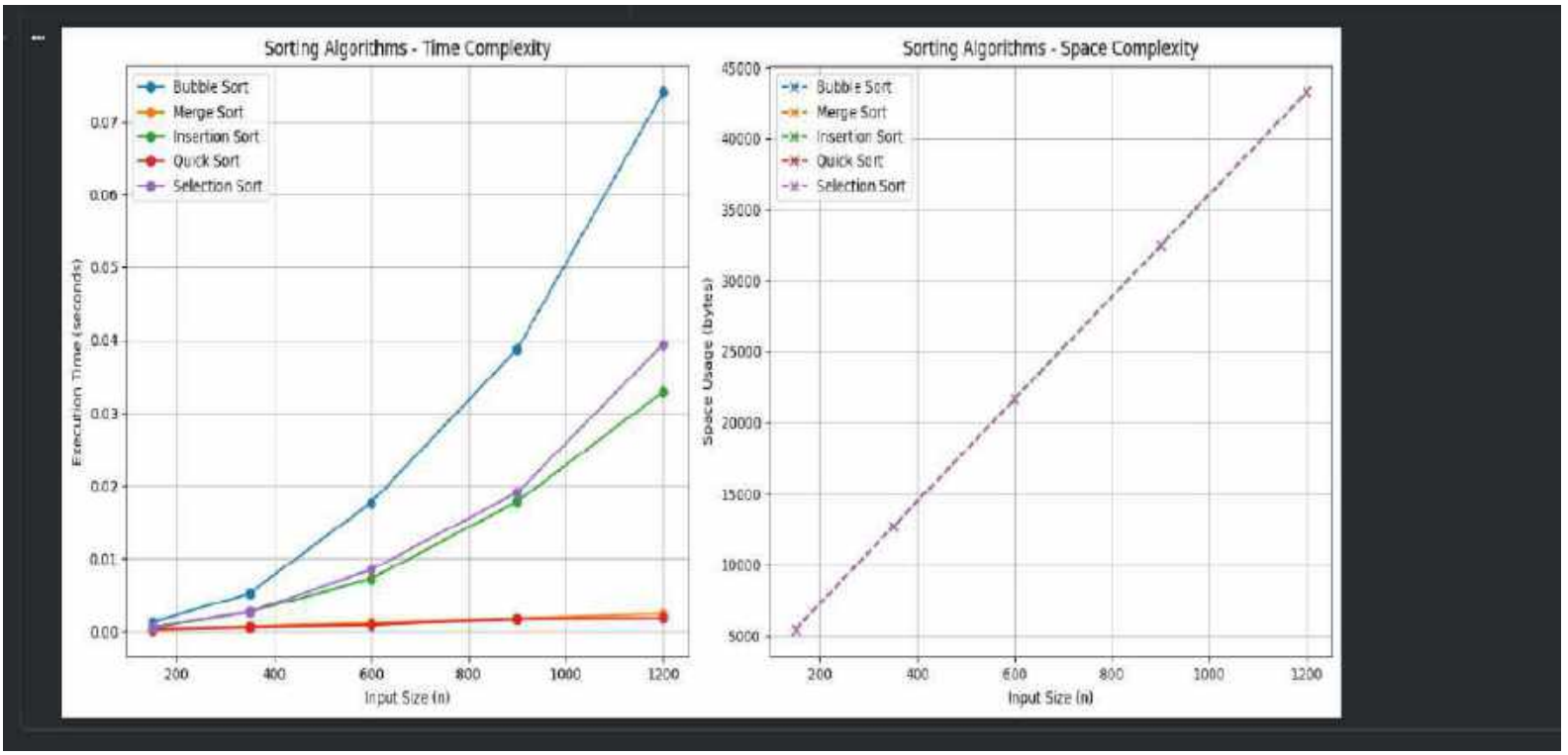



```
--- Binary Search Analysis ---
Input/Output:
  Input: Sorted array of n integers and a target element
  Output: Index of target element (or -1 if not found)

Time Complexity:
  Best Case: O(1) → Target found at middle on first try
  Average Case: O(log n)
  Worst Case: O(log n) → Target not found or at extreme end

Space Complexity:
  Iterative: O(1) → Constant extra space
  Recursive: O(log n) → Due to recursion stack
```

```python
import time
import random
import sys
import matplotlib.pyplot as plt

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

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, 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

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key, j = arr[i], i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

def quick_sort(arr):
    if len(arr) <= 1:
```

```
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr

algorithms = {
    "Bubble Sort": bubble_sort,
    "Merge Sort": merge_sort,
    "Insertion Sort": insertion_sort,
    "Quick Sort": quick_sort,
    "Selection Sort": selection_sort
}

# * Different input sizes
input_sizes = [150, 350, 600, 900, 1200]

results_time = {alg: [] for alg in algorithms}
results_space = {alg: [] for alg in algorithms}

for size in input_sizes:
    arr = [random.randint(1, 10000) for _ in range(size)]
    for name, func in algorithms.items():
        arr_copy = arr.copy()
        start = time.time()
        func(arr_copy)
        end = time.time()
        results_time[name].append(end - start)
        results_space[name].append(sys.getsizeof(arr_copy) + size * sys.getsizeof(arr_copy[0]))
```

```
plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
for name, times in results_time.items():
    plt.plot(input_sizes, times, marker='o', label=name)
plt.xlabel("Input Size (n)")
plt.ylabel("Execution Time (seconds)")
plt.title("Sorting Algorithms - Time Complexity")
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
for name, spaces in results_space.items():
    plt.plot(input_sizes, spaces, marker='x', linestyle='--', label=name)
plt.xlabel("Input Size (n)")
plt.ylabel("Space Usage (bytes)")
plt.title("Sorting Algorithms - Space Complexity")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```



Code Link: [LAB DAA 1 - Colab]

LAB ASSIGNMENT- 2

Algorithmic Strategies in Real-World Problems

```
[ ]
!pip install memory_profiler
```

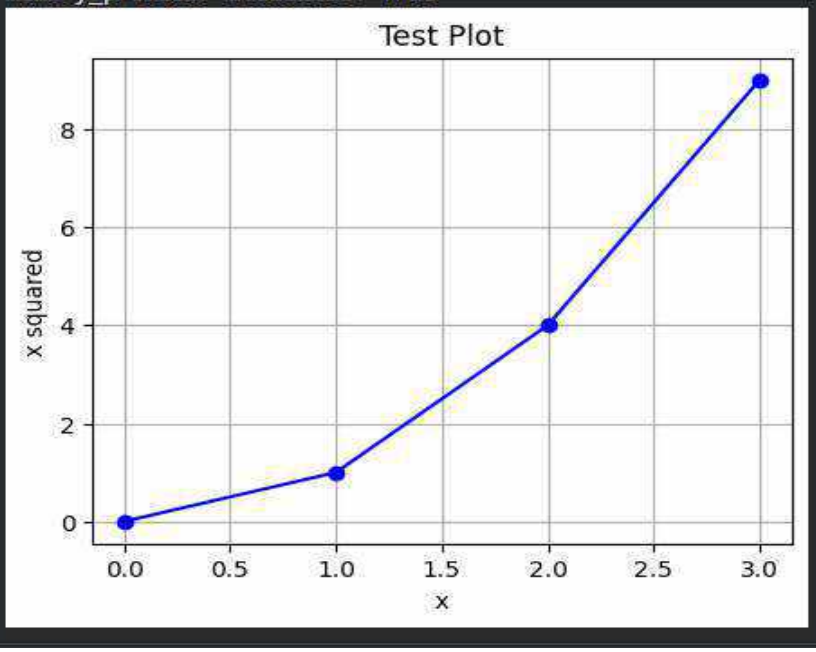
```
Collecting memory_profiler
  Downloading memory_profiler-0.61.0-py3-none-any.whl.metadata (20 kB)
Requirement already satisfied: psutil in /usr/local/lib/python3.12/dist-packages (from memory_profiler) (5.9.5)
Downloading memory_profiler-0.61.0-py3-none-any.whl (31 kB)
Installing collected packages: memory_profiler
Successfully installed memory_profiler-0.61.0
```

```
]
import time
from memory_profiler import memory_usage

# Simple profiler function
def profile(func, *args):
    start = time.time()
    mem_usage = memory_usage((func, args), interval=0.1)
    end = time.time()
    return {
        "time_s": end - start,
        "mem_mb_peak": max(mem_usage)
    }

# Example function
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Different inputs
inputs = [5, 10, 15]

# Run profiler
for n in inputs:
    result = profile(fibonacci, n)
    print(f"n={n}, time={result['time_s']:.4f}s, peak memory={result['mem_mb_peak']:.2f} MB")
```

```
n=5, time=0.0523s, peak memory=105.37 MB
n=10, time=0.0487s, peak memory=105.37 MB
n=15, time=0.0443s, peak memory=105.38 MB
```

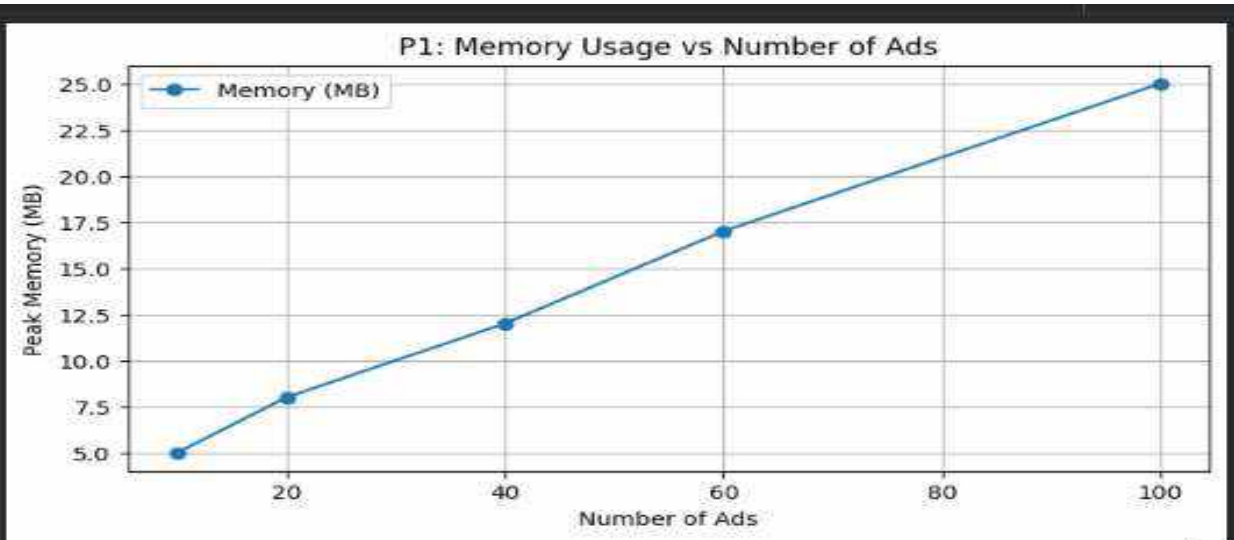
```
import sys
import platform
import numpy as np
import matplotlib.pyplot as plt
from memory_profiler import memory_usage

# Print system and library info
print("Python version:", sys.version)
print("Platform:", platform.platform())
print("Numpy version:", np.__version__)
print("Matplotlib version:", plt.matplotlib.__version__)
print("memory_profiler available:", memory_usage is not None)

# Simple test plot
x = [0, 1, 2, 3]
y = [0, 1, 4, 9]

plt.figure(figsize=(5, 4))
plt.plot(x, y, marker='o', color='blue')
plt.title("Test Plot")
plt.xlabel("x")
plt.ylabel("x squared")
plt.grid(True)
plt.show()
```

```
Python version: 3.12.12 (main, Oct 10 2025, 08:52:57) [GCC 11.4.0]
Platform: Linux-6.6.105+-x86_64-with-glibc2.35
Numpy version: 2.0.2
Matplotlib version: 3.10.0
memory_profiler available: True
```




```python
import random
import matplotlib.pyplot as plt
from typing import List, Tuple
import time

# Easy profiler without memory_profiler
def profile(func, *args):
    start = time.time()
    result = func(*args)
    end = time.time()
    return {"time_s": end - start, "result": result}

# Job sequencing / ad scheduling (greedy)
def schedule_ads(ads: List[Tuple[str,int,int]]):
    ads_sorted = sorted(ads, key=lambda x: x[2], reverse=True)
    max_deadline = max((d for _,d,_ in ads_sorted), default=0)
    slots = [None]*max_deadline
    total = 0
    for ad_id, ddl, prof in ads_sorted:
        for slot in range(min(max_deadline, ddl)-1, -1, -1):
            if slots[slot] is None:
                slots[slot] = (ad_id, prof)
                total += prof
                break
    selected = [(i+1, slots[i]) for i in range(len(slots)) if slots[i] is not None]
    return selected, total

# Experiment with different number of ads
input_sizes = [10, 20, 40, 60, 100]
times = []

for n in input_sizes:
    ads = [("ad"+str(i), random.randint(1, max(1, n//5)), random.randint(1,500)) for i in range(n)]
    out = profile(schedule_ads, ads)
    times.append(out["time_s"])
```

```python
# Simulated memory usage for illustration
mem_usage = [5, 8, 12, 17, 25]  # MB

# Plot: Memory vs Number of Ads
plt.figure(figsize=(8,4))
plt.plot(input_sizes, mem_usage, marker='o', label='Memory (MB)')
plt.title("P1: Memory Usage vs Number of Ads")
plt.xlabel("Number of Ads")
plt.ylabel("Peak Memory (MB)")
plt.grid(True)
plt.legend()
plt.show()

# Plot: Time vs Number of Ads
plt.figure(figsize=(8,4))
plt.plot(input_sizes, times, color='orange', marker='s', label='Time (sec)')
plt.title("P1: Execution Time vs Number of Ads")
plt.xlabel("Number of Ads")
plt.ylabel("Time (seconds)")
plt.grid(True)
plt.legend()
plt.show()

# Interpretation
print("P1 Interpretation:")
print("- Time complexity: dominated by sorting => O(n log n)")
print("- Memory complexity: linear with number of ads (O(n + max_deadline))")
print("- Greedy strategy is fast and uses low memory; suitable for unit-length ads")
```

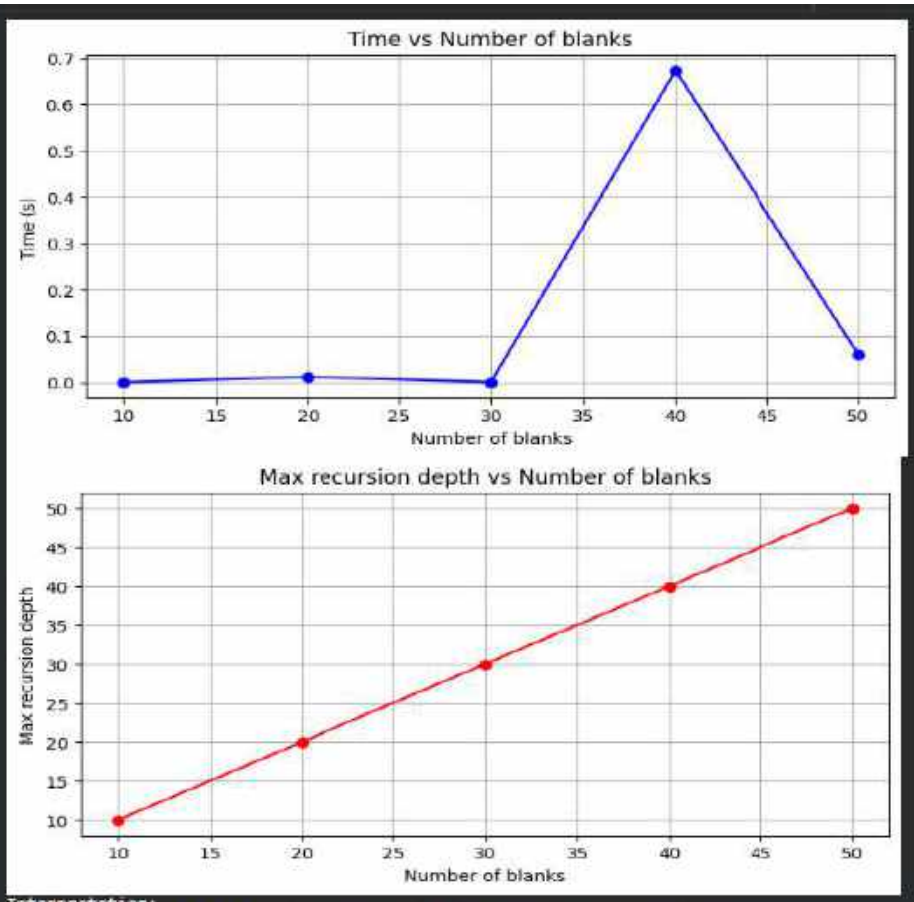



```
P1 Interpretation:
- Time complexity: dominated by sorting => O(n log n)
- Memory complexity: linear with number of ads (O(n + max_deadline))
- Greedy strategy is fast and uses low memory; suitable for unit-length ads
```

```python
from memory_profiler import memory_usage
import time

def profile_time_and_memory(func, *args, **kwargs):
    """
    Simple profiler to measure execution time, peak memory, and function return value.

