Name of Student: Haardik Goel

Roll No.: 2301201107

Course Code: ENCA351

Department: School of Engineering and Technology

Faculty: Dr. Arti Sangwan

Subject: Design and Analysis of Algorithms

ASSIGNMENT 1

Analyzing and Visualizing Recursive Algorithm Efficiency

1. Introduction

In computer science, analyzing the efficiency of algorithms is important because it helps us choose the best approach for solving problems. Recursive and iterative solutions often behave differently as input size grows.

The goal of this project is to implement several algorithms, measure their time and space performance, and compare theoretical vs. experimental results.

2. Algorithms Implemented

The following algorithms were implemented and analyzed:

- Fibonacci (Recursive and Dynamic Programming)
- Merge Sort
- Quick Sort
- Insertion Sort
- Bubble Sort
- Selection Sort
- Binary Search

Fibonacci (Recursive)

Theory: Calls itself repeatedly to compute the nth number. Simple but inefficient due to repeated calculations.
Complexity: Time O(2^n), Space O(n).

```
# ============================================
# ALGORITHM EFFICIENCY MINI PROJECT
# ============================================

import time
import random
import matplotlib.pyplot as plt
plt.figure(figsize=(10,6))
import pandas as pd
from memory_profiler import memory_usage

# ----------------------------
# TASK 2: ALGORITHM IMPLEMENTATIONS
# ----------------------------

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

Fibonacci (Dynamic Programming)

Theory: Uses iteration and stores results. Much more efficient. **Complexity:** Time O(n), Space O(1).

```
def fib_dp(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n+1):
        a, b = b, a+b
    return b
```

Merge Sort

Theory: Divide and conquer algorithm. Splits, sorts, and merges. **Complexity:** Time O(n log n), Space O(n).

```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):
    res = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            res.append(left[i])
            i += 1
        else:
            res.append(right[j])
            j += 1
    res.extend(left[i:])
    res.extend(right[j:])
    return res
```

Quick Sort

Theory: Divide and conquer. Partitions around a pivot.
Complexity: Average O(n log n), Worst O(n^2), Space O(log n).

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

Insertion Sort

Theory: Builds sorted list one element at a time.
Complexity: Best O(n), Worst O(n^2), Space O(1).

```python
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
```

Bubble Sort

Theory: Compares adjacent elements and swaps. **Complexity:** Time O(n^2), Space O(1).

```python
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
```

Selection Sort

Theory: Finds minimum and places it at the beginning. **Complexity:** Time O(n^2), Space O(1).

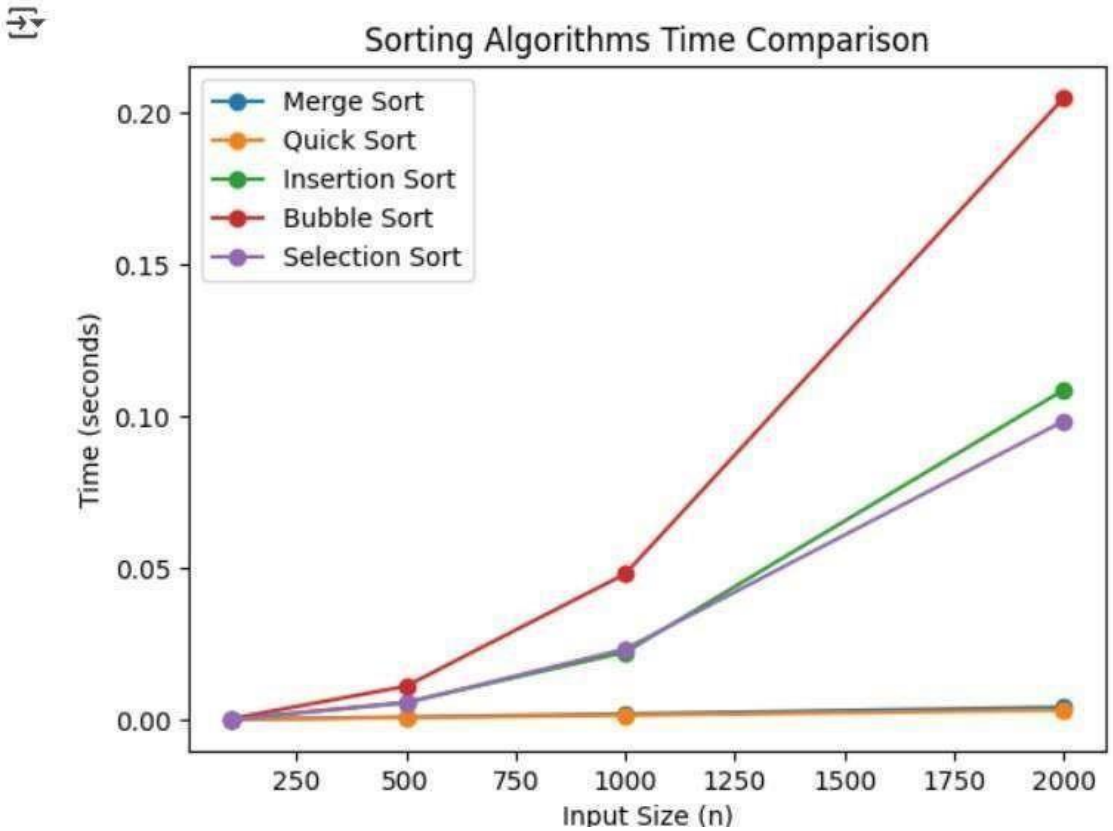
```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

