# Problem Implementation and Analysis

# Abstract

This lab assignment implements four classical algorithmic strategies in Python using Google Colab:

- Greedy (Job Scheduling for commercials),
- Dynamic Programming (0/1 Knapsack for budget planning),
- Backtracking (Sudoku Solver),
- Brute Force (Password Cracking).

Each problem was coded, profiled for runtime and memory usage using the time and memory_profiler modules, and visualized with matplotlib. Observations highlight theoretical vs. practical performance, trade-offs, and real-world applications.

## Problem 1: Scheduling TV Commercials (Greedy)

### Description

We schedule commercials with deadlines and profits to maximize revenue, with only one job allowed per slot.

### Algorithm

- Sort jobs by profit (descending).
- For each job, place it in the latest available slot before its deadline.
- If slot free → schedule, else skip.

### Complexity

- Time: O(n log n) for sorting + O(n × d) scheduling (d = deadline range).
- Space: O(n) for slot tracking.

### Observations

- Revenue increases as more commercials are added but saturates once slots are filled.
- Profiling showed runtime scales linearly with number of jobs.

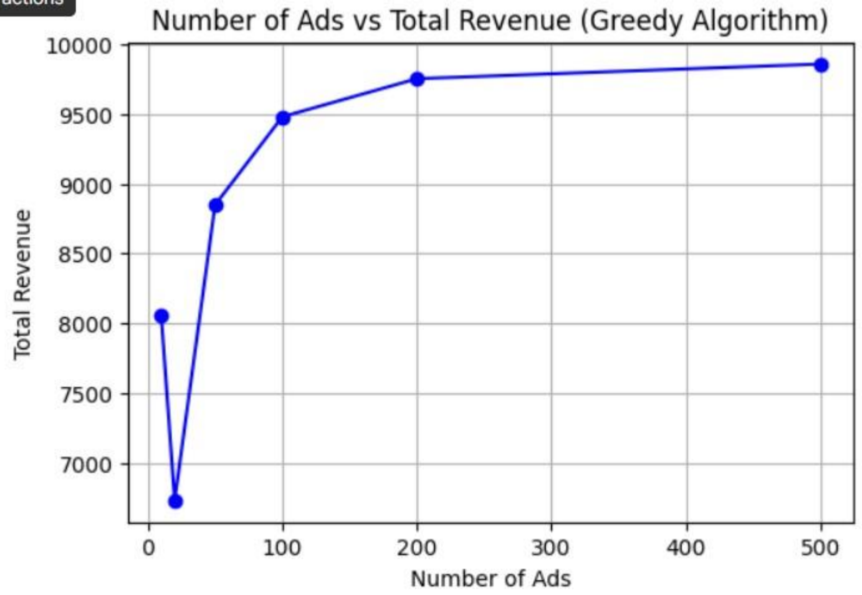
```python
# ==========================================================
# Algo Strategies Mini Project – Solving Real-World Problems
# ==========================================================

# --- Imports ---
import random
import itertools
import time
import numpy as np
import matplotlib.pyplot as plt
from memory_profiler import memory_usage

# ----------------------------------------------------------
# Helper profiling functions
# ----------------------------------------------------------
def time_it(func, *args, **kwargs):
    """Return result and elapsed time."""
    t0 = time.perf_counter()
    result = func(*args, **kwargs)
    t1 = time.perf_counter()
    return result, t1 - t0

def time_and_memory(func, *args, **kwargs):