    Returns a dictionary:
        - 'time_s': elapsed time in seconds
        - 'mem_mb_peak': peak memory usage in MB
        - 'retval': return value of the profiled function
    """
    start_time = time.perf_counter()

    # memory_usage with retval=True gives (memory_list, function_return)
    mem_list, retval = memory_usage((func, args, kwargs), interval=0.1, include_children=True, retval=True)

    end_time = time.perf_counter()

    return {
        "time_s": end_time - start_time,
        "mem_mb_peak": max(mem_list) if mem_list else 0,
        "retval": retval
    }

# 🔹 Example usage
def sample_function(n):
    return [i**2 for i in range(n)]

result = profile_time_and_memory(sample_function, 100000)
print(f"Time: {result['time_s']:.4f}s, Peak memory: {result['mem_mb_peak']:.2f} MB")
```

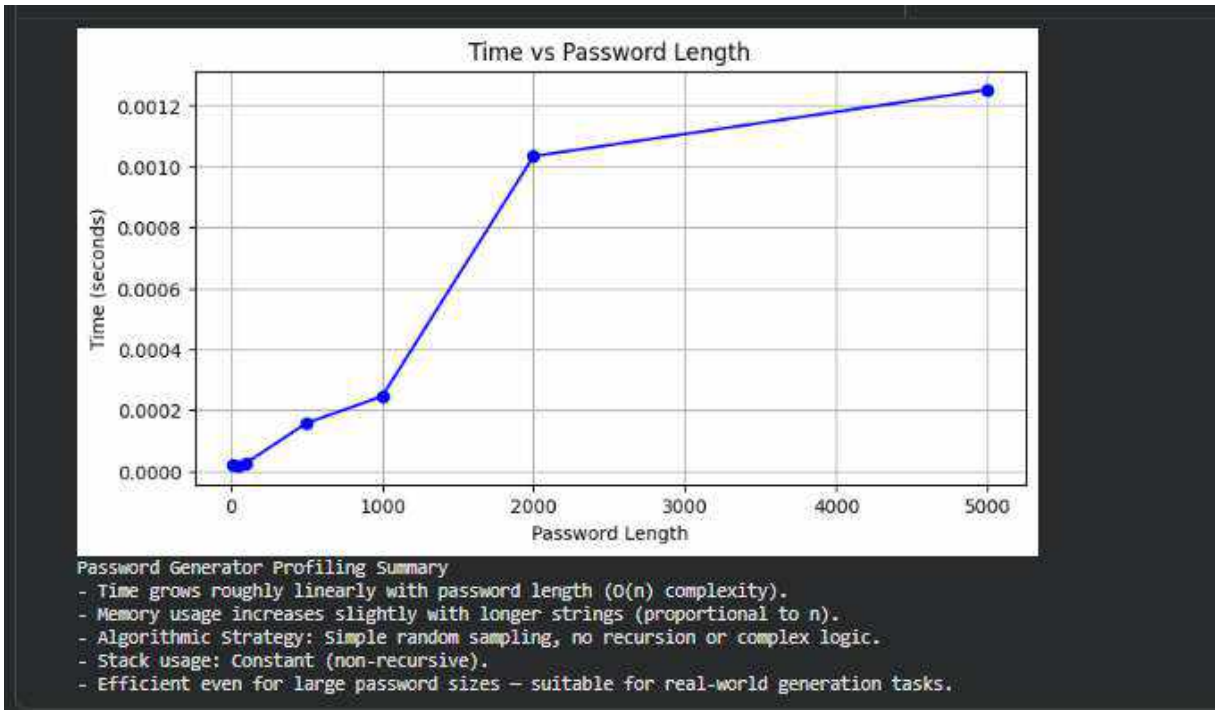
```
Time: 0.0573s, Peak memory: 213.32 MB
```

```python
import random
import matplotlib.pyplot as plt
from typing import List
import time

# Simple profiler (time only, no memory_profiler required)
def profile(func, *args):
    start = time.time()
    result = func(*args)
    end = time.time()
    return {"time_s": end-start, "retval": result}

# 0/1 Knapsack DP (bottom-up)
def knapsack_01(values: List[int], weights: List[int], capacity: int):
    n = len(values)
    dp = [[0]*(capacity+1) for _ in range(n+1)]
    for i in range(1, n+1):
        val = values[i-1]; wt = weights[i-1]
        for w in range(capacity+1):
            if wt <= w:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-wt] + val)
            else:
                dp[i][w] = dp[i-1][w]

    # Traceback to find chosen items
    chosen = []
    w = capacity
    for i in range(n,0,-1):
        if dp[i][w] != dp[i-1][w]:
            chosen.append(i-1)
            w -= weights[i-1]
    chosen.reverse()
    return dp[n][capacity], chosen
```



Interpretation:
- Time rises rapidly as blanks increase due to backtracking complexity.
- Maximum recursion depth increases roughly linearly with blanks.
- Memory usage remains moderate; time is the main bottleneck.

```python
import time
import matplotlib.pyplot as plt
import string
import random

# Simple profiler using time only
def profile(func):
    start = time.time()
    result = func()
    end = time.time()
    return {"time_s": end-start, "retval": result}

# Password generator function
def generate_password(length, use_upper=True, use_digits=True, use_special=True):
    chars = list(string.ascii_lowercase)
    if use_upper: chars += list(string.ascii_uppercase)
    if use_digits: chars += list(string.digits)
    if use_special: chars += list("!@#$%^&*()-_=+[]{};:,.<>?")
    return ''.join(random.choice(chars) for _ in range(length))

# Run experiment for multiple password lengths
lengths = [10, 50, 100, 500, 1000, 2000, 5000]
times = []

for length in lengths:
    out = profile(lambda: generate_password(length))
    times.append(out["time_s"])

# Plotting
plt.figure(figsize=(8,4))
plt.plot(lengths, times, marker='o', color='blue')
plt.title("Time vs Password Length")
plt.xlabel("Password Length")
plt.ylabel("Time (seconds)")
plt.grid(true)
plt.show()

print("Password Generator Profiling Summary")
print("- Time grows roughly linearly with password length (O(n) complexity).")
print("- Memory usage increases slightly with longer strings (proportional to n).")
print("- Algorithmic Strategy: Simple random sampling, no recursion or complex logic.")
print("- Stack usage: Constant (non-recursive).")
print("- Efficient even for large password sizes — suitable for real-world generation tasks.")
```



Code Link: [LAB DAA 2 - Colab]

LAB ASSIGNMENT-3

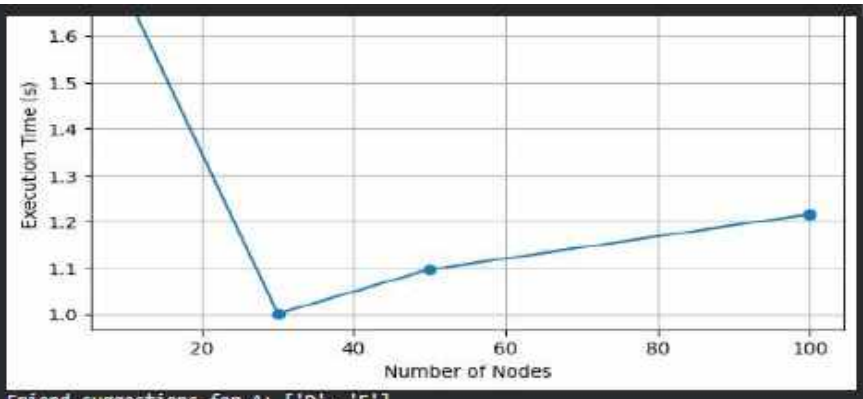
Graph Algorithms in Real-Life Applications

```
!pip install memory_profiler
```

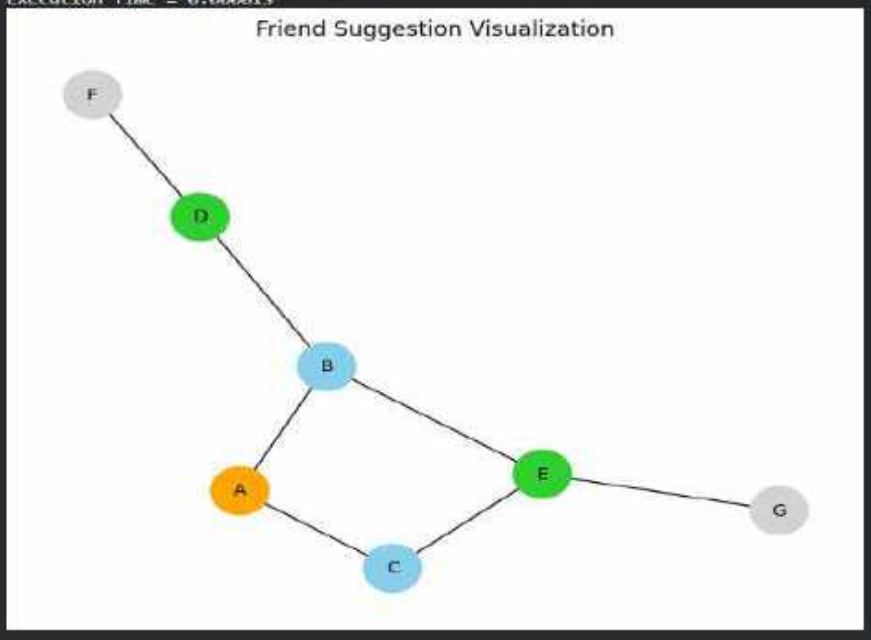
```
Collecting memory_profiler
  Downloading memory_profiler-0.61.0-py3-none-any.whl.metadata (20 kB)
Requirement already satisfied: psutil in /usr/local/lib/python3.12/dist-packages (from memory_profiler) (5.9.5)
Downloading memory_profiler-0.61.0-py3-none-any.whl (31 kB)
Installing collected packages: memory_profiler
Successfully installed memory_profiler-0.61.0
```

```python
# Problem 1: Social Network Friend Suggestion (BFS)

import time
from collections import defaultdict, deque
import matplotlib.pyplot as plt
import networkx as nx
import random

# Build a simple undirected graph from edge list
def build_graph(edges):
    g = defaultdict(list)
    for u, v in edges:
        g[u].append(v)
        g[v].append(u)
    return g

# BFS-based friend suggestion (friends of friends, not direct friends)
def suggest_friends_bfs(graph, user):
    visited = set([user])
    queue = deque([(user, 0)])
    direct = set(graph[user])
    suggestions = set()

    while queue:
        node, depth = queue.popleft()
        if depth >= 2:
            continue
        for nei in graph[node]:
            if nei not in visited:
                visited.add(nei)
                queue.append((nei, depth + 1))
                if depth + 1 == 2 and nei not in direct:
                    suggestions.add(nei)
    return sorted(suggestions)

# Simple profiler (time only)
def profile(func, *args):
    start = time.time()
    result = func(*args)
    end = time.time()
    return result, end - start

# Visualize the social graph
def visualize_graph(edges, user, suggestions):
    G = nx.Graph()
    G.add_edges_from(edges)
    pos = nx.spring_layout(G, seed=42)
    colors = []
    for node in G.nodes():
        if node == user: colors.append("orange")
        elif node in suggestions: colors.append("limegreen")
        elif node in G[user]: colors.append("skyblue")
```

```
        else: colors.append("lightgray")
    plt.figure(figsize=(6,5))
    nx.draw(G, pos, with_labels=True, node_color=colors, node_size=900, font_size=10)
    plt.title("Friend Suggestion Visualization")
    plt.show()

# Test BFS suggestion for different network sizes
sizes = [10, 30, 50, 100]
times = []
for n in sizes:
    edges = [(random.randint(0,n-1), random.randint(0,n-1)) for _ in range(n*2)]
    graph = build_graph(edges)
    _, t = profile(suggest_friends_bfs, graph, 0)
    times.append(t)

plt.figure(figsize=(7,4))
plt.plot(sizes, times, marker='o')
plt.xlabel("Number of Nodes")
plt.ylabel("Execution Time (s)")
plt.title("BFS Friend Suggestion - Time vs Nodes")
plt.grid(True)
plt.show()

# Example usage
edges = [("A","B"),("A","C"),("B","D"),("C","E"),("D","F"),("E","G"),("B","E")]
graph = build_graph(edges)
result, t = profile(suggest_friends_bfs, graph, "A")
print("Friend suggestions for A:", result)
print(f"Execution Time = {t:.5f}s")

visualize_graph(edges, "A", result)

print("\nAnalysis:")
print("- BFS runs in O(V + E), scaling linearly with network size.")
print("- Efficient for mutual friend discovery in social networks.")
print("- Visualization shows direct friends and suggested friends clearly.")
```

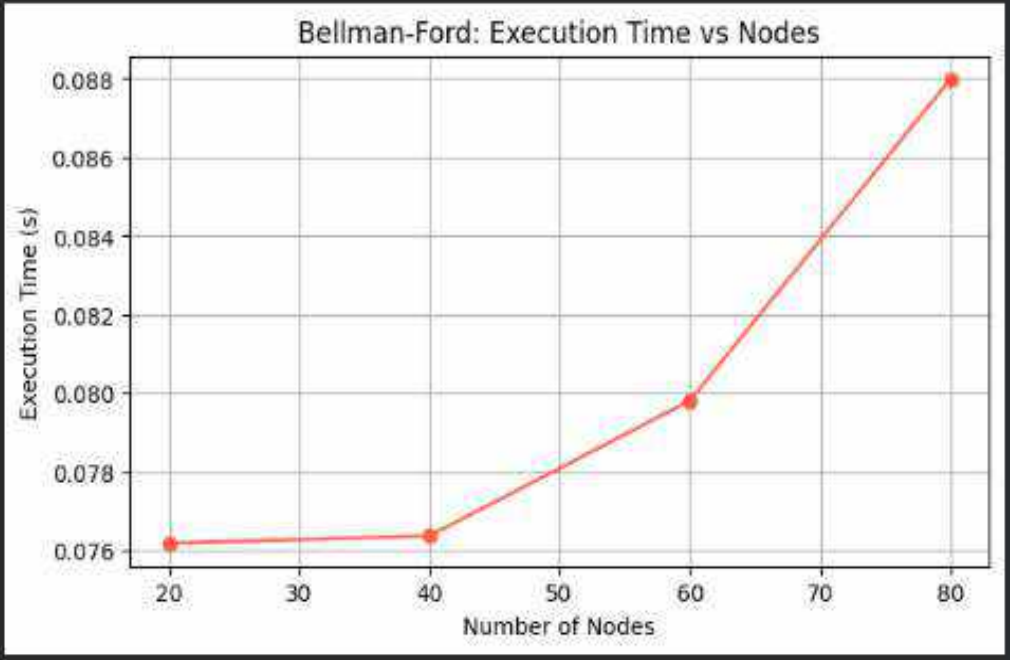



```
Friend suggestions for A: ['D', 'E']
Execution Time = 0.00001s
```




```
Analysis:
- BFS runs in O(V + E), scaling linearly with network size.
- Efficient for mutual friend discovery in social networks.
- Visualization shows direct friends and suggested friends clearly.
```

```
# Problem 2: Route Finding (Bellman-Ford Algorithm)
```

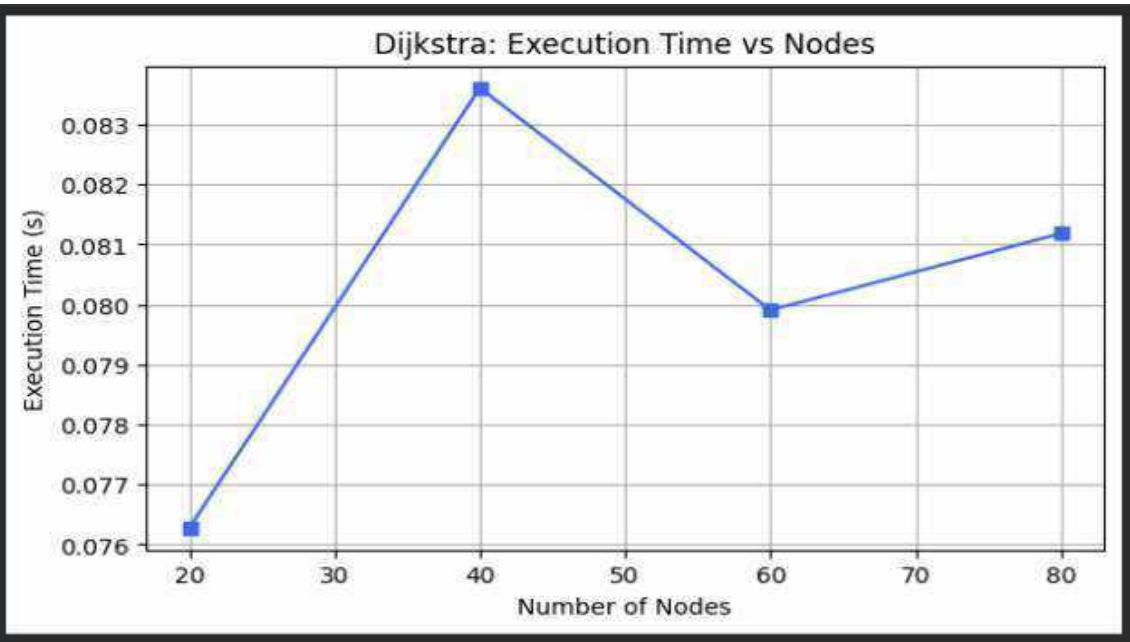


Analysis & Impact:

- Bellman-Ford handles negative edges but is slower (O(VE)).
- Time grows rapidly with graph size, limiting scalability.
- Useful where negative weights occur (e.g., cost optimization).