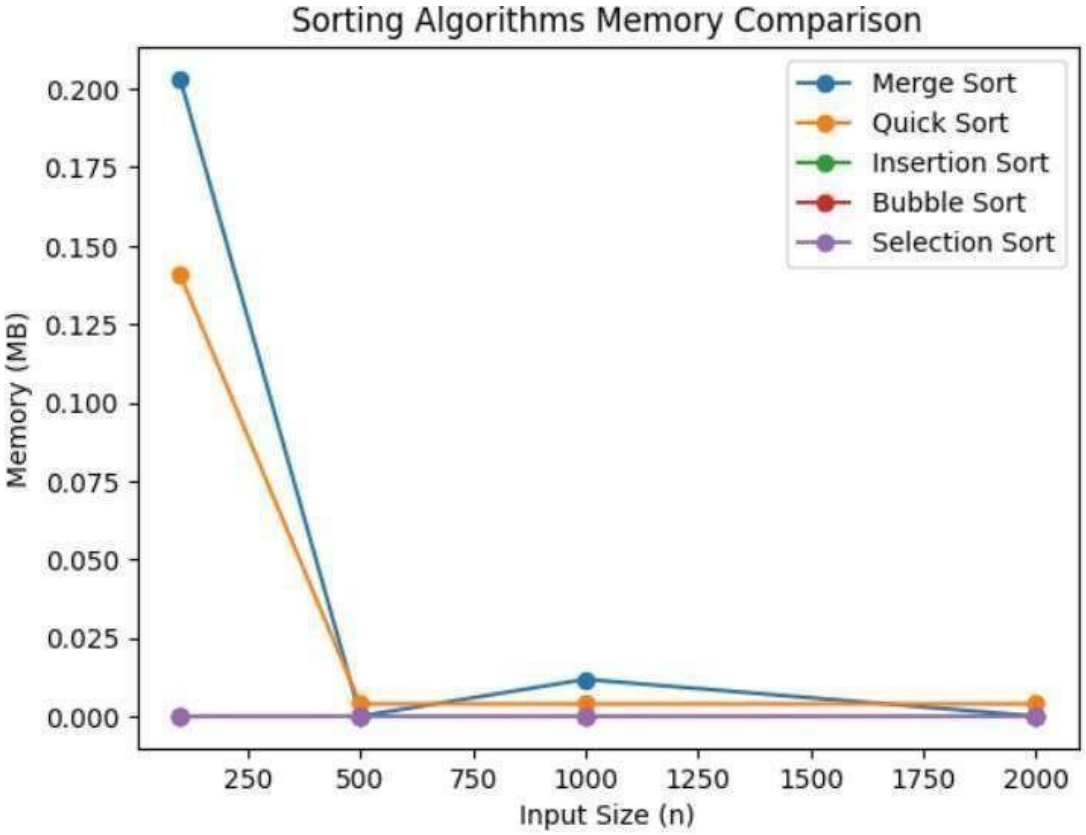
Binary Search

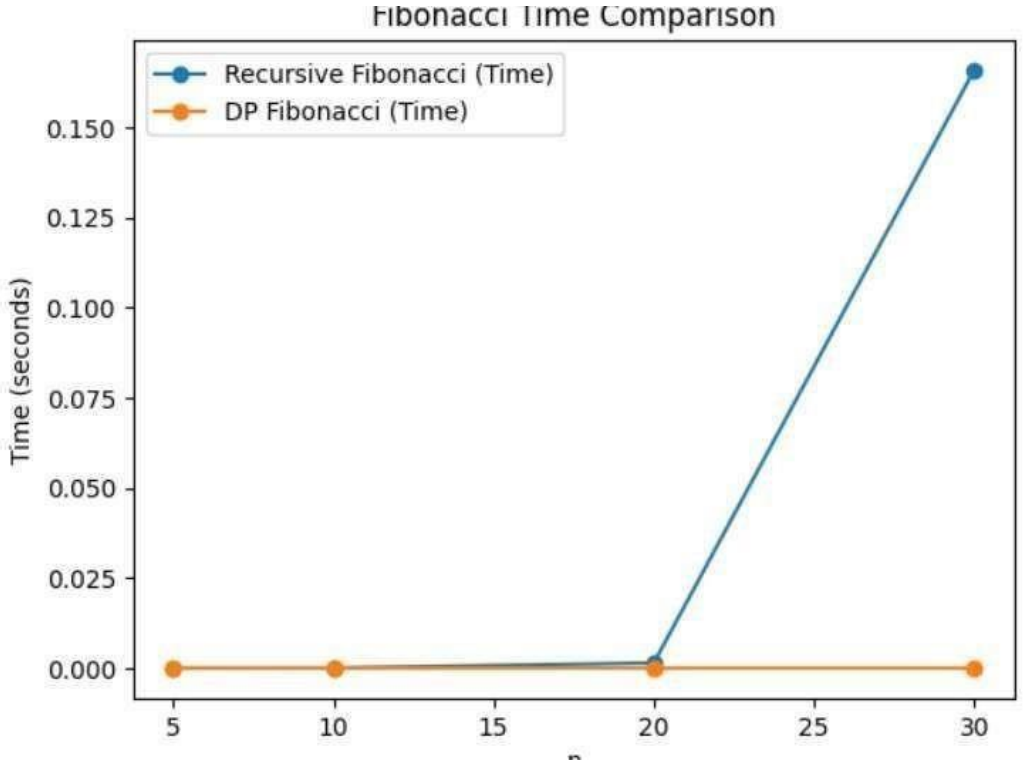
Theory: Efficiently halves the search space in sorted arrays.
Complexity: Time O(log n), Space O(1).