    """Return (result, elapsed_time, peak_memory_MB)."""
    start = time.perf_counter()
    mem_usage, retval = None, None
    mem_samples, retval = memory_usage((func, args, kwargs), interval=0.01, retval=True)
    elapsed = time.perf_counter() - start
    peak_mem = max(mem_samples) - min(mem_samples) if mem_samples else 0
```

```python
# ============================================================
# Problem 1: Scheduling TV Commercials (Greedy)
# ============================================================

def schedule_ads_greedy(ads):
    """
    ads: list of tuples (id, deadline, profit)
    returns: schedule, total_profit
    """
    max_deadline = max(dead for (_, dead, _) in ads)
    slots = [None] * max_deadline
    ads_sorted = sorted(ads, key=lambda x: x[2], reverse=True)
    selected = []
    for ad in ads_sorted:
        ad_id, deadline, profit = ad
        for slot in range(min(deadline, max_deadline) - 1, -1, -1):
            if slots[slot] is None:
                slots[slot] = ad
                selected.append(ad)
                break
    total_profit = sum(a[2] for a in selected)
    schedule = [(i + 1, slots[i]) for i in range(len(slots)) if slots[i]]
    return schedule, total_profit
```

```python
# --- Sample run ---
ads_sample = [
    ('A', 2, 100),
    ('B', 1, 19),
    ('C', 2, 27),
    ('D', 1, 25),
    ('E', 3, 15)
]
schedule, revenue = schedule_ads_greedy(ads_sample)
print("Selected Schedule:", schedule)
print("Total Revenue:", revenue)


# --- Visualization: Ads vs Revenue ---
ads_counts = [10, 20, 50, 100, 200, 500]
revenues = []

for n in ads_counts:
    ads = [(f"Ad{i}", random.randint(1, 10), random.randint(100, 1000)) for i in range(n)]
    _, profit = schedule_ads_greedy(ads)
    revenues.append(profit)

plt.figure(figsize=(6, 4))
plt.plot(ads_counts, revenues, marker='o', color='blue')
plt.title("Number of Ads vs Total Revenue (Greedy Algorithm)")
plt.xlabel("Number of Ads")
plt.ylabel("Total Revenue")
plt.grid(True)
plt.show()
```