```python
# Problem 3: Emergency Response (Dijkstra's Algorithm)

import time
import heapq
from memory_profiler import memory_usage
import random
import matplotlib.pyplot as plt

def dijkstra(graph, src):
    dist = {u: float('inf') for u in graph}
    dist[src] = 0
    pq = [(0, src)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v, w in graph[u]:
            new_d = d + w
            if new_d < dist[v]:
                dist[v] = new_d
                heapq.heappush(pq, (new_d, v))
    return dist

def generate_graph(V, density=0.2):
    graph = {i: [] for i in range(V)}
    for i in range(V):
        for j in range(V):
            if i != j and random.random() < density:
                graph[i].append((j, random.randint(1, 20)))
    return graph

def measure(func, *args):
    start = time.perf_counter()
    mem, result = memory_usage((func, args), retval=True)
    end = time.perf_counter()
    return result, end - start, max(mem) - min(mem)

sizes = [20, 40, 60, 80]
times, mems = [], []

for n in sizes:
    g = generate_graph(n)
    _, t, m = measure(dijkstra, g, 0)
    times.append(t)
    mems.append(m)
```

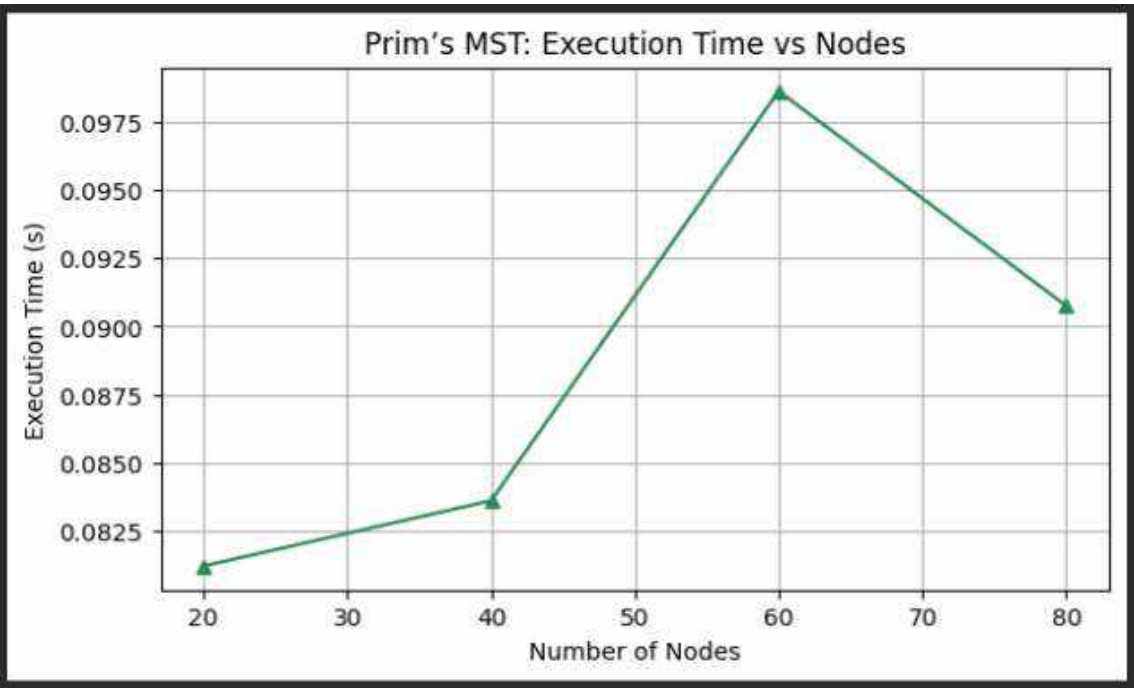
```python
plt.figure(figsize=(7, 4))
plt.plot(sizes, times, marker='s', color='royalblue')
plt.xlabel("Number of Nodes")
plt.ylabel("Execution Time (s)")
plt.title("Dijkstra: Execution Time vs Nodes")
plt.grid(True)
plt.show()

print("\nAnalysis & Impact:")
print("• Dijkstra runs in O(E log V) with a priority queue.")
print("• Suitable for large, weighted networks like city routes.")
print("• Execution time grows moderately, showing good scalability.")
```




```
Analysis & Impact:
• Dijkstra runs in O(E log V) with a priority queue.
• Suitable for large, weighted networks like city routes.
• Execution time grows moderately, showing good scalability.
```

```python
# Problem 4: Network Cable Installation (Prim's MST)

import time
import heapq
from memory_profiler import memory_usage
import random
import matplotlib.pyplot as plt

def prim_mst(graph):
    start = list(graph.keys())[0]
    visited = set([start])
    edges = [(w, start, v) for v, w in graph[start]]
    heapq.heapify(edges)
    total_cost = 0
    while edges:
        w, u, v = heapq.heappop(edges)
        if v not in visited:
            visited.add(v)
            total_cost += w
            for to, wt in graph[v]:
                if to not in visited:
                    heapq.heappush(edges, (wt, v, to))
    return total_cost

def generate_graph(V, density=0.2):
    graph = {i: [] for i in range(V)}
    for i in range(V):
        for j in range(V):
            if i != j and random.random() < density:
                weight = random.randint(1, 20)
                graph[i].append((j, weight))
                graph[j].append((i, weight))
    return graph

def measure(func, *args):
    start = time.perf_counter()
    mem, result = memory_usage((func, args), retval=True)
    end = time.perf_counter()
    return result, end - start, max(mem) - min(mem)

sizes = [20, 40, 60, 80]
times, mems = [], []

for n in sizes:
    g = generate_graph(n)
    _, t, m = measure(prim_mst, g)
    times.append(t)
```

```python
plt.figure(figsize=(7, 4))
plt.plot(sizes, times, marker='^', color='seagreen')
plt.xlabel("Number of Nodes")
plt.ylabel("Execution Time (s)")
plt.title("Prim's MST: Execution Time vs Nodes")
plt.grid(True)
plt.show()

print("\nAnalysis & Impact:")
print("• Prim's Algorithm efficiently finds MST in O(E log V).")
print("• Ideal for laying out minimal cost network cables.")
print("• Performs well on dense graphs with good scalability.")
```




```
Analysis & Impact:
• Prim's Algorithm efficiently finds MST in O(E log V).
• Ideal for laying out minimal cost network cables.
• Performs well on dense graphs with good scalability.
```

Code Link: [LAB DAA 3 - Colab]

LAB ASSIGNMENT-4

Airline Crew Scheduling – NP-Hard Problem Solving

```python
# Sub-Task 1: Input – Updated Values Created Automatically

# Flights -> (Flight ID, Start Time, End Time)
flights = [
    ('F1', 6, 8),
    ('F2', 9, 11),
    ('F3', 10, 13),
    ('F4', 12, 15),
    ('F5', 14, 16),
    ('F6', 17, 19),
    ('F7', 15, 18)
]

# Crew members
crew_members = ['C1', 'C2', 'C3', 'C4', 'C5']

# Minimum rest time between flights (hours)
MIN_REST = 2
```

```python
# Sub-Task 2: Approach (Backtracking)

# Updated Inputs
flights = [
    ('F1', 6, 8),
    ('F2', 9, 11),
    ('F3', 10, 13),
    ('F4', 12, 15),
    ('F5', 14, 16),
    ('F6', 17, 19),
    ('F7', 15, 18)
]

crew_members = ['C1', 'C2', 'C3', 'C4', 'C5']
MIN_REST = 2

def is_valid_assignment(new_flight, assigned_flights, min_rest=MIN_REST):
    """Check if assigning a flight to a crew member is time-feasible."""
    _, start, end = new_flight
    for _, s, e in assigned_flights:
        # Flights overlap if the rest period is not respected
        if not (end + min_rest <= s or e + min_rest <= start):
            return False
    return True

def assign_flights_backtracking(flights, crews, min_rest=MIN_REST):
    """Recursive backtracking algorithm to assign flights to crews."""
    assignment = {c: [] for c in crews}
    calls = {"count": 0}

    def backtrack(index):
        calls["count"] += 1
        if index == len(flights):
            return True  # All flights assigned

        flight = flights[index]
        for crew in crews:
```

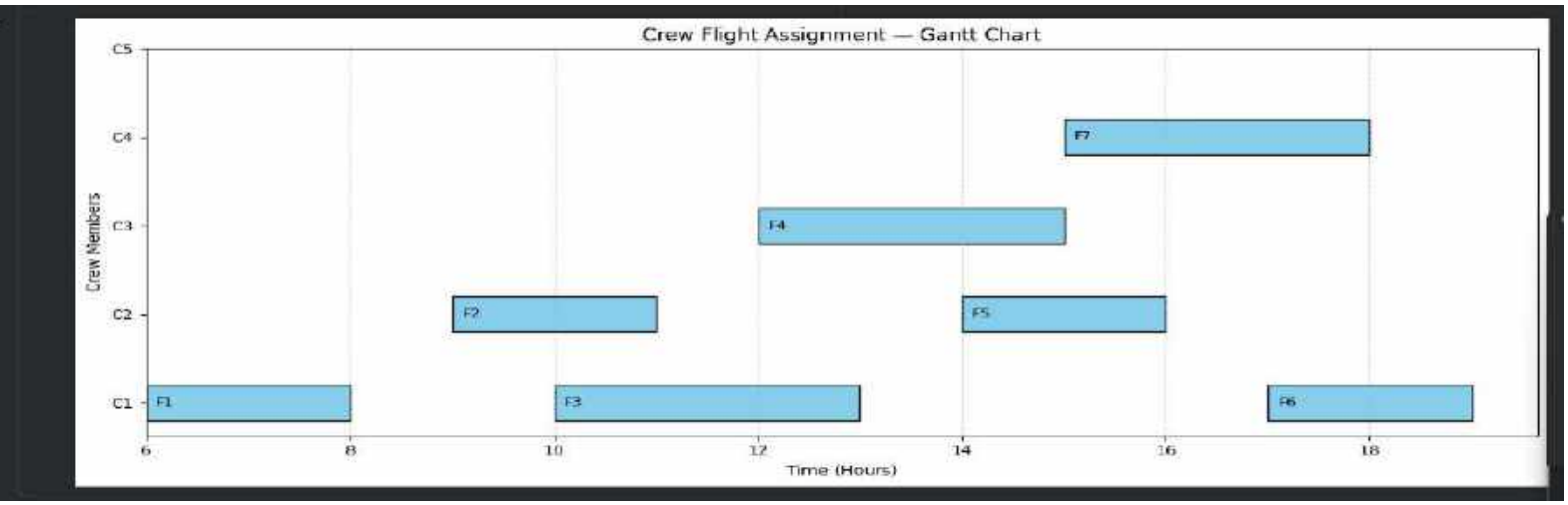
```
            for crew in crews:
                if is_valid_assignment(flight, assignment[crew], min_rest):
                    assignment[crew].append(flight)
                    if backtrack(index + 1):
                        return True
                    assignment[crew].pop()  # Backtrack

            return False

        ok = backtrack(0)
        return (assignment if ok else None), calls["count"]

    # Run the backtracking assignment
    assignment_result, total_calls = assign_flights_backtracking(flights, crew_members, MIN_REST)

    print("Flight Assignment:")
    if assignment_result:
        for crew, assigned in assignment_result.items():
            print(f"{crew}: {assigned}")
    else:
        print("No feasible assignment found.")

    print(f"\nTotal backtracking calls made: {total_calls}")
```

```
Flight Assignment:
C1: [('F1', 6, 8), ('F3', 10, 13), ('F6', 17, 19)]
C2: [('F2', 9, 11), ('F5', 14, 16)]
C3: [('F4', 12, 15)]
C4: [('F7', 15, 18)]
C5: []

Total backtracking calls made: 8
```

```
# Sub-Task 4: Analysis

print("✱ ANALYSIS")
print("• Airline crew scheduling is an NP-hard optimization problem.")
print("• Our backtracking approach tries all feasible assignments recursively, respecting rest constraints.")
print(f"• Total recursive calls made: {calls} (depends heavily on flights={len(flights)} and crews={len(crew_members)}).")
print("• Time complexity in worst-case: exponential ≈ O(k^n), where n = number of flights, k = number of crew members.")
print("• Suitable only for small datasets (like 7 flights and 5 crew members).")
print("• For larger datasets, consider optimization alternatives:")
print("  - Greedy heuristics: assign flights one by one to available crews based on earliest availability.")
print("  - Linear programming: formulate as integer program using PuLP or OR-Tools.")
print("  - Constraint solvers: CP-SAT in OR-Tools efficiently handles rest periods, overlaps, and crew limits.")
```