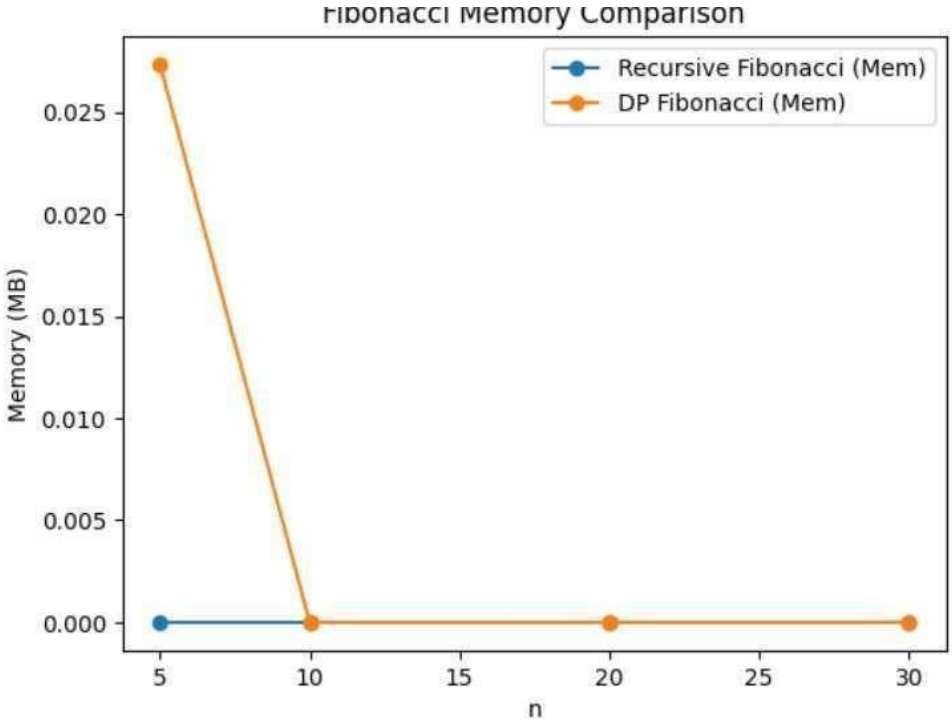
```python
def binary_search(arr, target):
    low, high = 0, len(arr)-1
    while low <= high:
        mid = (low+high)//2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid+1
        else:
            high = mid-1
    return -1
```