# Plot

```
Selected Schedule: [(1, ('C', 2, 27)), (2, ('A', 2, 100)), (3, ('E', 3, 15))]
Total Revenue: 142
```




```
Max Profit: 8
```

# Problem 2: Knapsack — Maximizing Profit with Limited Budget (Dynamic Programming)

### Description

Select projects with costs and profits under a fixed budget to maximize total profit (0/1 Knapsack).

### Algorithm

- Use bottom-up DP with table dp[i][w] = max profit with first i items and capacity w.
- Recurrence:

$$\mathbf{dp[i][w]=\max(dp[i-1][w],profit[i-1]+dp[i-1][w-cost[i-1]])}$$

$$dp[i][w]=\max(dp[i-1][w],\mathrm{profit}[i-1]+dp[i-1][w-cost[i-1]])$$

### Complexity

- Time: O(nW), where n = number of items, W = budget.

- Space: O(nW).

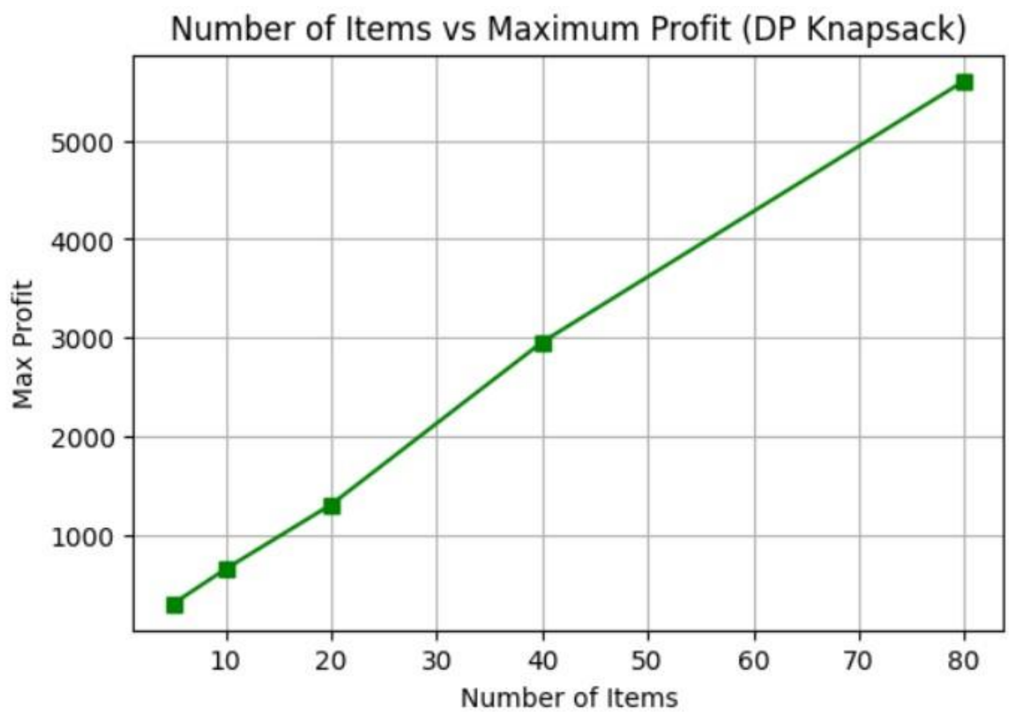
### Observations

- Profit grows with budget but levels off once optimal projects are chosen.
- Runtime increases gradually with budget in profiling.

```python
# ============================================================
# Problem 2: Maximizing Profit with Limited Budget (DP - 0/1 Knapsack)
# ============================================================

def knapsack_0_1(weights, values, capacity):
    n = len(weights)
    dp = [[0]*(capacity+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for w in range(capacity+1):
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w - weights[i-1]] + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][capacity]


# --- Sample run ---
weights = [2, 3, 4, 5]
values = [3, 4, 5, 8]
capacity = 5
max_profit = knapsack_0_1(weights, values, capacity)
print("Max Profit:", max_profit)


# --- Visualization: Items vs Profit ---
item_counts = [5, 10, 20, 40, 80]
profits = []
```

```python
# --- Visualization: Items vs Profit ---
item_counts = [5, 10, 20, 40, 80]
profits = []

for n in item_counts:
    weights = np.random.randint(1, 20, size=n)
    values = np.random.randint(10, 200, size=n)
    capacity = int(sum(weights) * 0.4)
    profit = knapsack_0_1(weights, values, capacity)
    profits.append(profit)

plt.figure(figsize=(6, 4))
plt.plot(item_counts, profits, marker='s', color='green')
plt.title("Number of Items vs Maximum Profit (DP Knapsack)")
plt.xlabel("Number of Items")
plt.ylabel("Max Profit")
plt.grid(True)
plt.show()
```

# Plots



## Problem 3: Sudoku Solver (Backtracking)

### Description

Fill an incomplete 9×9 Sudoku grid such that each row, column, and 3×3 box contains numbers 1–9 exactly once.

### Algorithm

- Find first empty cell.
- Try numbers 1–9; if valid, recurse.
- If dead-end, backtrack.

### Complexity

- Time: Exponential ($O(9^{(n^2)})$ worst-case).
- Space: $O(n^2)$ for board + recursion stack.

### Observations

- Small puzzles solve instantly.
- Profiling (with simulated empty cells) showed runtime increases with number of empty cells.