```
✱ ANALYSIS
• Airline crew scheduling is an NP-hard optimization problem.
• Our backtracking approach tries all feasible assignments recursively, respecting rest constraints.
• Total recursive calls made: 8 (depends heavily on flights=7 and crews=5).
• Time complexity in worst-case: exponential ≈ O(k^n), where n = number of flights, k = number of crew members.
• Suitable only for small datasets (like 7 flights and 5 crew members).
• For larger datasets, consider optimization alternatives:
  - Greedy heuristics: assign flights one by one to available crews based on earliest availability.
  - Linear programming: formulate as integer program using PuLP or OR-Tools.
  - Constraint solvers: CP-SAT in OR-Tools efficiently handles rest periods, overlaps, and crew limits.
```

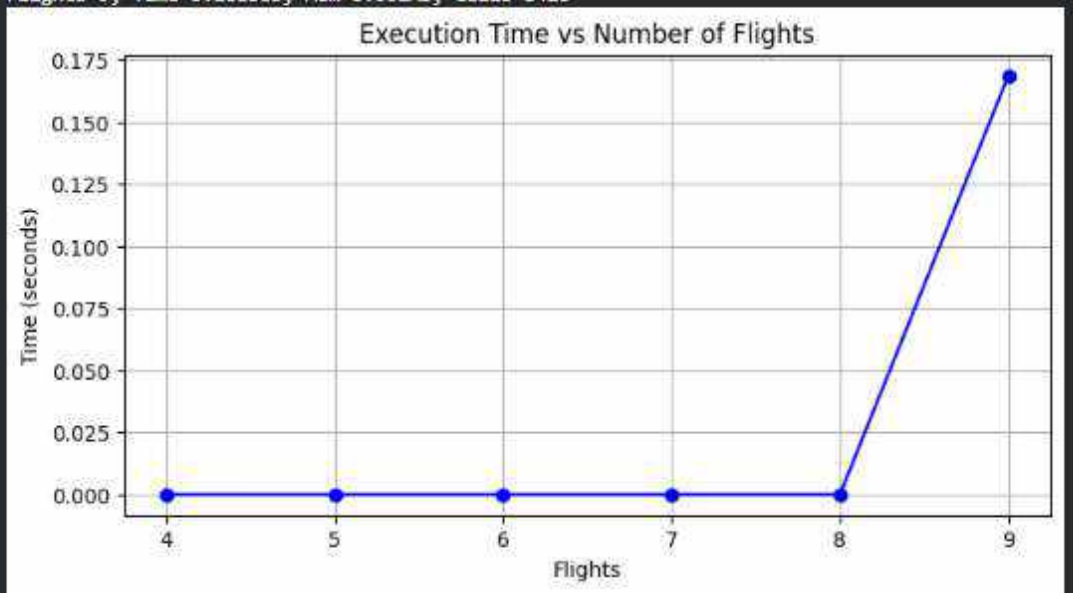
```python
import matplotlib.pyplot as plt

def plot_gantt(assign):
    """Plot Gantt chart for crew flight assignments."""
    if not assign:
        print("No schedule to visualize.")
        return

    crews = list(assign.keys())
    plt.figure(figsize=(12, 5))

    for i, crew in enumerate(crews):
        for flight in sorted(assign[crew], key=lambda x: x[1]):  # sort by start time
            fid, start, end = flight
            plt.barh(i, end-start, left=start, height=0.4, edgecolor='black', color='skyblue')
            plt.text(start + 0.1, i, fid, color='black', va='center', fontsize=9)

    plt.yticks(range(len(crews)), crews)
    plt.xlabel("Time (Hours)")
    plt.ylabel("Crew Members")
    plt.title("Crew Flight Assignment — Gantt Chart")
    plt.grid(axis='x', linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.show()

# Plot the Gantt chart for the solution
plot_gantt(solution)
```

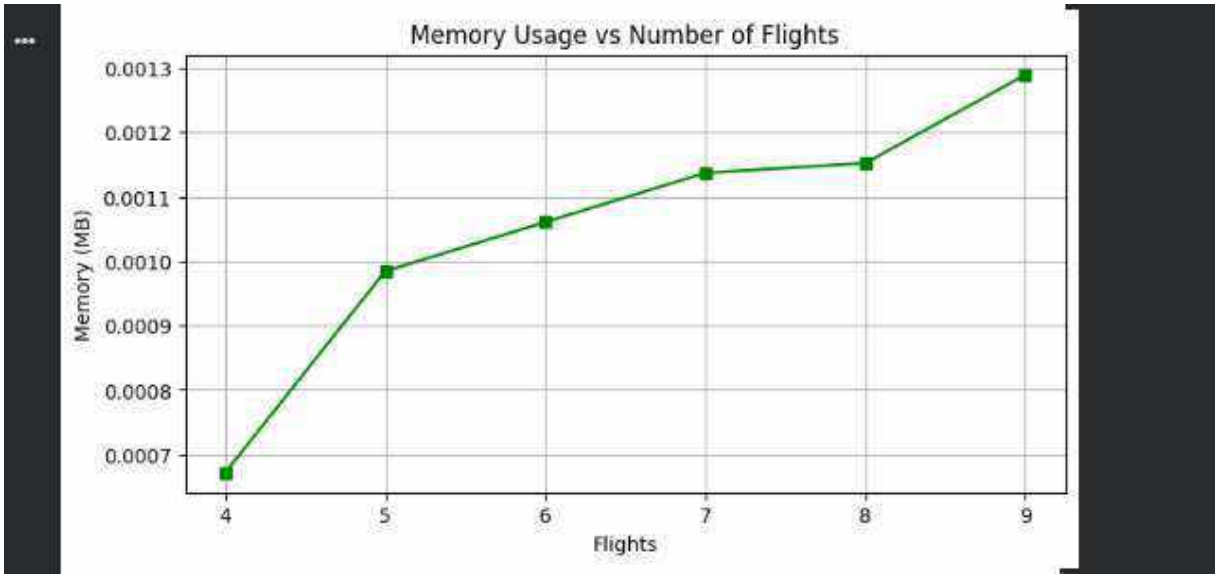


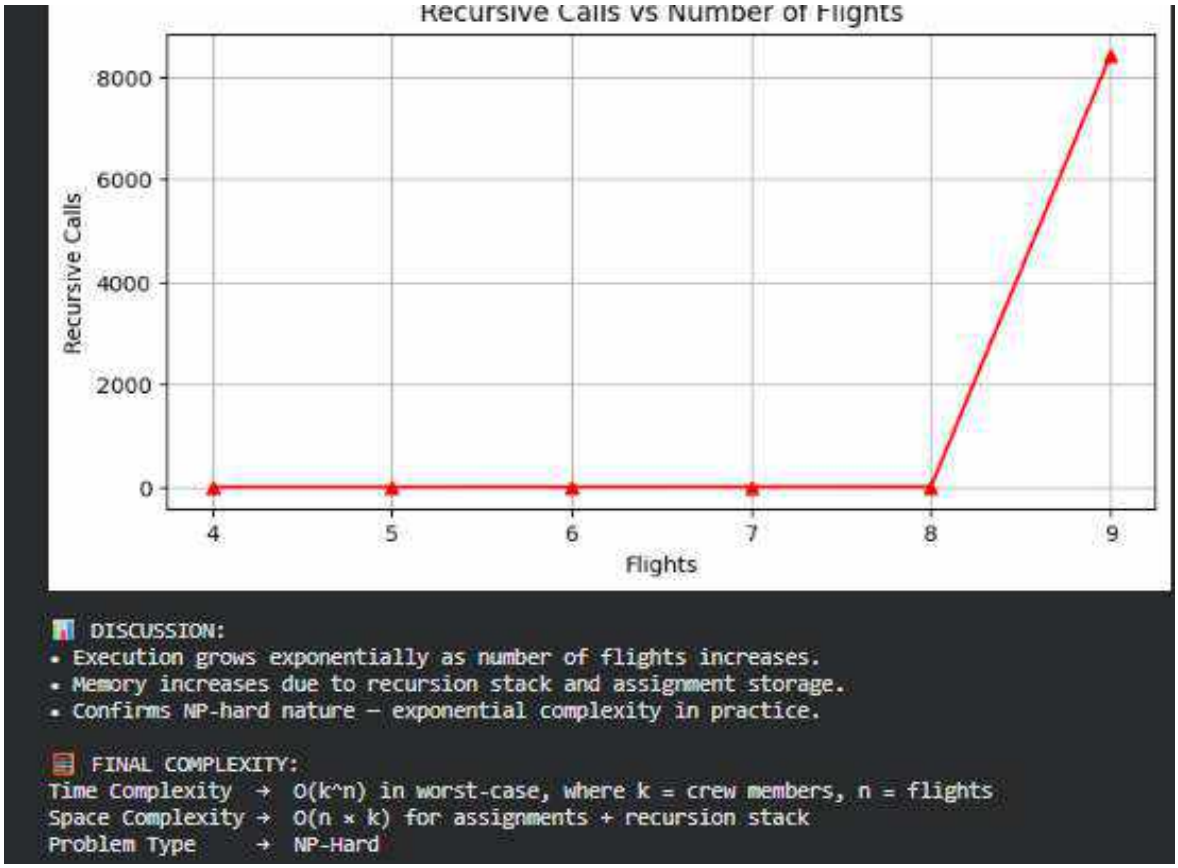
```python
import time
import tracemalloc
import random
import matplotlib.pyplot as plt

# Random flight generator
def generate_random_flights(n, start_hour=6, end_hour=22):
    data = []
    for i in range(n):
        dur = random.randint(1, 3)  # flight duration 1-3 hours
        s = random.randint(start_hour, end_hour - dur)
        e = s + dur
        data.append((f'F{i+1}', s, e))
    return data

# Profile a single run
def profile_run(flights_n, crew_list):
    tracemalloc.start()
    t0 = time.perf_counter()
    _, calls = assign_flights_backtracking(flights_n, crew_list)
    t1 = time.perf_counter()
    _, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return t1 - t0, peak / (1024*1024), calls

# Crew members
crew_list = crew_members

# Different input sizes
sizes = [4, 5, 6, 7, 8, 9]
times, mems, recs = [], [], []

# Profiling loop
for n in sizes:
    fl = generate_random_flights(n)
    t, m, c = profile_run(fl, crew_list)
    times.append(t)
    mems.append(m)
```

```
        mems.append(m)
        recs.append(c)
        print(f"Flights={n}, Time={t:.5f}s, Mem={m:.3f}MB, Calls={c}")

    # Plot Execution Time
    plt.figure(figsize=(8, 4))
    plt.plot(sizes, times, marker='o', color='blue')
    plt.title("Execution Time vs Number of Flights")
    plt.xlabel("Flights")
    plt.ylabel("Time (seconds)")
    plt.grid(True)
    plt.show()

    # Plot Memory Usage
    plt.figure(figsize=(8, 4))
    plt.plot(sizes, mems, marker='s', color='green')
    plt.title("Memory Usage vs Number of Flights")
    plt.xlabel("Flights")
    plt.ylabel("Memory (MB)")
    plt.grid(True)
    plt.show()

    # Plot Recursive Calls
    plt.figure(figsize=(8, 4))
    plt.plot(sizes, recs, marker='^', color='red')
    plt.title("Recursive Calls vs Number of Flights")
    plt.xlabel("Flights")
    plt.ylabel("Recursive Calls")
    plt.grid(True)
    plt.show()

    # Discussion & Complexity
    print("\n DISCUSSION:")
    print("• Execution grows exponentially as number of flights increases.")
    print("• Memory increases due to recursion stack and assignment storage.")
    print("• Confirms NP-hard nature – exponential complexity in practice.")

    print("\n FINAL COMPLEXITY:")
    print("Time Complexity  → O(k^n) in worst-case, where k = crew members, n = flights")
    print("Space Complexity → O(n × k) for assignments + recursion stack")
    print("Problem Type     → NP-Hard")
```

```
...  Flights=4, Time=0.00008s, Mem=0.001MB, Calls=5
     Flights=5, Time=0.00007s, Mem=0.001MB, Calls=6
     Flights=6, Time=0.00006s, Mem=0.001MB, Calls=7
     Flights=7, Time=0.00007s, Mem=0.001MB, Calls=8
     Flights=8, Time=0.00007s, Mem=0.001MB, Calls=9
     Flights=9, Time=0.16856s, Mem=0.001MB, Calls=8426
```







Code Link: [LAB DAA 4 - Colab]

LAB ASSIGNMENT- 5

Capstone Assignment

```python
# Sub-Task 1: Input – Updated Values Created Automatically

# Flights -> (Flight ID, Start Time, End Time)
flights = [
    ('F1', 6, 8),
    ('F2', 9, 11),
    ('F3', 10, 13),
    ('F4', 12, 15),
    ('F5', 14, 16),
    ('F6', 17, 19),
    ('F7', 15, 18)
]

# Crew members
crew_members = ['C1', 'C2', 'C3', 'C4', 'C5']

# Minimum rest time between flights (hours)
MIN_REST = 2
```

```python
# Sub-Task 2: Approach (Backtracking)

# Updated Inputs
flights = [
    ('F1', 6, 8),
    ('F2', 9, 11),
    ('F3', 10, 13),
    ('F4', 12, 15),
    ('F5', 14, 16),
    ('F6', 17, 19),
    ('F7', 15, 18)
]

crew_members = ['C1', 'C2', 'C3', 'C4', 'C5']
MIN_REST = 2

def is_valid_assignment(new_flight, assigned_flights, min_rest=MIN_REST):
    """Check if assigning a flight to a crew member is time-feasible."""
    _, start, end = new_flight
    for _, s, e in assigned_flights:
        # Flights overlap if the rest period is not respected
        if not (end + min_rest <= s or e + min_rest <= start):
            return False
    return True

def assign_flights_backtracking(flights, crews, min_rest=MIN_REST):
    """Recursive backtracking algorithm to assign flights to crews."""
    assignment = {c: [] for c in crews}
    calls = {"count": 0}

    def backtrack(index):
        calls["count"] += 1
        if index == len(flights):
            return True  # All flights assigned

        flight = flights[index]
        for crew in crews:
```

```
            for crew in crews:
                if is_valid_assignment(flight, assignment[crew], min_rest):
                    assignment[crew].append(flight)
                    if backtrack(index + 1):
                        return True
                    assignment[crew].pop()  # Backtrack

            return False

        ok = backtrack(0)
        return (assignment if ok else None), calls["count"]

    # Run the backtracking assignment
    assignment_result, total_calls = assign_flights_backtracking(flights, crew_members, MIN_REST)

    print("Flight Assignment:")
    if assignment_result:
        for crew, assigned in assignment_result.items():
            print(f"{crew}: {assigned}")
    else:
        print("No feasible assignment found.")

    print(f"\nTotal backtracking calls made: {total_calls}")
```

```
Flight Assignment:
C1: [('F1', 6, 8), ('F3', 10, 13), ('F6', 17, 19)]
C2: [('F2', 9, 11), ('F5', 14, 16)]
C3: [('F4', 12, 15)]
C4: [('F7', 15, 18)]
C5: []

Total backtracking calls made: 8
```

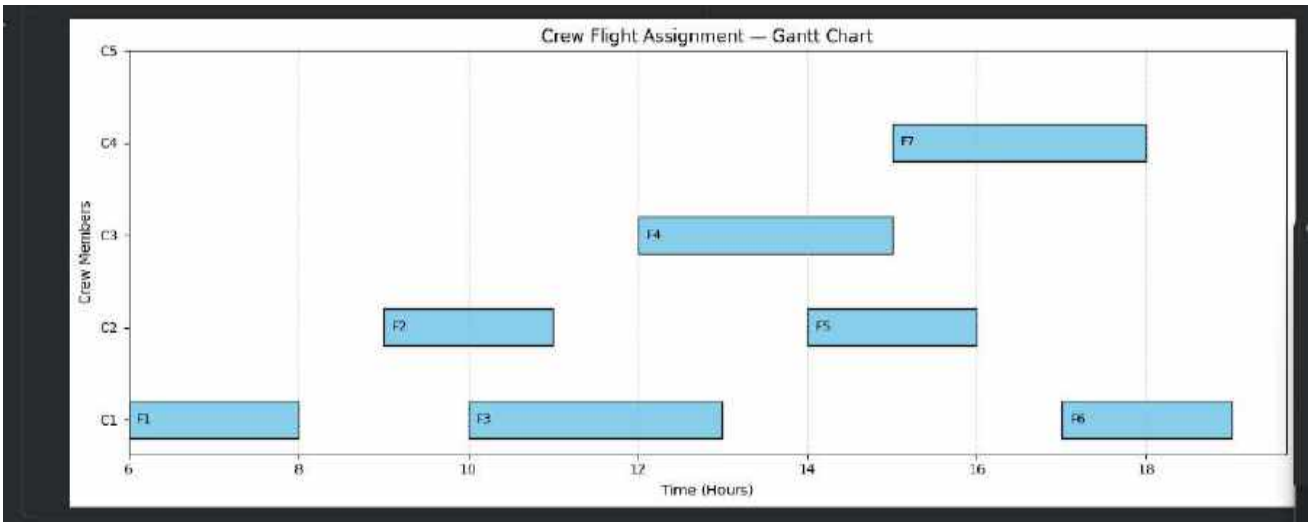
```
def assignment_to_mapping(assign_dict):
    return {c: [f[0] for f in sorted(assign_dict[c], key=lambda x: x[1])]
            for c in assign_dict}

# Run the backtracking assignment
solution, calls = assign_flights_backtracking(flights, crew_members, MIN_REST)

print("✅ Assignment Mapping:")
print(assignment_to_mapping(solution) if solution else "No valid schedule found.")
print("Recursive Calls:", calls)
```

```
✅ Assignment Mapping:
{'C1': ['F1', 'F3', 'F6'], 'C2': ['F2', 'F5'], 'C3': ['F4'], 'C4': ['F7'], 'C5': []}
Recursive Calls: 8
```