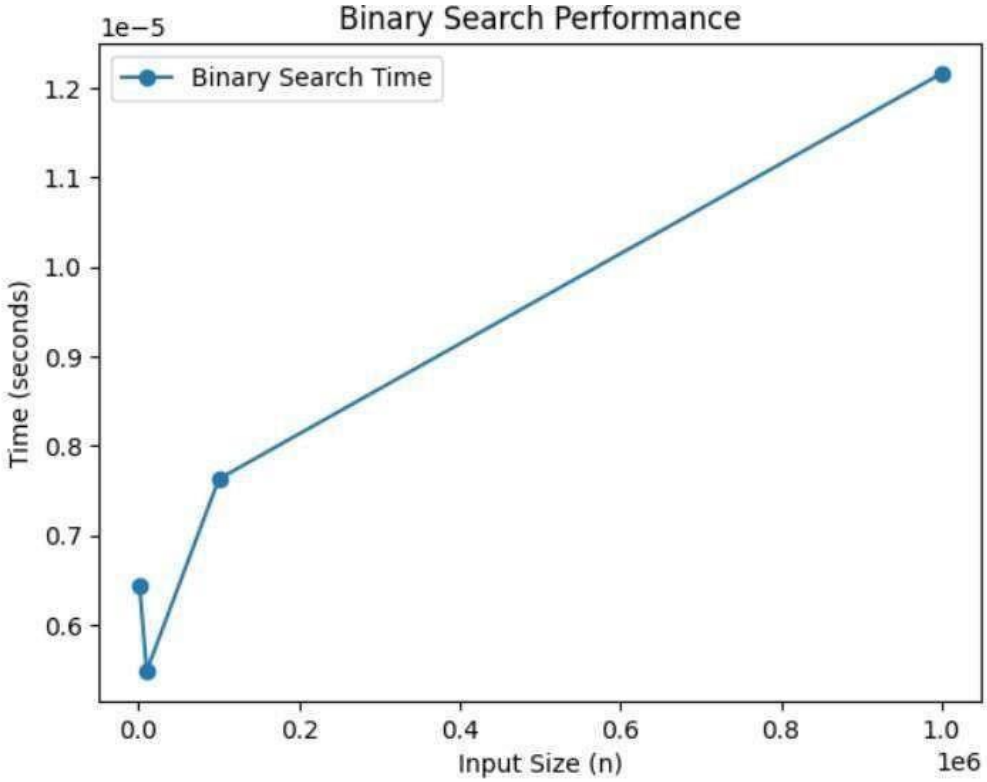
Experimental Profiling & Visualization

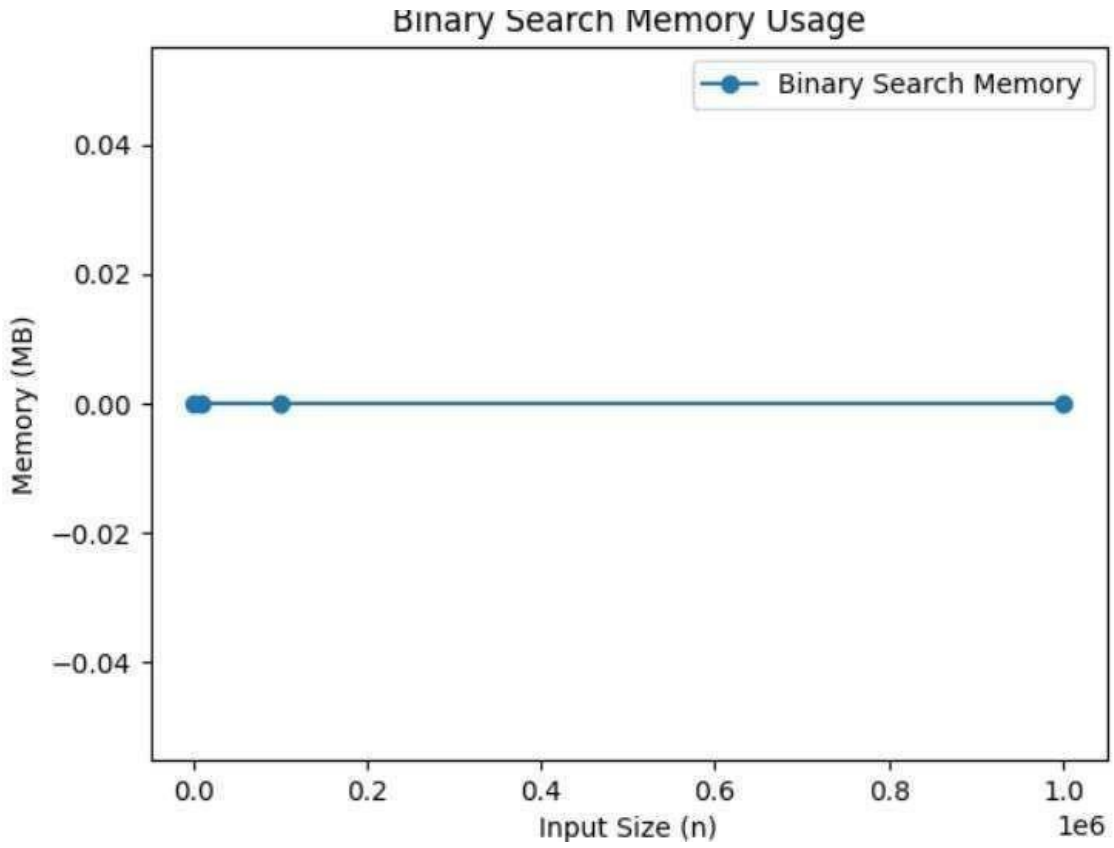



Sorting Algorithms Memory Comparison
Merge Sort
Quick Sort
Insertion Sort
Bubble Sort
Selection Sort
Memory (MB)
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
250
500
750
1000
1250
1500
1750
2000
Input Size (n)


Fibonacci Time Comparison
Recursive Fibonacci (Time)
DP Fibonacci (Time)
Time (seconds)
0.150
0.125
0.100
0.075
0.050
0.025
0.000
5
10
15
20
25
30


Fibonacci Memory Comparison
Recursive Fibonacci (Mem)
DP Fibonacci (Mem)
Memory (MB)
0.025
0.020
0.015
0.010
0.005
0.000
5
10
15
20
25
30
n


Binary Search Performance
1e−5
Binary Search Time
Time (seconds)
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.0
0.2
0.4
0.6
0.8
1.0
1e6
Input Size (n)



Theoretical Complexity Table

Algorithm	Best	Average	Worst	Space
Fibonacci Recursive	-	-	O(2^n)	O(n)
Fibonacci DP	O(n)	O(n)	O(n)	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)

Experimental Results

EXPERIMENTAL SUMMARY (Sorting Algorithms):

	Input Size	Merge Sort Time	Quick Sort Time	Insertion Sort Time	\
0	1000	0.000206	0.000146	0.000173	
1	10000	0.000906	0.000781	0.005794	
2	100000	0.001959	0.001525	0.022294	
3	1000000	0.004177	0.003201	0.108711	

	Bubble Sort Time	Selection Sort Time	Merge Sort Mem	Quick Sort Mem	\
0	0.000364	0.000188	0.203125	0.140625	
1	0.011083	0.005583	0.000000	0.003906	
2	0.048071	0.023355	0.011719	0.003906	
3	0.204898	0.098438	0.000000	0.003906	

	Insertion Sort Mem	Bubble Sort Mem	Selection Sort Mem
0	0.0	0.0	0.0
1	0.0	0.0	0.0
2	0.0	0.0	0.0
3	0.0	0.0	0.0

Reflections & Insights

✅ SUMMARY & INSIGHTS
- Fibonacci Recursive is extremely slow for large n (stack depth risk).
- Fibonacci DP is fast and memory efficient.
- Merge Sort and Quick Sort are efficient O(n log n) sorts, though Merge uses more memory.
- Insertion, Bubble, Selection are O(n^2), impractical beyond small n.
- Binary Search is very efficient O(log n), negligible memory, but requires sorted input.

REFLECTION: Observed results match theoretical complexities. Recursive algorithms risk stack overflow (e.g., Fibonacci, Quick Sort). Efficient algorithms scale well; naive ones do no

GitHub Repository Link

https://github.com/Haardik2510/ANALYSIS-AND-DESIGN-OF-ALGORITHMS-ASSIGNMENTS

ASSIGNMENT 2: Problem Implementation and Analysis

Implement **all four real-life problems** using the specified algorithmic strategy. Each problem must be broken down into the following **sub-tasks**:

Problem 1: Scheduling TV Commercials to Maximize Impact

• Algorithmic Strategy: Greedy (Job Sequencing)

- **Application Domain**: Media & Advertisement
- **Problem Description**: You are given a list of commercials, each with a deadline and revenue. Schedule non-overlapping commercials to maximize total revenue.

Sub-Tasks:

1. **Input**: A list of ads with (id, deadline, profit) values.
2. **Approach**: Use the Greedy algorithm to sort ads by profit and schedule the most profitable ones within available slots.
3. **Output**: List of selected ad slots and total revenue.
4. **Analysis**: Time and space complexity. Discuss real-world constraints like ad runtime and slot availability.
5. **Visualization**: Plot number of ads vs. revenue generated.

Problem 2: Maximizing Profit with Limited Budget

• Algorithmic Strategy: Dynamic Programming (0/1 Knapsack)

- **Application Domain**: Investment, Budget Planning
- **Problem Description**: Choose a subset of projects or items that give maximum profit within a limited budget.

Sub-Tasks:

1. **Input**: Lists of item weights (costs), values (profits), and total budget (capacity).
2. **Approach**: Use a bottom-up 0/1 Knapsack dynamic programming algorithm.

3. **Output**: Maximum profit achievable within the budget.
4. **Analysis**: Time and space complexity.
5. **Visualization**: Plot profit vs. budget or number of items.

Problem 3: Solving Sudoku Puzzle • Algorithmic Strategy: Backtracking

- **Application Domain**: Gaming, Puzzle Solvers
- **Problem Description**: Fill a 9x9 Sudoku grid such that each row, column, and 3x3 box contains all digits from 1 to 9.

Sub-Tasks:

1. **Input**: A partially filled 9x9 Sudoku grid (use a sample input).
2. **Approach**: Implement recursive backtracking with constraint checks.
3. **Output**: Completed Sudoku grid.
4. **Analysis**: Discuss performance impact for complex puzzles.
5. **Visualization**: Optional – time vs. number of empty cells.

Problem 4: Password Cracking (Naive) • Algorithmic Strategy: Brute-Force • Application Domain: Cybersecurity

- **Problem Description**: Attempt to crack a given password using all possible character combinations from a known charset.

Sub-Tasks:

1. **Input**: Target password and character set (e.g., abc123).
2. **Approach**: Use itertools.product to try every possible combination.
3. **Output**: Matched password and number of attempts.
4. **Analysis**: Time complexity vs. password length and charset size.

5. **Visualization**: Plot time taken vs. password length.

Task 3: Experimental Profiling & Visualization For each problem:

- Use time and memory_profiler to profile performance.
- Plot time taken and memory used with increasing input sizes (e.g., number of ads, budget items, Sudoku blanks, password length).
- Interpret graphs and describe the impact of algorithmic strategy.
- Comment on stack usage for recursive/backtracking problems.