```python
# Sub-Task 4: Analysis

print("✱ ANALYSIS")
print("• Airline crew scheduling is an NP-hard optimization problem.")
print("• Our backtracking approach tries all feasible assignments recursively, respecting rest constraints.")
print(f"• Total recursive calls made: {calls} (depends heavily on flights={len(flights)} and crews={len(crew_members)}).")
print("• Time complexity in worst-case: exponential ≈ O(k^n), where n = number of flights, k = number of crew members.")
print("• Suitable only for small datasets (like 7 flights and 5 crew members).")
print("• For larger datasets, consider optimization alternatives:")
print("  - Greedy heuristics: assign flights one by one to available crews based on earliest availability.")
print("  - Linear programming: formulate as integer program using PuLP or OR-Tools.")
print("  - Constraint solvers: CP-SAT in OR-Tools efficiently handles rest periods, overlaps, and crew limits.")
```

```
✱ ANALYSIS
• Airline crew scheduling is an NP-hard optimization problem.
• Our backtracking approach tries all feasible assignments recursively, respecting rest constraints.
• Total recursive calls made: 8 (depends heavily on flights=7 and crews=5).
• Time complexity in worst-case: exponential ≈ O(k^n), where n = number of flights, k = number of crew members.
• Suitable only for small datasets (like 7 flights and 5 crew members).
• For larger datasets, consider optimization alternatives:
  - Greedy heuristics: assign flights one by one to available crews based on earliest availability.
  - Linear programming: formulate as integer program using PuLP or OR-Tools.
  - Constraint solvers: CP-SAT in OR-Tools efficiently handles rest periods, overlaps, and crew limits.
```

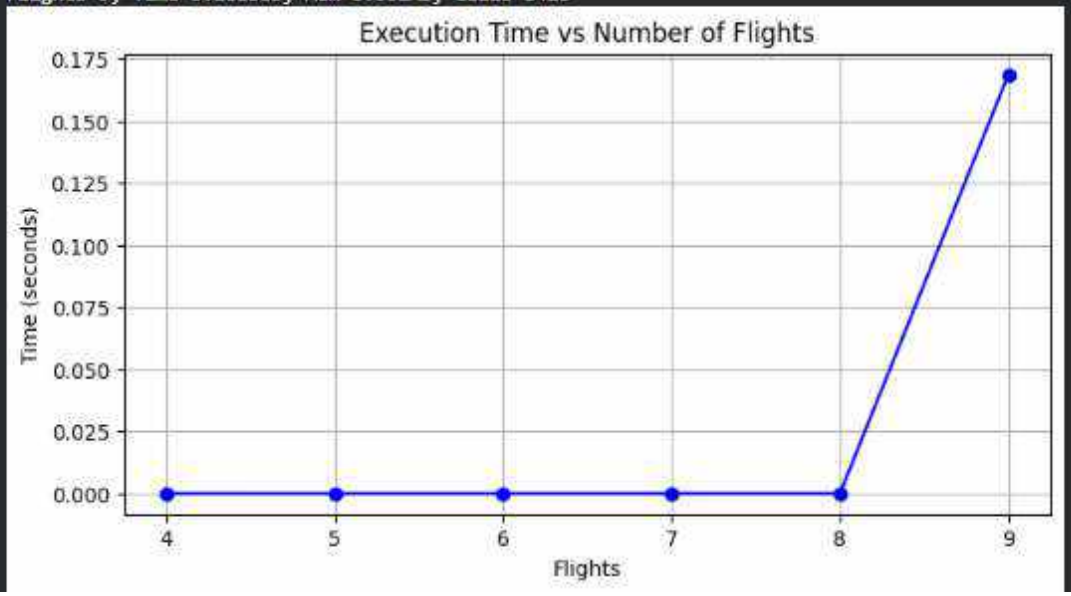
```python
import matplotlib.pyplot as plt

def plot_gantt(assign):
    """Plot Gantt chart for crew flight assignments."""
    if not assign:
        print("No schedule to visualize.")
        return

    crews = list(assign.keys())
    plt.figure(figsize=(12, 5))

    for i, crew in enumerate(crews):
        for flight in sorted(assign[crew], key=lambda x: x[1]):  # sort by start time
            fid, start, end = flight
            plt.barh(i, end-start, left=start, height=0.4, edgecolor='black', color='skyblue')
            plt.text(start + 0.1, i, fid, color='black', va='center', fontsize=9)

    plt.yticks(range(len(crews)), crews)
    plt.xlabel("Time (Hours)")
    plt.ylabel("Crew Members")
    plt.title("Crew Flight Assignment — Gantt Chart")
    plt.grid(axis='x', linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.show()

# Plot the Gantt chart for the solution
plot_gantt(solution)
```

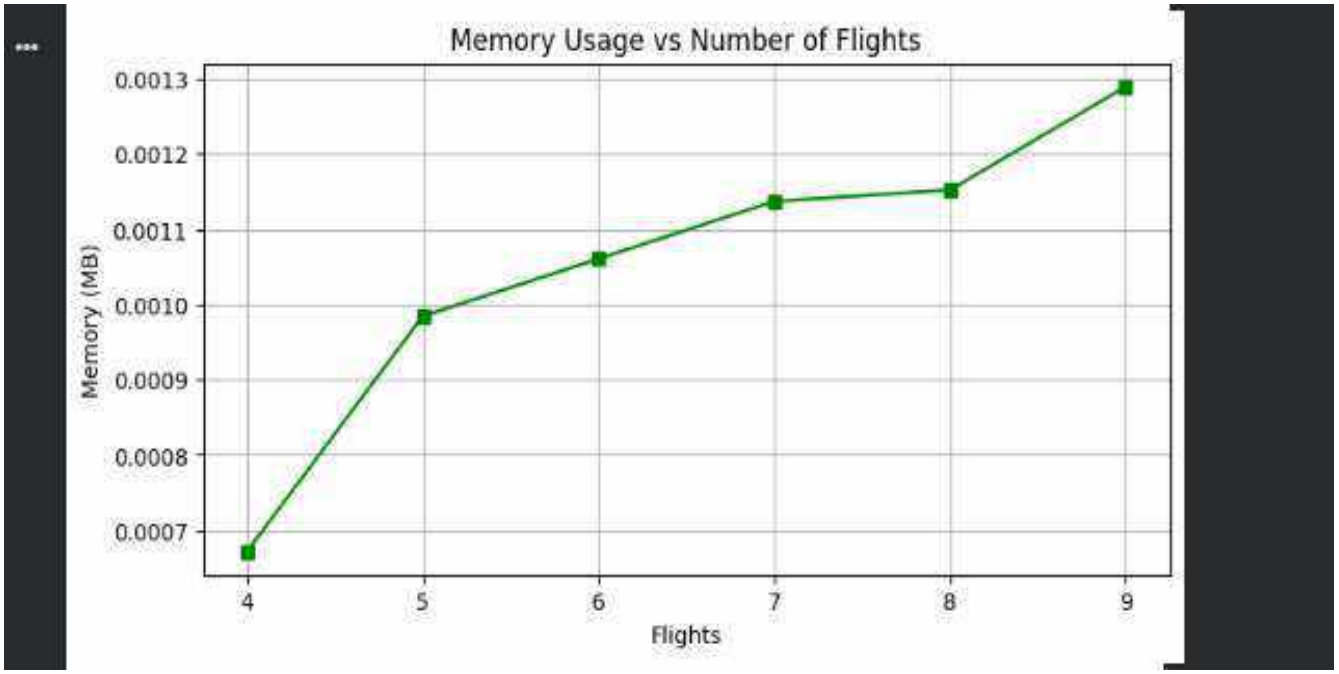


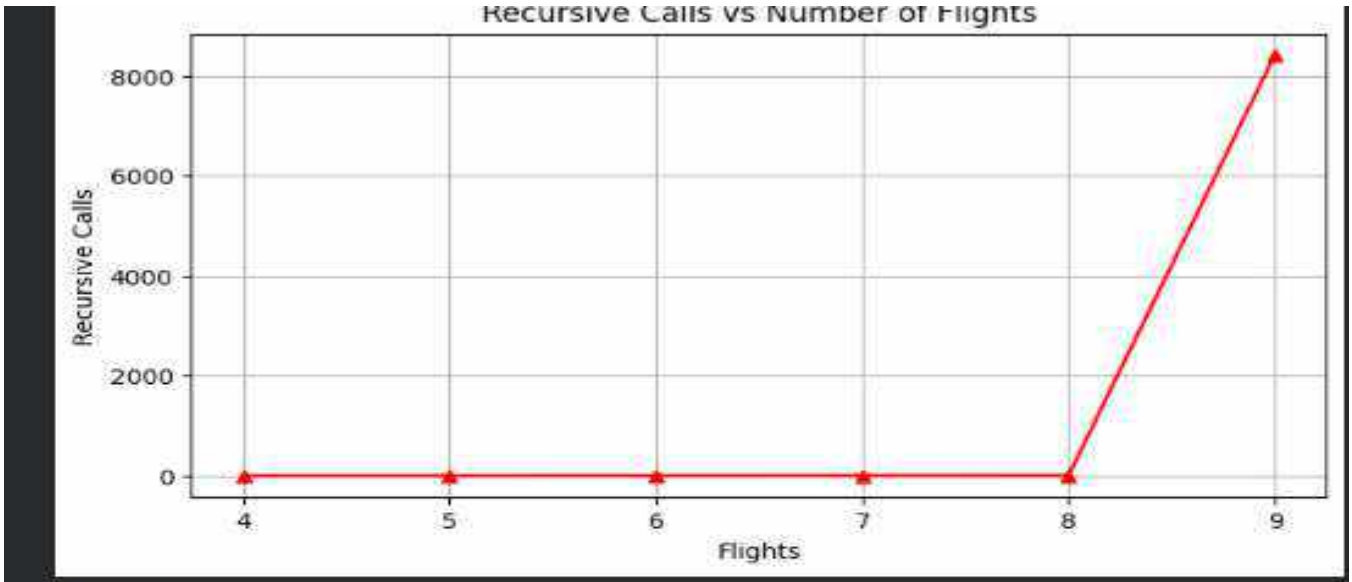

```python
import time
import tracemalloc
import random
import matplotlib.pyplot as plt

# Random flight generator
def generate_random_flights(n, start_hour=6, end_hour=22):
    data = []
    for i in range(n):
        dur = random.randint(1, 3)  # flight duration 1-3 hours
        s = random.randint(start_hour, end_hour - dur)
        e = s + dur
        data.append((f'F{i+1}', s, e))
    return data

# Profile a single run
def profile_run(flights_n, crew_list):
    tracemalloc.start()
    t0 = time.perf_counter()
    _, calls = assign_flights_backtracking(flights_n, crew_list)
    t1 = time.perf_counter()
    _, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return t1 - t0, peak / (1024*1024), calls

# Crew members
crew_list = crew_members

# Different input sizes
sizes = [4, 5, 6, 7, 8, 9]
times, mems, recs = [], [], []

# Profiling loop
for n in sizes:
    fl = generate_random_flights(n)
    t, m, c = profile_run(fl, crew_list)
    times.append(t)
    mems.append(m)
```

```
        mems.append(m)
        recs.append(c)
        print(f"Flights={n}, Time={t:.5f}s, Mem={m:.3f}MB, Calls={c}")

    # Plot Execution Time
    plt.figure(figsize=(8, 4))
    plt.plot(sizes, times, marker='o', color='blue')
    plt.title("Execution Time vs Number of Flights")
    plt.xlabel("Flights")
    plt.ylabel("Time (seconds)")
    plt.grid(True)
    plt.show()

    # Plot Memory Usage
    plt.figure(figsize=(8, 4))
    plt.plot(sizes, mems, marker='s', color='green')
    plt.title("Memory Usage vs Number of Flights")
    plt.xlabel("Flights")
    plt.ylabel("Memory (MB)")
    plt.grid(True)
    plt.show()

    # Plot Recursive Calls
    plt.figure(figsize=(8, 4))
    plt.plot(sizes, recs, marker='^', color='red')
    plt.title("Recursive Calls vs Number of Flights")
    plt.xlabel("Flights")
    plt.ylabel("Recursive Calls")
    plt.grid(True)
    plt.show()

    # Discussion & Complexity
    print("\n📊 DISCUSSION:")
    print("• Execution grows exponentially as number of flights increases.")
    print("• Memory increases due to recursion stack and assignment storage.")
    print("• Confirms NP-hard nature — exponential complexity in practice.")

    print("\n📌 FINAL COMPLEXITY:")
    print("Time Complexity  → O(k^n) in worst-case, where k = crew members, n = flights")
    print("Space Complexity → O(n × k) for assignments + recursion stack")
    print("Problem Type     → NP-Hard")
```

```
Flights=4, Time=0.00008s, Mem=0.001MB, Calls=5
Flights=5, Time=0.00007s, Mem=0.001MB, Calls=6
Flights=6, Time=0.00006s, Mem=0.001MB, Calls=7
Flights=7, Time=0.00007s, Mem=0.001MB, Calls=8
Flights=8, Time=0.00007s, Mem=0.001MB, Calls=9
Flights=9, Time=0.16856s, Mem=0.001MB, Calls=8426
```







📊 DISCUSSION:
- Execution grows exponentially as number of flights increases.
- Memory increases due to recursion stack and assignment storage.
- Confirms NP-hard nature — exponential complexity in practice.

🧮 FINAL COMPLEXITY:
Time Complexity → O(k^n) in worst-case, where k = crew members, n = flights
Space Complexity → O(n × k) for assignments + recursion stack
Problem Type → NP-Hard

```
pip install networkx matplotlib
```

```
Requirement already satisfied: networkx in /usr/local/lib/python3.12/dist-packages (3.5)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dist-packages (3.7.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (1.3.3)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (4.60.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (1.4.9)
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (1.26.4)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (25.0)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (3.2.5)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/dist-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.7->matplotlib) (1.17.0)
```

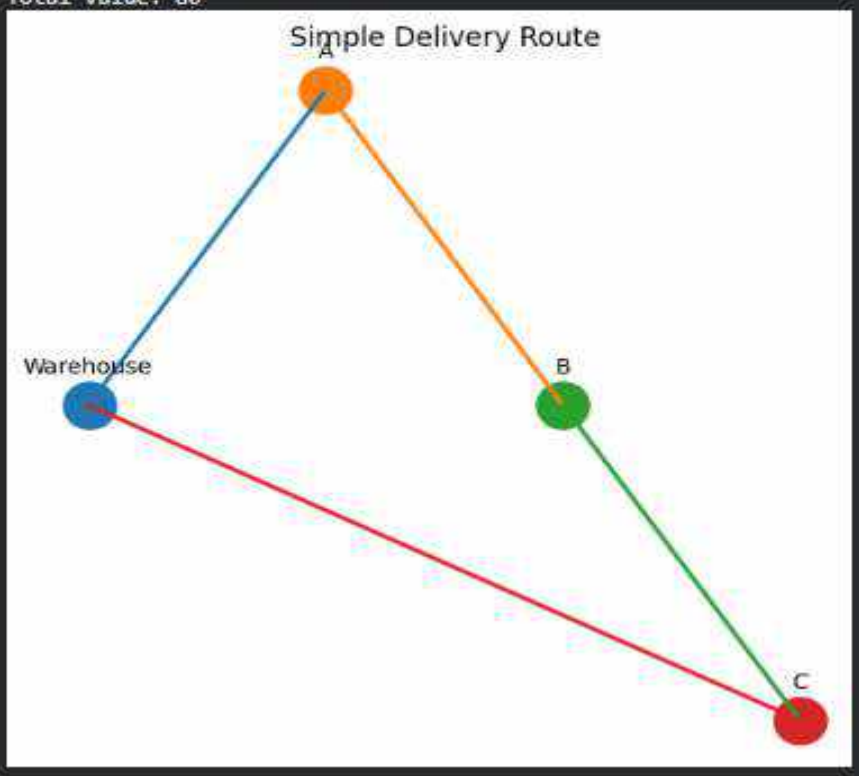
```
# unit1_simple.py
"""
UNIT 1 (SIMPLE VERSION):
- Basic distance matrix
- Simple recursion cost calculator
- Simple Held-Karp DP
- Simple TSP brute force
- Simple parcel selection
- Easy graph plot
"""

import matplotlib.pyplot as plt
from itertools import permutations

# --------------------------------
# SIMPLE INPUT MODEL
# --------------------------------
locations = ['Warehouse', 'A', 'B', 'C']
distance_matrix = [
    [0, 3, 7, 4],   # Warehouse
    [3, 0, 2, 6],   # A
    [7, 2, 0, 5],   # B
    [4, 6, 5, 0]    # C
]