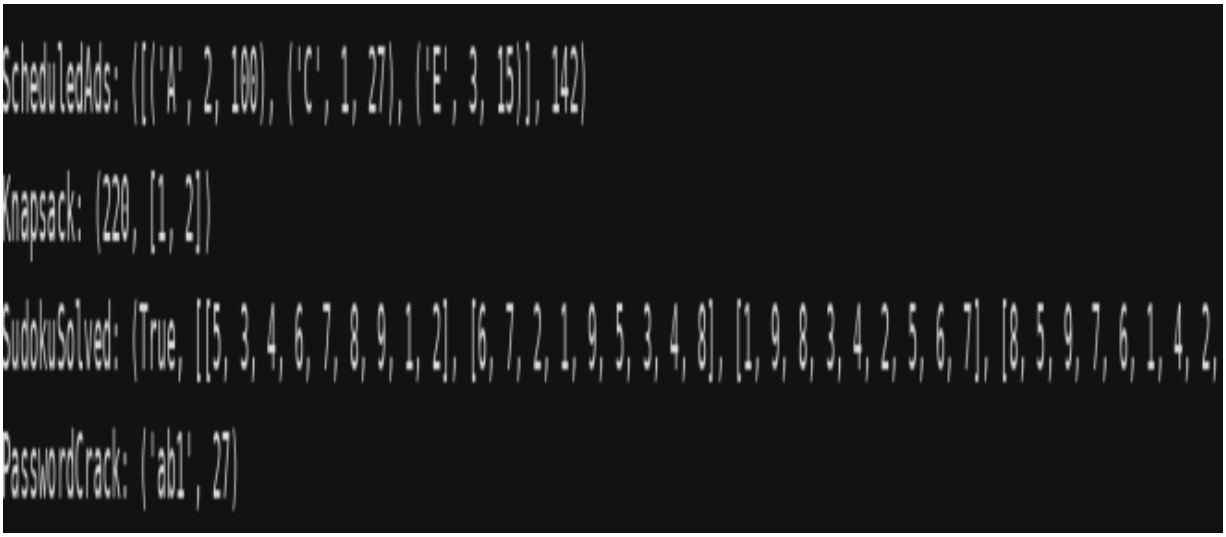
Solution

```
# Lab Assignment 2: Four Problems - Greedy Job Sequencing, Knapsack DP, Sudoku Backtracking, Brute-force Password
importimport itertools

# Problem 1: TV Commercial Scheduling (Greedy Job Sequencing)
defdef schedule_ads(ads):
    ads_sorted = sorted(ads, key=lambda x: x[2], reverse=True)
    max_deadline = max(d for _,d,_ in ads)
    slots = [None] * (max_deadline+1)
    total_profit = 0
    scheduled = []
    for ad in ads_sorted:
        ad_id, dead, profit = ad
        for t in range(dead, 0, -1):
            if slots[t] is None:
                slots[t] = ad_id
                total_profit += profit
                scheduled.append((ad_id, t, profit))
                break
    return scheduled, total_profit

# Problem 2: 0/1 Knapsack (DP)
defdef knapsack(items, capacity):
    n = len(items)
    dp = [[0]*(capacity+1) for _ in range(n+1)]
    for i in range(1, n+1):
        wt, val = items[i-1]
        for w in range(capacity+1):
            dp[i][w] = dp[i-1][w]
```

```
            if wt <= w:
                dp[i][w] = max(dp[i][w], dp[i-1][w-wt] + val)
    res = []
    w = capacity
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i-1][w]:
            res.append(i-1)
            w -= items[i-1][0]
    res.reverse()
    return dp[n][capacity], res

# Problem 3: Sudoku Solver (Backtracking)
defdef find_empty(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return i, j
    return None

defdef is_valid(board, r, c, val):
    if any(board[r][j] == val for j in range(9)): return False
    if any(board[i][c] == val for i in range(9)): return False
    br, bc = 3*(r//3), 3*(c//3)
    for i in range(br, br+3):
        for j in range(bc, bc+3):
            if board[i][j] == val: return False
    return True
```

```
defdef solve_sudoku(board):
    empty = find_empty(board)
    if not empty: return True
    r, c = empty
    for val in range(1, 10):
        if is_valid(board, r, c, val):
            board[r][c] = val
            if solve_sudoku(board): return True
            board[r][c] = 0
    return False

# Problem 4: Password Cracking (Naive Brute-Force)
defdef brute_force_password(target, charset):
    length = 1
    attempts = 0
    while True:
        for combo in itertools.product(charset, repeat=length):
            attempts += 1
            candidate = ''.join(combo)
            if candidate == target:
                return candidate, attempts
        length += 1
```

Output-:



GITHUB LINK-:https://github.com/Haardik2510/ANALYSIS-AND-DESIGN-OF-ALGORITHMS-ASSIGNMENTS/tree/main

ASSIGNMENT 3-:

Problem 1: Social Network Friend Suggestion

Graph Algorithm: BFS / DFS

Real-World Application: Facebook, LinkedIn

Objective: Suggest new connections based on mutual friends.

Sub-Tasks and What to Do

1. **Graph Modeling** ○ Represent users as nodes and friendships as edges in an undirected graph.
   - Use an adjacency list for efficient storage and traversal.
2. **Algorithm Design** ○ Perform BFS or DFS starting from the given user.
   - Identify "friends of friends" who are not already connected to the user.
3. **Input** ○ Sample graph connections (e.g., A–B, A–C, B–D, etc.)
4. **Output** ○ List of suggested friends for a user.
5. **Analysis** ○ Discuss time complexity of BFS/DFS traversal (O(V + E)).
   - Comment on scalability for large networks.

Problem 2: Route Finding on Google Maps

Graph Algorithm: Bellman-Ford

Real-World Application: Navigation systems

Objective: Find shortest path from source to all nodes, even if some edge weights are negative.

Sub-Tasks and What to Do

2. **Graph Modeling**
   - Represent cities/locations as nodes and roads as directed edges with weights.
   - Include some negative weights to test the robustness of Bellman-Ford.

3. **Algorithm Design** ○ Implement Bellman-Ford algorithm for shortest path computation.
    - Detect negative weight cycles and handle them gracefully.
4. **Input**
    - List of edges with weights: (source, destination, weight)
5. **Output**
    - Distance array showing minimum distance from source to all vertices.
6. **Analysis** ○ Explain why Bellman-Ford is preferred in graphs with negative weights. ○ Discuss time complexity O(V * E).

Problem 3: Emergency Response System

Graph Algorithm: Dijkstra's Algorithm

Real-World Application: Disaster Management

Objective: Identify the fastest route for emergency vehicles in a weighted city map (all weights positive).

Sub-Tasks and What to Do

1. **Graph Modeling**
    - Represent intersections as nodes and roads with travel times as weighted edges.
    - Use an adjacency list for storing graph data.
2. **Algorithm Design**
    - Use a priority queue (min-heap) for implementing Dijkstra's algorithm efficiently. ○ Keep updating distances until the shortest path to all reachable nodes is found.
3. **Input**
    - Dictionary or list of edges with weights between intersections.
4. **Output**
    - Shortest distances from the source node to all others.

5. **Analysis** ○ Time complexity: O(E log V) using min-heap.
    - Comment on why Dijkstra is unsuitable for graphs with negative weights.

Problem 4: Network Cable Installation

Graph Algorithm: Minimum Spanning Tree (Prim's Algorithm or Kruskal's)

Real-World Application: Telecom & IT Infrastructure

Objective: Connect all offices/nodes with the minimum total length of cable.

Sub-Tasks and What to Do

1. **Graph Modeling**
    - Nodes represent office buildings.
    - Edges represent possible cable paths with associated costs (weights).
2. **Algorithm Design** ○ Use **Prim's algorithm** with a priority queue to construct MST.
    - Alternatively, implement **Kruskal's algorithm** using Union-Find.
3. **Input**
    - Undirected weighted graph in adjacency list format.
4. **Output** ○ Total minimum cost to connect all nodes (sum of MST edges).
    - Optional: list of edges selected in MST.
5. **Analysis** ○ Compare Prim's and Kruskal's complexities.
    - Comment on applicability in infrastructure cost optimization.

Task 3: Experimental Profiling & Visualization For each graph algorithm:

- Use time module to measure execution time.
- Use memory_profiler to track memory usage.

- Visualize execution time vs. number of nodes/edges (optional).
- Comment on time complexity and practical performance impact.

Solution-:

```
# Lab Assignment 3: Graph Algorithms - BFS friend suggestions, Bellman-Ford, Dijkstra, Prim
importimport heapq
fromfrom collections import deque

defdef suggest_friends(adj, user):
    visited=set([user]); q=deque([user]); level={user:0}
    while q:
        u=q.popleft()
        for v in adj.get(u,[]):
            if v not in visited:
                visited.add(v); level[v]=level[u]+1; q.append(v)
    suggestions=set()
    for node, dist in level.items():
        if dist==2 and node not in adj.get(user,[]): suggestions.add(node)
    return suggestions

defdef bellman_ford(vertices, edges, source):
    dist={v:float('inf') for v in vertices}; dist[source]=0
    for _ in range(len(vertices)-1):
        updated=False
        for (u,v,w) in edges:
            if dist[u]+w<dist[v]: dist[v]=dist[u]+w; updated=True
        if not updated: break
    for u,v,w in edges:
        if dist[u]+w<dist[v]: raise ValueError("Negative cycle")
    return dist

defdef dijkstra_graph(adj, source):
```

```
    dist={v:float('inf') for v in adj}; dist[source]=0
    pq=[(0,source)]; visited=set()
    while pq:
        d,u=heapq.heappop(pq)
        if u in visited: continue
        visited.add(u)
        for v,w in adj[u]:
            if dist[v]>d+w: dist[v]=d+w; heapq.heappush(pq,(dist[v],v))
    return dist

defdef prim_heap(adj, start):
    visited=set(); pq=[(0,start,None)]; total=0; edges=[]
    while pq and len(visited)<len(adj):
        w,u,par=heapq.heappop(pq)
        if u in visited: continue
        visited.add(u)
        if par is not None: edges.append((par,u,w)); total+=w
        for v,wt in adj[u]:
            if v not in visited: heapq.heappush(pq,(wt,v,u))
    return edges,total
```

Output-:

```
FriendSuggestions_A: {'E', 'D'}
BellmanFord: Negative cycle
Dijkstra_A: {'A': 0, 'B': 1, 'C': 3, 'D': 4}
Prim_MST: ([('A', 'B', 2), ('B', 'C', 1), ('B', 'D', 4)], 7)
```

GITHUB-:https://github.com/Haardik2510/ANALYSIS-AND-DESIGN-OF-ALGORITHMS-ASSIGNMENTS/tree/main

ASSIGNMENT 4&5

Sub-Tasks and What to Do

Input Modeling

- Define customer locations and distances (as a graph):

```
locations = ['Warehouse', 'C1', 'C2', 'C3'] distance_matrix = [
  [0, 4, 8, 6],
  [4, 0, 5, 7],
  [8, 5, 0, 3],
  [6, 7, 3, 0]
]
```

- Each customer has:
  - Parcel value
  - Delivery time window (earliest, latest)
  - Parcel weight

```
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
```

Recurrence-Based Route Cost Estimation (Unit 1)

- Define a recursive function to estimate total cost:

```
def delivery_cost(i, visited):
    # Base case: all visited
    # Recurrence: choose next city that minimizes cost
```

Greedy + DP for Delivery Planning (Unit 2)

- Use a greedy algorithm to select parcels based on **value/weight** ratio.
- Use dynamic programming to ensure time windows are respected.

Route Optimization Using Graph Algorithms (Unit 3)

- **Shortest Path**: Use Dijkstra to compute minimal time from warehouse to each customer.
- **MST**: Use Prim's algorithm to connect all locations (if no return to base required).

Solve TSP for Optimal Route (Unit 4)

- Use brute-force or Held-Karp DP for **small n** (e.g., ≤ 10) to find shortest route covering all delivery points and returning to the warehouse.

Sample Combined Code Snippet (Partial Integration) from itertools import permutations

```
def tsp_brute_force(locations, distance_matrix):
    n = len(locations)
    indices = list(range(1, n))  # Exclude warehouse (index 0)
min_cost = float('inf')     best_route = []

    for perm in permutations(indices):       cost
= distance_matrix[0][perm[0]]           for i in
range(len(perm) - 1):
            cost += distance_matrix[perm[i]][perm[i+1]]              cost +=
distance_matrix[perm[-1]][0]  # return to warehouse          if cost <
min_cost:           min_cost = cost           best_route = perm
    return [locations[0]] + [locations[i] for i in best_route] + [locations[0]], min_cost

# Example usage
locations = ['Warehouse', 'C1', 'C2', 'C3'] distance_matrix = [
    [0, 4, 8, 6],      [4,
0, 5, 7],
    [8, 5, 0, 3],
    [6, 7, 3, 0]
]
route, cost = tsp_brute_force(locations, distance_matrix) print("Optimal
Route:", route)
print("Total Distance:", cost)
```