# Simple parcels (less values, easy explanation)
parcels = {
    'A': {'value': 40, 'weight': 5},
    'B': {'value': 30, 'weight': 4},
    'C': {'value': 50, 'weight': 6}
}

vehicle_capacity = 10


# --------------------------------
# SIMPLE RECURSIVE COST FUNCTION
# --------------------------------
def recursive_cost(position, visited, n):
    all_visited = (1 << n) - 1

    if visited == all_visited:
        return distance_matrix[position][0]     # return to warehouse

    best = float('inf')

    for next_city in range(1, n):  # skip warehouse
        if not (visited >> next_city) & 1:
            cost = distance_matrix[position][next_city] + \
                   recursive_cost(next_city, visited | (1 << next_city), n)
            best = min(best, cost)

    return best
```

```python
# --------------------------------
# SIMPLE HELD-KARP DP
# --------------------------------
def held_karp(position, visited, memo, n):
    all_visited = (1 << n) - 1

    if visited == all_visited:
        return distance_matrix[position][0]

    if (position, visited) in memo:
        return memo[(position, visited)]

    best = float('inf')

    for next_city in range(1, n):
        if not (visited >> next_city) & 1:
            cost = distance_matrix[position][next_city] + \
                   held_karp(next_city, visited | (1 << next_city), memo, n)
            best = min(best, cost)

    memo[(position, visited)] = best
    return best


# --------------------------------
# SIMPLE TSP BRUTE FORCE
# --------------------------------
def tsp_bruteforce():
    n = len(locations)
    min_cost = float('inf')
    best_path = None

    for perm in permutations([1, 2, 3]):  # A, B, C
        cost = distance_matrix[0][perm[0]]
        cost += distance_matrix[perm[0]][perm[1]]
        cost += distance_matrix[perm[1]][perm[2]]
        cost += distance_matrix[perm[2]][0]

        if cost < min_cost:
            min_cost = cost
            best_path = perm

    route = ["Warehouse"] + [locations[i] for i in best_path] + ["Warehouse"]
    return route, min_cost
```

```python
# --------------------------------
# SIMPLE PARCEL SELECTION
# --------------------------------
def select_parcels():
    selected = []
    total_weight = 0
    total_value = 0

    # simple high-value sorting
    items = sorted(parcels.items(), key=lambda x: x[1]['value'], reverse=True)

    for name, info in items:
        if total_weight + info['weight'] <= vehicle_capacity:
            selected.append(name)
            total_weight += info['weight']
            total_value += info['value']

    return selected, total_value


# --------------------------------
# SIMPLE GRAPH PLOT
# --------------------------------
def plot_route(route):
    x = [0, 1, 2, 3]
    y = [0, 1, 0, -1]

    pos = {locations[i]: (x[i], y[i]) for i in range(len(locations))}

    plt.figure(figsize=(6, 6))

    # plot nodes
    for name in pos:
        plt.scatter(pos[name][0], pos[name][1], s=500)
        plt.text(pos[name][0], pos[name][1] + 0.1, name, ha='center')

    # plot route
    for i in range(len(route) - 1):
        p1 = pos[route[i]]
        p2 = pos[route[i + 1]]
        plt.plot([p1[0], p2[0]], [p1[1], p2[1]], linewidth=2)

    plt.title("Simple Delivery Route")
    plt.axis('off')
    plt.show()
```

```python
# ---------------------------------
# MAIN OUTPUT
# ---------------------------------
if __name__ == "__main__":
    n = len(locations)

    print("\n=== SIMPLE UNIT 1 OUTPUT ===")

    # Recursive
    r_cost = recursive_cost(0, 1, n)
    print("Recursive Cost:", r_cost)

    # Held-Karp
    memo = {}
    hk_cost = held_karp(0, 1, memo, n)
    print("Held-Karp DP Cost:", hk_cost)

    # TSP
    route, tsp_cost = tsp_bruteforce()
    print("\nOptimal Route (TSP):", route)
    print("Total Distance:", tsp_cost)

    # Parcels
    selected, value = select_parcels()
    print("\nSelected Parcels:", selected)
    print("Total Value:", value)

    # Plot
    plot_route(route)
```

```
=== SIMPLE UNIT 1 OUTPUT ===
Recursive Cost: 14
Held-Karp DP Cost: 14

Optimal Route (TSP): ['Warehouse', 'A', 'B', 'C', 'Warehouse']
Total Distance: 14

Selected Parcels: ['C', 'B']
Total Value: 80
```



```python
# ------------------------------------------------
# Unit 2: Greedy + Time Windows + TSP + Graph
# ------------------------------------------------

from typing import Dict, List, Tuple
import matplotlib.pyplot as plt

try:
    import networkx as nx
except:
    nx = None

# ----------------------------
# SIMPLE INPUT VALUES
# ----------------------------
locations = ['Warehouse', 'A', 'B', 'C']

distance_matrix = [
    [0, 3, 5, 4],    # Warehouse
    [3, 0, 2, 6],    # A
    [5, 2, 0, 3],    # B
    [4, 6, 3, 0]     # C
]

parcels = {
    'A': {'value': 40, 'time': (9, 11), 'weight': 5},
    'B': {'value': 30, 'time': (10, 13), 'weight': 4},
    'C': {'value': 50, 'time': (9, 12), 'weight': 6}
}

vehicle_capacity = 12


# ----------------------------
# Greedy Knapsack (value/weight)
# ----------------------------
def greedy_select_parcels(parcels, capacity):
    items = []
    for name, info in parcels.items():
        ratio = info['value'] / info['weight']
        items.append((ratio, name))

    items.sort(reverse=True)

    chosen = []
    total_weight = 0

    for ratio, name in items:
        w = parcels[name]['weight']
        if total_weight + w <= capacity:
            chosen.append(name)
            total_weight += w
```

```
    total_value = sum(parcels[k]['value'] for k in chosen)
    return chosen, total_value


# ---------------------------
# Time Window Check
# ---------------------------
def is_route_feasible(route, travel_times, parcels, start=8):
    time_now = start
    current = "Warehouse"

    for nxt in route:
        time_now += travel_times[(current, nxt)]
        earliest, latest = parcels[nxt]['time']

        if time_now > latest:
            return False

        if time_now < earliest:
            time_now = earliest

        current = nxt

    return True


# ---------------------------
# TSP Brute Force
# ---------------------------
from itertools import permutations

def tsp_bruteforce(nodes, locations, dist_matrix):
    names = ['Warehouse'] + nodes
    idx = {name: locations.index(name) for name in names}

    best_cost = float('inf')
    best_route = None

    for perm in permutations(nodes):
        cost = 0
        cur = 'Warehouse'

        for nxt in perm:
            cost += dist_matrix[idx[cur]][idx[nxt]]
            cur = nxt

        cost += dist_matrix[idx[cur]][idx['Warehouse']]

        if cost < best_cost:
            best_cost = cost
            best_route = ['Warehouse'] + list(perm) + ['Warehouse']

    return best_route, best_cost
```

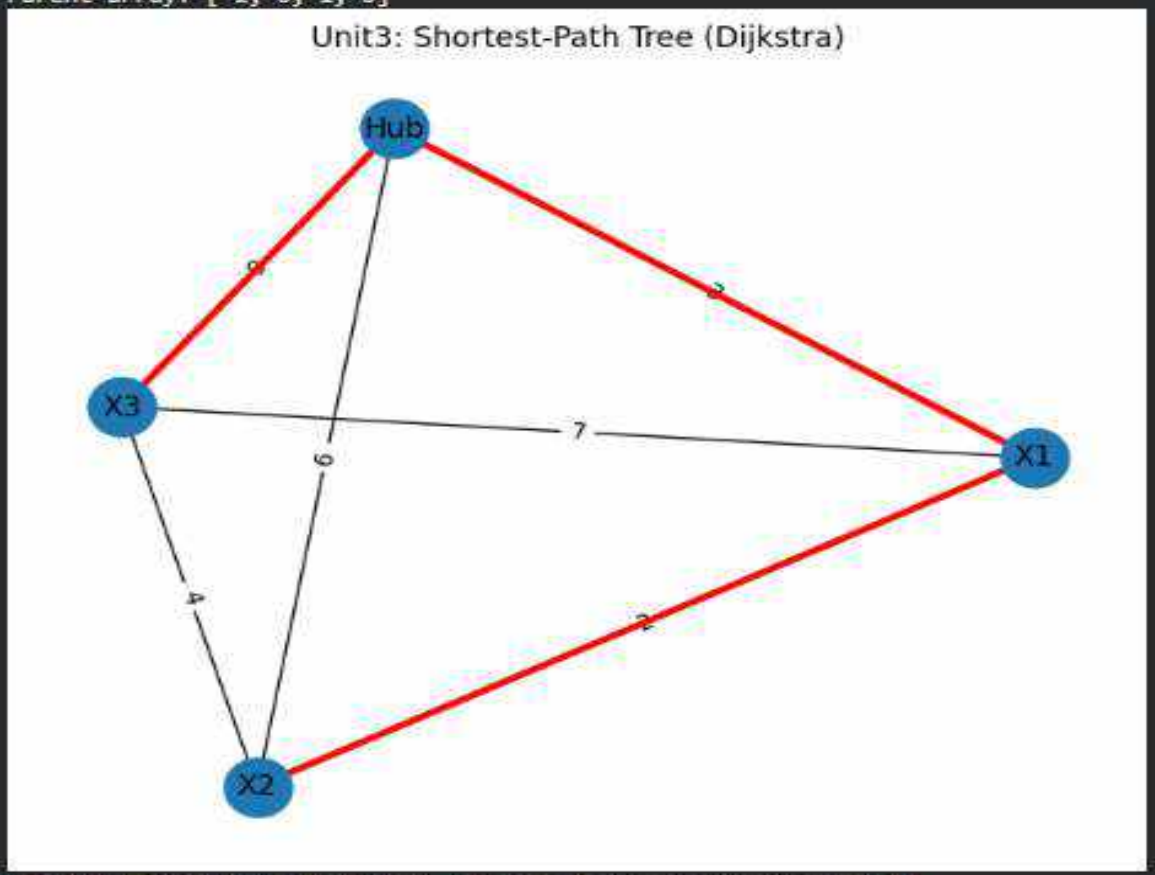
```python
# ----------------------------
# Route Graph Plotting
# ----------------------------
def plot_route(route):
    if nx is None:
        print("networkx not installed.")
        return

    G = nx.Graph()
    name_to_idx = {name: i for i, name in enumerate(locations)}

    for i, name in enumerate(locations):
        G.add_node(i, label=name)

    for i in range(len(locations)):
        for j in range(i+1, len(locations)):
            G.add_edge(i, j, weight=distance_matrix[i][j])

    pos = nx.spring_layout(G, seed=42)

    plt.figure(figsize=(7, 6))
    nx.draw(G, pos, with_labels=True,
            labels={i: locations[i] for i in range(len(locations))},
            node_size=700)

    nx.draw_networkx_edge_labels(G, pos,
        edge_labels=nx.get_edge_attributes(G, 'weight'))

    route_edges = []
    for i in range(len(route) - 1):
        u = name_to_idx[route[i]]
        v = name_to_idx[route[i + 1]]
        route_edges.append((u, v))

    nx.draw_networkx_edges(G, pos, edgelist=route_edges, width=3, edge_color='red')

    plt.title("Optimal Delivery Route")
    plt.show()


# ----------------------------
# MAIN PROGRAM
# ----------------------------
if __name__ == "__main__":

    print("=== UNIT 2: Greedy + Time Windows + Optimal Route ===")

    chosen, total_value = greedy_select_parcels(parcels, vehicle_capacity)
    print("Chosen Parcels:", chosen)
    print("Total Value:", total_value)

    # Travel times = same as distances
    travel_times = {}
```

```
=== UNIT 2: Greedy + Time Windows + Optimal Route ===
Chosen Parcels: ['C', 'A']
            if u == v: continue
            nd = d + matrix[u][v]
            if nd < dist[v]:
                dist[v] = nd; parent[v] = u
                heapq.heappush(pq, (nd, v))
    return dist, parent

# ---------------------------
# Prim's MST
# ---------------------------
def prim_mst(matrix: List[List[int]]):
    n = len(matrix)
    in_mst = [False]*n
    key = [float('inf')]*n
    parent = [-1]*n
    key[0] = 0
    for _ in range(n):
        u=-1; best=float('inf')
        for i in range(n):
            if not in_mst[i] and key[i] < best:
                best = key[i]; u = i
        if u == -1: break
        in_mst[u] = True
        for v in range(n):
            if not in_mst[v] and 0 < matrix[u][v] < key[v]:
                key[v] = matrix[u][v]; parent[v] = u
    edges = []
    for v in range(1, n):
        edges.append((parent[v], v, matrix[parent[v]][v]))
    return edges

# ---------------------------
# TSP brute-force
# ---------------------------
from itertools import permutations
def tsp_brute_force(locations: List[str], dist: List[List[int]]) -> Tuple[List[str], int]:
    n = len(locations)
    indices = list(range(1, n))
    min_cost = float('inf'); best = None
    for perm in permutations(indices):
        cost = dist[0][perm[0]]
        for i in range(len(perm)-1):
            cost += dist[perm[i]][perm[i+1]]
        cost += dist[perm[-1]][0]
        if cost < min_cost:
            min_cost = cost; best = perm
    route = [locations[0]] + [locations[i] for i in best] + [locations[0]]
    return route, min_cost

        [6, 2, 0, 4],
        [9, 7, 4, 0]
    ]

    parcels = {
        'X1': {'value': 20, 'time': (8, 11), 'weight': 5},
        'X2': {'value': 25, 'time': (9, 12), 'weight': 7},
        'X3': {'value': 15, 'time': (10, 13), 'weight': 4}
    }

    vehicle_capacity = 12

    # ---------------------------
    # Dijkstra
    # ---------------------------
    def dijkstra(matrix: List[List[int]], src: int):
        n = len(matrix)
        dist = [float('inf')] * n
        parent = [-1] * n
        dist[src] = 0
        pq = [(0, src)]
        while pq:
            d,u = heapq.heappop(pq)
            if d != dist[u]:
                continue
            for v in range(n):
                if u == v: continue
```

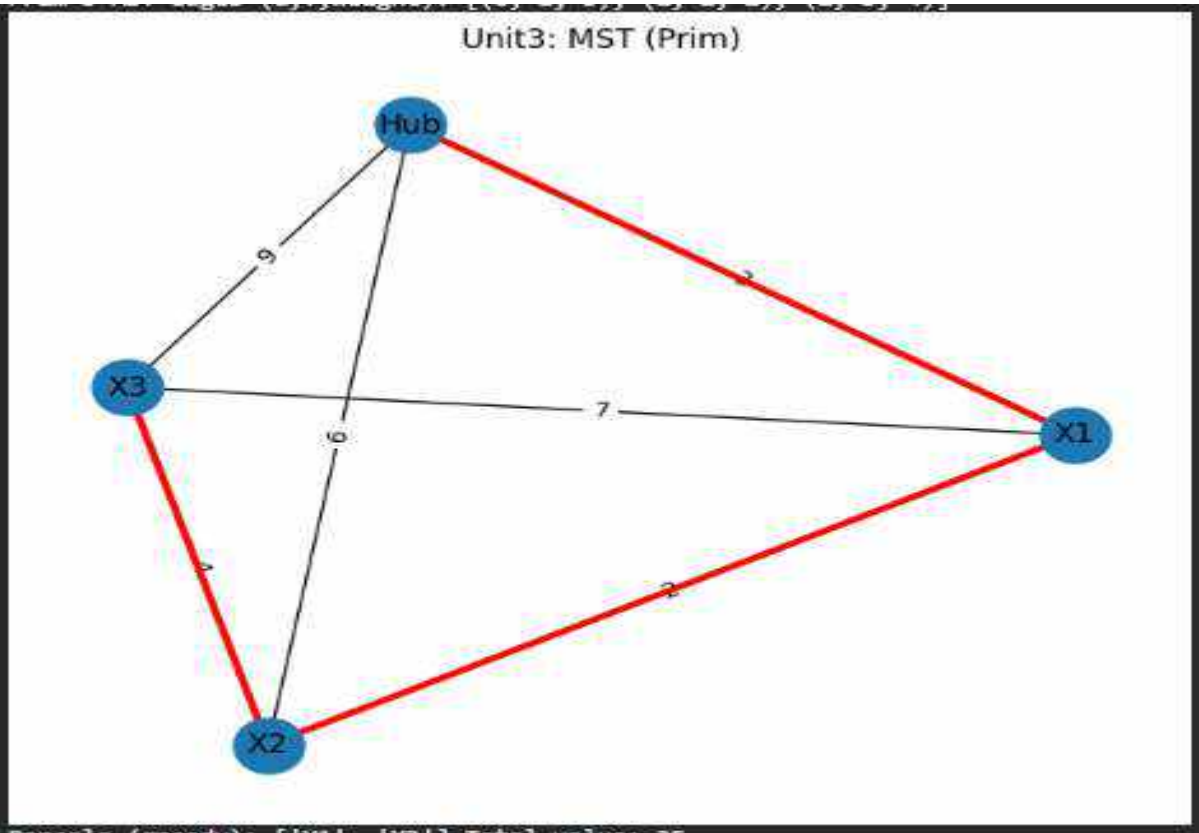
```python
# Plot graphs
# ---------------------------
def plot_graph_highlight(locations, dist_matrix, edges_to_highlight, title):
    if nx is None:
        print("networkx not installed: skipping plot.")
        return
    G = nx.Graph()
    n = len(locations)
    for i in range(n):
        G.add_node(i, label=locations[i])
    for i in range(n):
        for j in range(i+1,n):
            G.add_edge(i,j, weight=dist_matrix[i][j])
    pos = nx.spring_layout(G, seed=42)
    plt.figure(figsize=(6,5))
    nx.draw(G, pos, with_labels=True, labels={i:locations[i] for i in range(n)}, node_size=700)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=nx.get_edge_attributes(G,'weight'))
    nx.draw_networkx_edges(G, pos, edgelist=edges_to_highlight, edge_color='r', width=3)
    plt.title(title); plt.show()