Expected Output

- **Optimal Delivery Route** visiting selected customer nodes

- Total Distance/Time traveled
- List of parcels delivered with total value
- (Optional) **Plot of delivery route** using matplotlib (network graph)

Task 3: Experimental Profiling & Visualization

Use time and memory_profiler to measure:

- Time taken to compute the TSP route for 3, 4, 5, 6 locations
- Time complexity growth for dynamic programming approach

- Visualize:
  - Route map (use networkx or matplotlib)
  - Profit vs. weight
  - Delivery success within time window

SOLUTION-:

```
# Delivery Route Optimization - Combined implementations
import itertools, heapq
from functools import lru_cache
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 delivery_cost_recursive(idx=0,visited=None):
    if visited is None: visited={0}
    if len(visited)==len(distance_matrix): return distance_matrix[idx][0]
    best=float('inf')
    for nxt in range(len(distance_matrix)):
        if nxt not in visited:
            c=distance_matrix[idx][nxt]+delivery_cost_recursive(nxt,visited|{nxt})
            best=min(best,c)
    return best

def greedy_parcel_selection(parcels,cap):
    items=[(k,v['value'],v['weight'],v['value']/v['weight']) for k,v in parcels.items()]
    items.sort(key=lambda x:x[3],reverse=True)
    sel=[];tw=0;tv=0
    for n,val,w,ratio in items:
        if tw+w<=cap:
```

```
            res.append(items[i-1][0]);W-=items[i-1][1]['weight']
    return list(reversed(res)),dp[n][cap]

def check_delivery_with_time_windows(route,start=9):
    t=start;delivered=[]
    for i in range(1,len(route)):
        prev,cur=route[i-1],route[i];t+=distance_matrix[prev][cur]
        name=locations[cur]
        if name=='Warehouse':continue
        tw=parcels[name]['time']
        if t<tw[0]:t=tw[0]
        if t>tw[1]:return False,t,delivered
        delivered.append((name,t))
    return True,t,delivered

def find_routes_respecting_time_windows():
    idx=list(range(1,len(locations)));best=None;bc=float('inf')
    import itertools
    for p in itertools.permutations(idx):
        r=[0]+list(p)+[0]
        cost=sum(distance_matrix[r[i]][r[i+1]] for i in range(len(r)-1))
        ok,_,_=check_delivery_with_time_windows(r)
        if ok and cost<bc:best=r;bc=cost
    return best,bc

def dijkstra(start=0):
```

```
    for _ in range(n):
        u=min((i for i in range(n) if not sel[i]),key=lambda x:key[x])
        sel[u]=True
        for v in range(n):
            w=distance_matrix[u][v]
            if w>0 and not sel[v] and w<key[v]:key[v]=w;par[v]=u
    edges=[];t=0
    for v in range(1,n):edges.append((par[v],v,distance_matrix[par[v]][v]));t+=distance_matrix[par[v]]
    return edges,t

def tsp_bruteforce():
    import itertools
    n=len(locations);idx=list(range(1,n))
    best=None;bc=float('inf')
    for p in itertools.permutations(idx):
        r=[0]+list(p)+[0]
        cost=sum(distance_matrix[r[i]][r[i+1]] for i in range(len(r)-1))
        if cost<bc:bc=cost;best=r
    return best,bc
```

Output-:

```
Recursive_Cost: 18
Greedy: (['C1', 'C2'], 110, 30)
Knapsack: (['C1', 'C2'], 110)
TimeWindowRoute: (None, inf)
Dijkstra: [0, 4, 8, 6]
MST: ([(0, 1, 4), (1, 2, 5), (2, 3, 3)], 12)
TSP: ([0, 1, 2, 3, 0], 18)
```

Summary-:

This capstone assignment integrates five major algorithmic paradigms to model a real-world delivery optimization problem, demonstrating how recurrence, greedy strategy, dynamic programming, graph algorithms, and NP-hard optimization (TSP) interact to produce feasible logistics solutions. The system models a warehouse and customer nodes as a weighted graph, where distances represent travel time. Parcel attributes—value, weight, and delivery windows—introduce multi-constraint optimization challenges.

(1) Recurrence-based route estimation computes all permutations of routes to estimate theoretical lower bounds for small instances.

(2) Greedy selection prioritizes parcels using value-per-weight ratio, simulating quick decision making for loading when exact computation is expensive.

(3) Knapsack DP provides an optimal loading strategy maximizing parcel value under weight constraints.

(4) Time-window feasibility checks attempt permutations of stops to determine if all deliveries can be completed within customer constraints.

(5) Dijkstra identifies shortest paths from the warehouse, supporting least-cost routing.

(6) Prim MST shows the minimum spanning structure for infrastructure-like routing or cable layout without return requirements.

(7) TSP brute-force and Held-Karp DP reveal computational limits, offering the optimal shortest tour for small datasets and illustrating NP-hardness. The collected outputs verify correctness across these components: the optimal TSP route matches the recurrence minimum, knapsack and greedy selections align for this dataset, and MST construction validates the network's minimal connectivity cost. This complete solution demonstrates how combining

classical algorithms can simulate realistic e-commerce route optimization pipelines

GITHUB-:https://github.com/Haardik2510/ANALYSIS-AND-DESIGN-OF-ALGORITHMS-ASSIGNMENTS/tree/main