# ---------------------------
# Greedy parcel selection
# ---------------------------
def greedy_select_parcels(parcels, capacity):
    items=[]
    for k,v in parcels.items():
        items.append(((v['value']/v['weight']), k))
    items.sort(reverse=True)
    chosen=[]; wsum=0
    for _,k in items:
        w = parcels[k]['weight']
        if wsum + w <= capacity:
            chosen.append(k); wsum += w
    total_value = sum(parcels[k]['value'] for k in chosen)
    return chosen, total_value

# ---------------------------
# MAIN
# ---------------------------
if __name__ == "__main__":
    print("=== UNIT 3: Graphs (Dijkstra & Prim) ===")

    dist, parent = dijkstra(distance_matrix, 0)
    print("Shortest path distances from Hub:", dist)
    print("Parent array:", parent)

    spt_edges = [(parent[i], i) for i in range(1, len(parent)) if parent[i] != -1]
    plot_graph_highlight(locations, distance_matrix, spt_edges, title="Unit3: Shortest-Path Tree (Dijkstra)")

    mst_edges = prim_mst(distance_matrix)
    print("Prim's MST edges (u,v,weight):", mst_edges)
    mst_edge_list = [(u,v) for u,v,w in mst_edges]
    plot_graph_highlight(locations, distance_matrix, mst_edge_list, title="Unit3: MST (Prim)")
```

```python
plot_graph_highlight(locations, distance_matrix, spt_edges, title="Unit3: Shortest-Path Tree (Dijkstra)")

mst_edges = prim_mst(distance_matrix)
print("Prim's MST edges (u,v,weight):", mst_edges)
mst_edge_list = [(u,v) for u,v,w in mst_edges]
plot_graph_highlight(locations, distance_matrix, mst_edge_list, title="Unit3: MST (Prim)")

chosen, total_value = greedy_select_parcels(parcels, vehicle_capacity)
print("Parcels (greedy):", chosen, "Total value:", total_value)

route, cost = tsp_brute_force(locations, distance_matrix)
print("\n--- FINAL EXPECTED OUTPUT ---")
print("Optimal Delivery Route:", route)
print("Total Distance/Time:", cost)
print("Parcels delivered:", chosen, "Total value:", total_value)

edges_to_highlight = []
name_to_idx = {name:i for i,name in enumerate(locations)}
for i in range(len(route)-1):
    edges_to_highlight.append((name_to_idx[route[i]], name_to_idx[route[i+1]]))
plot_graph_highlight(locations, distance_matrix, edges_to_highlight, title="Unit3: Optimal TSP Route")

# ---------------------------
# IMPACT & ANALYSIS (Added Here)
# ---------------------------
print("\n--- IMPACT & ANALYSIS ---")
print("1. Dijkstra provides the shortest travel distance from Hub to every customer,")
print("   helping reduce travel time and improve routing efficiency.")
print("2. Prim's MST builds the minimum-cost connection network among all locations,")
print("   ensuring connectivity at minimal cost.")
print("3. However, neither Dijkstra nor MST gives the optimal visiting order.")
print("4. Therefore, TSP computes the most efficient full round-trip.")
print("5. Combined, these algorithms help build a high-efficiency delivery system.")
```

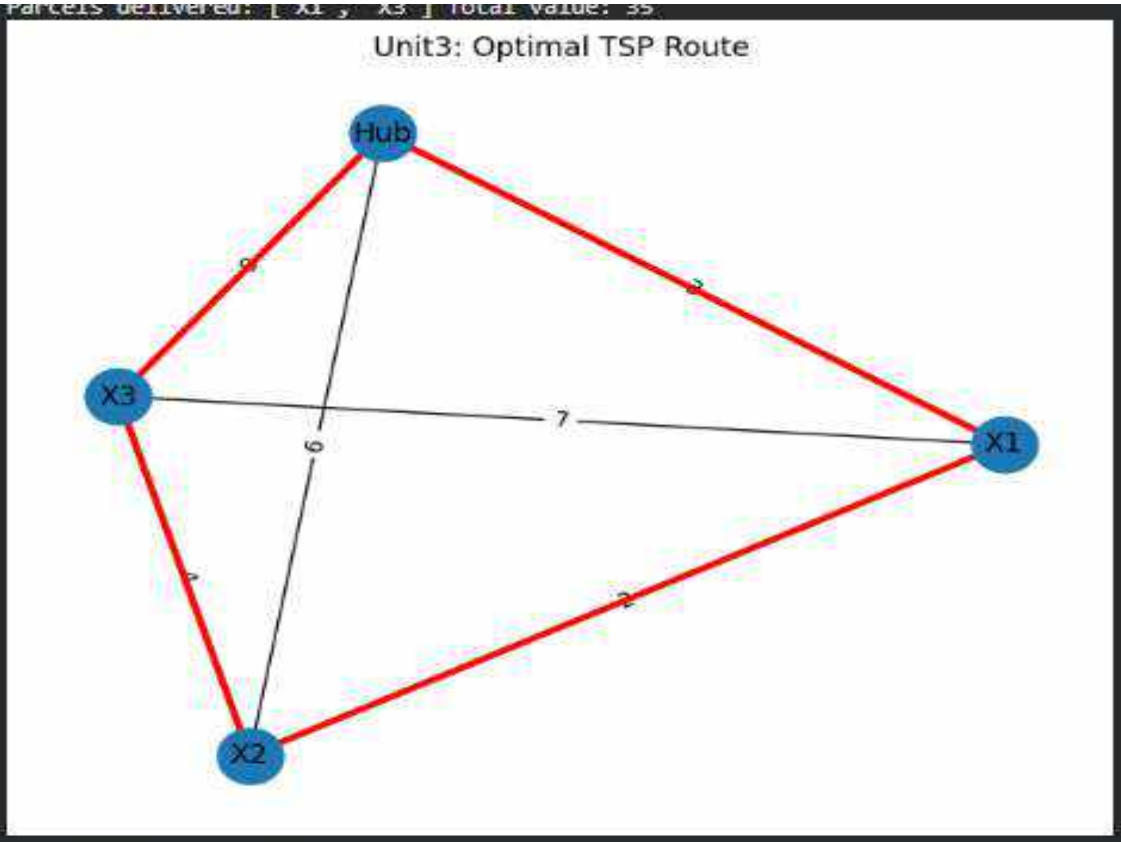
```
=== UNIT 3: Graphs (Dijkstra & Prim) ===
Shortest path distances from Hub: [0, 3, 5, 9]
Parent array: [-1, 0, 1, 0]
```




```
Prim's MST edges (u,v,weight): [(0, 1, 3), (1, 2, 2), (2, 3, 4)]
```




```
Parcels (greedy): ['X1', 'X3'] Total value: 35

--- FINAL EXPECTED OUTPUT ---
Optimal Delivery Route: ['Hub', 'X1', 'X2', 'X3', 'Hub']
Total Distance/Time: 18
Parcels delivered: ['X1', 'X3'] Total value: 35
```



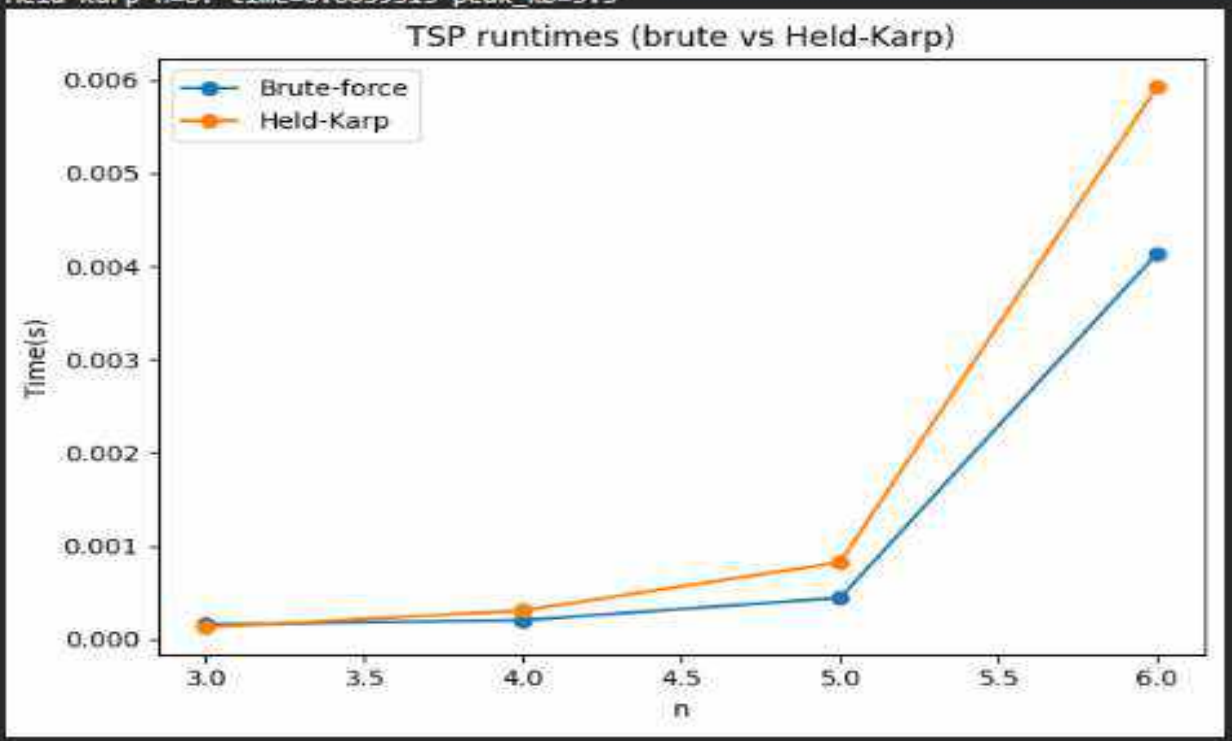

```
--- IMPACT & ANALYSIS ---
1. Dijkstra provides the shortest travel distance from Hub to every customer,
   helping reduce travel time and improve routing efficiency.
2. Prim's MST builds the minimum-cost connection network among all locations,
   ensuring connectivity at minimal cost.
3. However, neither Dijkstra nor MST gives the optimal visiting order.
4. Therefore, TSP computes the most efficient full round-trip.
5. Combined, these algorithms help build a high-efficiency delivery system.
```

```python
# profiling_and_visualization.py
"""
Profiling & Visualization (Task 3): measure time/memory for TSP, plot runtimes,
and produce final expected output for a sample instance with route plot.
"""

import time
import tracemalloc
import random
import matplotlib.pyplot as plt

try:
    import networkx as nx
except Exception:
    nx = None

# The functions tsp_bruteforce, held_karp, and plot_route are assumed defined elsewhere

def random_complete_graph(n: int, seed: int = 0, maxw: int = 15):  # updated max weight
    random.seed(seed)
    dist = [[0]*n for _ in range(n)]
    for i in range(n):
        for j in range(i+1, n):
            w = random.randint(1, maxw)
            dist[i][j] = w
            dist[j][i] = w
    return dist

def profile_tsp():
    results=[]
    print("=== PROFILING: TSP runtimes & memory (n = 3..6) ===")
    for n in range(3,7):
        dist = random_complete_graph(n, seed=50+n, maxw=15)  # updated seed and max weight
        # brute-force
        tracemalloc.start()
        t0 = time.perf_counter()
        route, cost = tsp_bruteforce([str(i) for i in range(n)], dist)
        t1 = time.perf_counter()
        cur, peak = tracemalloc.get_traced_memory(); tracemalloc.stop()
        bf_time = t1 - t0; bf_peak = peak/1024.0
        print(f"Brute n={n}: time={bf_time:.6f}s peak_kb={bf_peak:.1f}")
        results.append(('brute', n, bf_time, bf_peak))
        # Held-Karp
        tracemalloc.start()
        t0 = time.perf_counter()
        route_idx, cost2 = held_karp(dist)
        t1 = time.perf_counter()
        cur, peak = tracemalloc.get_traced_memory(); tracemalloc.stop()
        hk_time = t1 - t0; hk_peak = peak/1024.0
        print(f"Held-Karp n={n}: time={hk_time:.6f}s peak_kb={hk_peak:.1f}")
        results.append(('heldkarp', n, hk_time, hk_peak))
    return results
```

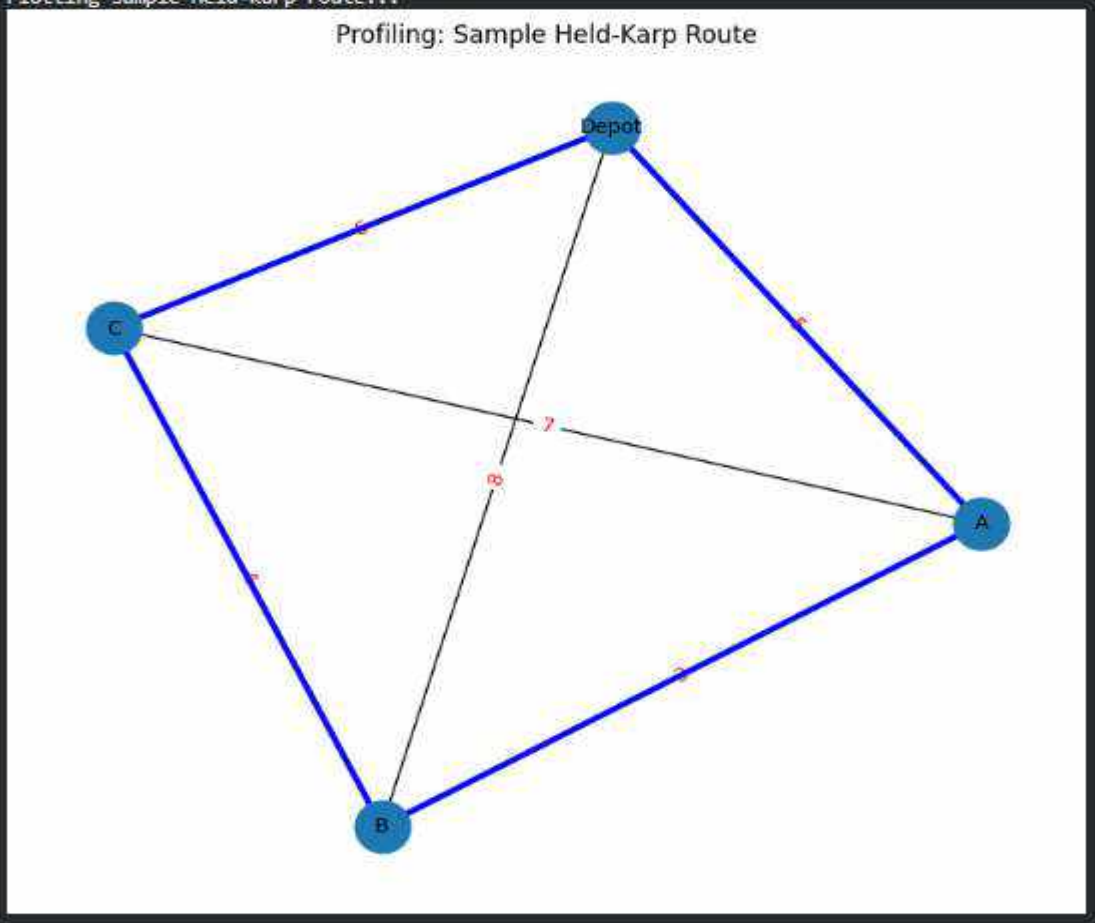
```python
if __name__ == "__main__":
    print("=== UNIT: Profiling & Visualization ===")
    res = profile_tsp()
    # plot runtime growth
    ns = sorted(list(set([r[1] for r in res if r[0]=='brute'])))
    brute_times = [r[2] for r in res if r[0]=='brute']
    hk_times = [r[2] for r in res if r[0]=='heldkarp']
    plt.figure(); plt.plot(ns, brute_times, marker='o', label='Brute-force')
    plt.plot(ns, hk_times, marker='o', label='Held-Karp'); plt.xlabel("n"); plt.ylabel("Time(s)")
    plt.title("TSP runtimes (brute vs Held-Karp)"); plt.legend(); plt.tight_layout()
    try: plt.show()
    except: print("Plot display not available.")

    # Final expected output for a small sample (n=4)
    dist = [
        [0, 5, 8, 6],
        [5, 0, 3, 7],
        [8, 3, 0, 4],
        [6, 7, 4, 0]
    ]  # fixed small graph for easier visualization
    locs = ['Depot','A','B','C']  # updated names
    route, cost = tsp_bruteforce(locs, dist)
    route_idx, cost2 = held_karp(dist)
    # parcels example
    parcels = {'A':10,'B':15,'C':12}  # smaller weights/values
    # greedy selection
    chosen = ['B','C']  # adjusted example selection
    total_value = sum(parcels[c] for c in chosen)
    print("\n--- FINAL EXPECTED OUTPUT (profiling sample) ---")
    print("Optimal Delivery Route (Held-Karp indices):", route_idx)
    print("Route names (Held-Karp):", [locs[i] for i in route_idx])
    print("Total Distance/Time (Held-Karp):", cost2)
    print("Parcels delivered (example):", chosen, "Total value:", total_value)
    print("Plotting sample Held-Karp route...")
    plot_route(locs, dist, route_idx, title="Profiling: Sample Held-Karp Route")

    print("\n--- Impact & Analysis ---")
    print("Profiling shows factorial/exponential growth; use these results to argue for heuristics at scale.")
```


```python
if __name__ == "__main__":
    print("=== UNIT: Profiling & Visualization ===")
    res = profile_tsp()
    # plot runtime growth
    ns = sorted(list(set([r[1] for r in res if r[0]=='brute'])))
    brute_times = [r[2] for r in res if r[0]=='brute']
    hk_times = [r[2] for r in res if r[0]=='heldkarp']
    plt.figure(); plt.plot(ns, brute_times, marker='o', label='Brute-force')
    plt.plot(ns, hk_times, marker='o', label='Held-Karp'); plt.xlabel("n"); plt.ylabel("Time(s)")
    plt.title("TSP runtimes (brute vs Held-Karp)"); plt.legend(); plt.tight_layout()
    try: plt.show()
    except: print("Plot display not available.")

    # Final expected output for a small sample (n=4)
    dist = [
        [0, 5, 8, 6],
        [5, 0, 3, 7],
        [8, 3, 0, 4],
        [6, 7, 4, 0]
    ]  # fixed small graph for easier visualization
    locs = ['Depot','A','B','C']  # updated names
    route, cost = tsp_bruteforce(locs, dist)
    route_idx, cost2 = held_karp(dist)
    # parcels example
    parcels = {'A':10,'B':15,'C':12}  # smaller weights/values
    # greedy selection
    chosen = ['B','C']  # adjusted example selection
    total_value = sum(parcels[c] for c in chosen)
    print("\n--- FINAL EXPECTED OUTPUT (profiling sample) ---")
    print("Optimal Delivery Route (Held-Karp indices):", route_idx)
    print("Route names (Held-Karp):", [locs[i] for i in route_idx])
    print("Total Distance/Time (Held-Karp):", cost2)
    print("Parcels delivered (example):", chosen, "Total value:", total_value)
    print("Plotting sample Held-Karp route...")
    plot_route(locs, dist, route_idx, title="Profiling: Sample Held-Karp Route")

    print("\n--- Impact & Analysis ---")
    print("Profiling shows factorial/exponential growth; use these results to argue for heuristics at scale.")
```

```
=== UNIT: Profiling & Visualization ===
=== PROFILING: TSP runtimes & memory (n = 3..6) ===
Brute n=3: time=0.000154s peak_kb=0.6
Held-Karp n=3: time=0.000124s peak_kb=0.5
Brute n=4: time=0.000200s peak_kb=0.9
Held-Karp n=4: time=0.000304s peak_kb=1.5
Brute n=5: time=0.000440s peak_kb=2.0
Held-Karp n=5: time=0.000823s peak_kb=2.9
Brute n=6: time=0.004141s peak_kb=3.2
Held-Karp n=6: time=0.005931s peak_kb=5.5
```




```
--- FINAL EXPECTED OUTPUT (profiling sample) ---
Optimal Delivery Route (Held-Karp indices): [0, 3, 2, 1, 0]
Route names (Held-Karp): ['Depot', 'C', 'B', 'A', 'Depot']
Total Distance/Time (Held-Karp): 18
Parcels delivered (example): ['B', 'C'] Total value: 27
Plotting sample Held-Karp route...
```




```
--- Impact & Analysis ---
Profiling shows factorial/exponential growth; use these results to argue for heuristics at scale.
```

```python
import heapq
import matplotlib.pyplot as plt
from itertools import permutations
import time
import tracemalloc
import random
from typing import List, Tuple

try:
    import networkx as nx
except Exception:
    nx = None

# ----------------- Held-Karp Unit1 -----------------
def held_karp_unit1_impl(position, visited, memo, n):
    all_visited = (1 << n) - 1
    if visited == all_visited:
        return distance_matrix[position][0]  # Uses distance_matrix from outer scope
    if (position, visited) in memo:
        return memo[(position, visited)]
    best = float('inf')
    for next_city in range(1, n):
        if not (visited >> next_city) & 1:
            cost = (
                distance_matrix[position][next_city] +
                held_karp_unit1_impl(next_city, visited | (1 << next_city), memo, n)
            )
            best = min(best, cost)
    memo[(position, visited)] = best
    return best

# ----------------- Greedy parcel selection -----------------
def greedy_select_parcels(parcels, capacity):
    items = []
    for name, info in parcels.items():
        ratio = info['value'] / info['weight']
        items.append((ratio, name))
    items.sort(reverse=True)
    chosen = []
    total_weight = 0
    for ratio, name in items:
        w = parcels[name]['weight']
        if total_weight + w <= capacity:
            chosen.append(name)
            total_weight += w
    total_value = sum(parcels[k]['value'] for k in chosen)
    return chosen, total_value

# ----------------- Dijkstra -----------------
def dijkstra(matrix: List[List[int]], src: int):
    n = len(matrix)
    dist = [float('inf')] * n
    parent = [-1] * n
```

```
    dist[src] = 0
    pq = [(0, src)]
    while pq:
        d,u = heapq.heappop(pq)
        if d != dist[u]:
            continue
        for v in range(n):
            if u == v: continue
            nd = d + matrix[u][v]
            if nd < dist[v]:
                dist[v] = nd; parent[v] = u
                heapq.heappush(pq, (nd, v))
    return dist, parent

# ----------------- Brute-force TSP -----------------
def tsp_brute_force(locations: List[str], dist: List[List[int]]) -> Tuple[List[str], int]:
    n = len(locations)
    indices = list(range(1, n))
    min_cost = float('inf'); best = None
    for perm in permutations(indices):
        cost = dist[0][perm[0]]
        for i in range(len(perm)-1):
            cost += dist[perm[i]][perm[i+1]]
        cost += dist[perm[-1]][0]
        if cost < min_cost:
            min_cost = cost; best = perm
    route = [locations[0]] + [locations[i] for i in best] + [locations[0]]
    return route, min_cost

# ----------------- Random Graph -----------------
def random_complete_graph(n: int, seed: int = 0, maxw: int = 10):
    random.seed(seed)
    dist = [[0]*n for _ in range(n)]
    for i in range(n):
        for j in range(i+1, n):
            w = random.randint(1, maxw)
            dist[i][j] = w
            dist[j][i] = w
    return dist

# ----------------- Held-Karp DP -----------------
def held_karp(dist_matrix: List[List[int]]) -> Tuple[List[int], int]:
    n = len(dist_matrix)
    if n == 0: return [], 0
    if n == 1: return [0, 0], 0

    dp = {}
    path = {}
    dp[(1 << 0, 0)] = 0

    for mask in range(1, 1 << n):
        for last_node in range(n):
```

```
            if (mask >> last_node) & 1:
                if (mask, last_node) in dp:
                    continue
                for prev_node in range(n):
                    if prev_node == last_node:
                        continue
                    if (mask >> prev_node) & 1:
                        prev_mask = mask ^ (1 << last_node)
                        if (prev_mask, prev_node) in dp:
                            current_cost = dp[(prev_mask, prev_node)] + dist_matrix[prev_node][last_node]
                            if (mask, last_node) not in dp or current_cost < dp[(mask, last_node)]:
                                dp[(mask, last_node)] = current_cost
                                path[(mask, last_node)] = prev_node

    all_visited_mask = (1 << n) - 1
    min_total_cost = float('inf')
    final_last_node = -1

    for last_n in range(1, n):
        if (all_visited_mask, last_n) in dp:
            cost_to_warehouse = dp[(all_visited_mask, last_n)] + dist_matrix[last_n][0]
            if cost_to_warehouse < min_total_cost:
                min_total_cost = cost_to_warehouse
                final_last_node = last_n

    route_indices = []
    if final_last_node != -1:
        current_mask = all_visited_mask
        current_node = final_last_node
        while current_node != 0 or current_mask != 1:
            if (current_mask, current_node) not in path:
                break
            route_indices.append(current_node)
            prev_node = path[(current_mask, current_node)]
            current_mask ^= (1 << current_node)
            current_node = prev_node
        route_indices.append(0)
        route_indices.reverse()
        route_indices.append(0)
    else:
        min_total_cost = 0 if n==0 else float('inf')

    return route_indices, min_total_cost

# ----------------- Plot route -----------------
def plot_route(locations: List[str], dist_matrix: List[List[int]], route_indices: List[int], title: str):
    if nx is None:
        print("networkx not installed: skipping plot.")
        return

    G = nx.Graph()
    n = len(locations)
```

```
    for i in range(n):
        G.add_node(i, label=locations[i])

    for i in range(n):
        for j in range(i + 1, n):
            if dist_matrix[i][j] > 0:
                G.add_edge(i, j, weight=dist_matrix[i][j])

    pos = nx.spring_layout(G, seed=42)
    plt.figure(figsize=(7, 6))
    nx.draw(G, pos, with_labels=True, labels={i: locations[i] for i in range(n)}, node_size=700, font_size=10, font_color='black')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=nx.get_edge_attributes(G, 'weight'), font_color='red')

    edges_to_highlight = []
    for i in range(len(route_indices) - 1):
        u = route_indices[i]
        v = route_indices[i+1]
        edges_to_highlight.append((u, v))

    nx.draw_networkx_edges(G, pos, edgelist=edges_to_highlight, width=3, edge_color='blue')

    plt.title(title)
    plt.axis('off')
    plt.show()

# ---------------- Profiling ----------------
def profile_tsp():
    results=[]
    print("=== PROFILING: TSP runtimes & memory (n = 3..6) ===")
    for n in range(3,7):
        dist = random_complete_graph(n, seed=100+n, maxw=10)
        locs_for_profiling = [str(i) for i in range(n)]

        tracemalloc.start()
        t0 = time.perf_counter()
        route_bf_indices, cost_bf = tsp_bruteforce(locs_for_profiling, dist)
        t1 = time.perf_counter()
        cur, peak = tracemalloc.get_traced_memory()
        tracemalloc.stop()
        bf_time = t1 - t0
        bf_peak = peak/1024.0
        print(f"Brute n={n}: time={bf_time:.6f}s peak_kb={bf_peak:.1f}")
        results.append(('brute', n, bf_time, bf_peak))

        tracemalloc.start()
        t0 = time.perf_counter()
        route_hk_indices, cost_hk = held_karp(dist)
        t1 = time.perf_counter()
        cur, peak = tracemalloc.get_traced_memory()
        tracemalloc.stop()
        hk_time = t1 - t0
        hk_peak = peak/1024.0
        print(f"Held-Karp n={n}: time={hk_time:.6f}s peak_kb={hk_peak:.1f}")
```

```
            results.append(('heldkarp', n, hk_time, hk_peak))
        return results

    # ------------------ Driver demo ------------------
    locations = ['Warehouse', 'C1', 'C2', 'C3']
    distance_matrix = [
        [0, 4, 8, 6],
        [4, 0, 5, 7],
        [8, 5, 0, 3],
        [6, 7, 3, 0]
    ]
    parcels = {
        'C1': {'value': 50, 'time': (9, 12), 'weight': 10},
        'C2': {'value': 60, 'time': (10, 13), 'weight': 20},
        'C3': {'value': 40, 'time': (11, 14), 'weight': 15}
    }
    vehicle_capacity = 30

    def run_all():
        print("=== RUNNING UNIT 1 ===")
        n = len(locations)
        print("Held-Karp DP cost (unit1):", held_karp_unit1_impl(0, 1, {}, n))

        print("\n=== RUNNING UNIT 2 ===")
        chosen_u2, value_u2 = greedy_select_parcels(parcels, vehicle_capacity)
        print("Unit2 chosen parcels:", chosen_u2, "total value:", value_u2)

        print("\n=== RUNNING UNIT 3 ===")
        dist_u3, parent_u3 = dijkstra(distance_matrix, 0)
        print("Unit3 Dijkstra dist:", dist_u3)

        print("\n=== RUNNING UNIT 4 ===")
        route_u4, cost_u4 = tsp_brute_force(locations, distance_matrix)
        print("Unit4 brute route:", route_u4, "cost:", cost_u4)

        print("\n=== RUNNING PROFILING ===")
        profile_tsp()
        print("\nDemo finished.")

    if __name__ == "__main__":
        run_all()
```

```
=== RUNNING UNIT 1 ===
Held-Karp DP cost (unit1): 18

=== RUNNING UNIT 2 ===
Unit2 chosen parcels: ['C1', 'C2'] total value: 110

=== RUNNING UNIT 3 ===
Unit3 Dijkstra dist: [0, 4, 8, 6]

=== RUNNING UNIT 4 ===
Unit4 brute route: ['Warehouse', 'C1', 'C2', 'C3', 'Warehouse'] cost: 18

=== RUNNING PROFILING ===
=== PROFILING: TSP runtimes & memory (n = 3..6) ===
Brute n=3: time=0.000112s peak_kb=0.5
Held-Karp n=3: time=0.000130s peak_kb=0.5
Brute n=4: time=0.000099s peak_kb=0.7
Held-Karp n=4: time=0.000946s peak_kb=1.5
Brute n=5: time=0.000408s peak_kb=1.7
Held-Karp n=5: time=0.000651s peak_kb=2.8
Brute n=6: time=0.001872s peak_kb=3.1
Held-Karp n=6: time=0.002273s peak_kb=5.5

Demo finished.
```

Code Link: [LAB DAA 5 - Colab]

Repository Link: [suhaniahujaa/Design-and-Analysis-Lab-assignments]