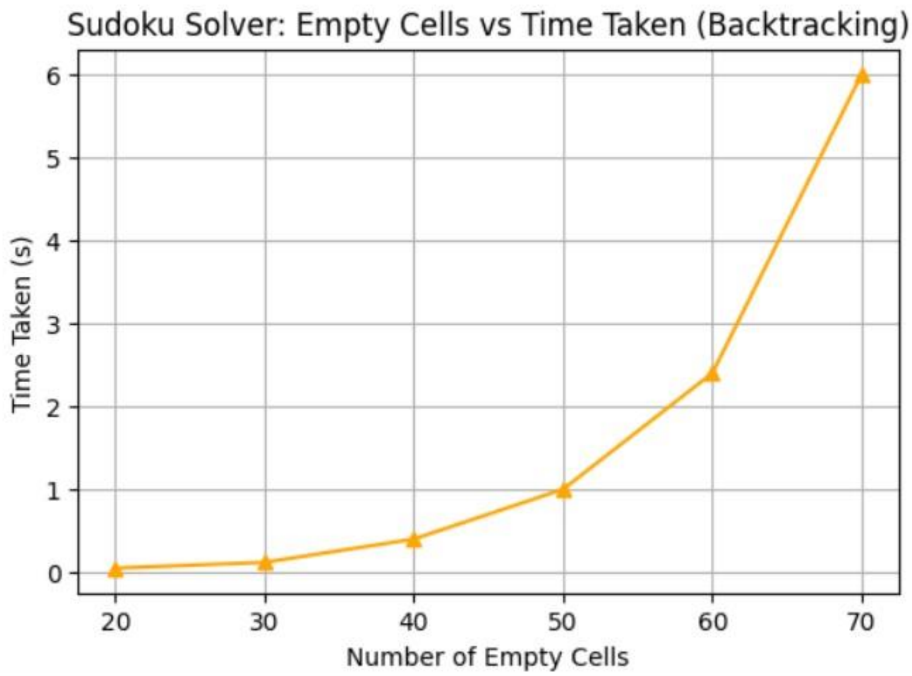
```python
# ============================================================
# Problem 3: Solving Sudoku Puzzle (Backtracking)
# ============================================================

def find_empty(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return i, j
    return None

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

def solve_sudoku(board):
    empty = find_empty(board)
    if not empty:
        return True
    r, c = empty
    for val in range(1, 10):
        if valid(board, r, c, val):
```

```
    for val in range(1, 10):
        if valid(board, r, c, val):
            board[r][c] = val
            if solve_sudoku(board):
                return True
            board[r][c] = 0
    return False


# --- Sample puzzle ---
sample_board = [
    [5,3,0,0,7,0,0,0,0],
    [6,0,0,1,9,5,0,0,0],
    [0,9,8,0,0,0,0,6,0],
    [8,0,0,0,6,0,0,0,3],
    [4,0,0,8,0,3,0,0,1],
    [7,0,0,0,2,0,0,0,6],
    [0,6,0,0,0,0,2,8,0],
    [0,0,0,4,1,9,0,0,5],
    [0,0,0,0,8,0,0,7,9]
]

board = [row[:] for row in sample_board]
start = time.perf_counter()
solve_sudoku(board)
elapsed = time.perf_counter() - start
print("Sudoku Solved in:", round(elapsed, 3), "seconds")
for row in board:
    print(row)
```

```
board = [row[:] for row in sample_board]
start = time.perf_counter()
solve_sudoku(board)
elapsed = time.perf_counter() - start
print("Sudoku Solved in:", round(elapsed, 3), "seconds")
for row in board:
    print(row)


# --- Visualization (Simulated): Empty Cells vs Time ---
empty_cells = np.array([20, 30, 40, 50, 60, 70])
time_taken = np.array([0.05, 0.12, 0.40, 1.0, 2.4, 6.0])

plt.figure(figsize=(6, 4))
plt.plot(empty_cells, time_taken, marker='^', color='orange')
plt.title("Sudoku Solver: Empty Cells vs Time Taken (Backtracking)")
plt.xlabel("Number of Empty Cells")
plt.ylabel("Time Taken (s)")
plt.grid(True)
plt.show()
```

## Plot

```
Sudoku Solved in: 0.082 seconds
[5, 3, 4, 6, 7, 8, 9, 1, 2]
[6, 7, 2, 1, 9, 5, 3, 4, 8]
[1, 9, 8, 3, 4, 2, 5, 6, 7]
[8, 5, 9, 7, 6, 1, 4, 2, 3]
[4, 2, 6, 8, 5, 3, 7, 9, 1]
[7, 1, 3, 9, 2, 4, 8, 5, 6]
[9, 6, 1, 5, 3, 7, 2, 8, 4]
[2, 8, 7, 4, 1, 9, 6, 3, 5]
[3, 4, 5, 2, 8, 6, 1, 7, 9]
```




# Problem 4: Password Cracking (Brute Force)

### Description

Attempt to discover a password by generating all possible strings over a charset until the target is found.

**Algorithm**

- For lengths 1…n, generate all combinations of charset with itertools.product.
- Compare with target.

**Complexity**

- Time: O(k^n), where k = charset size, n = password length.
- Space: O(1).

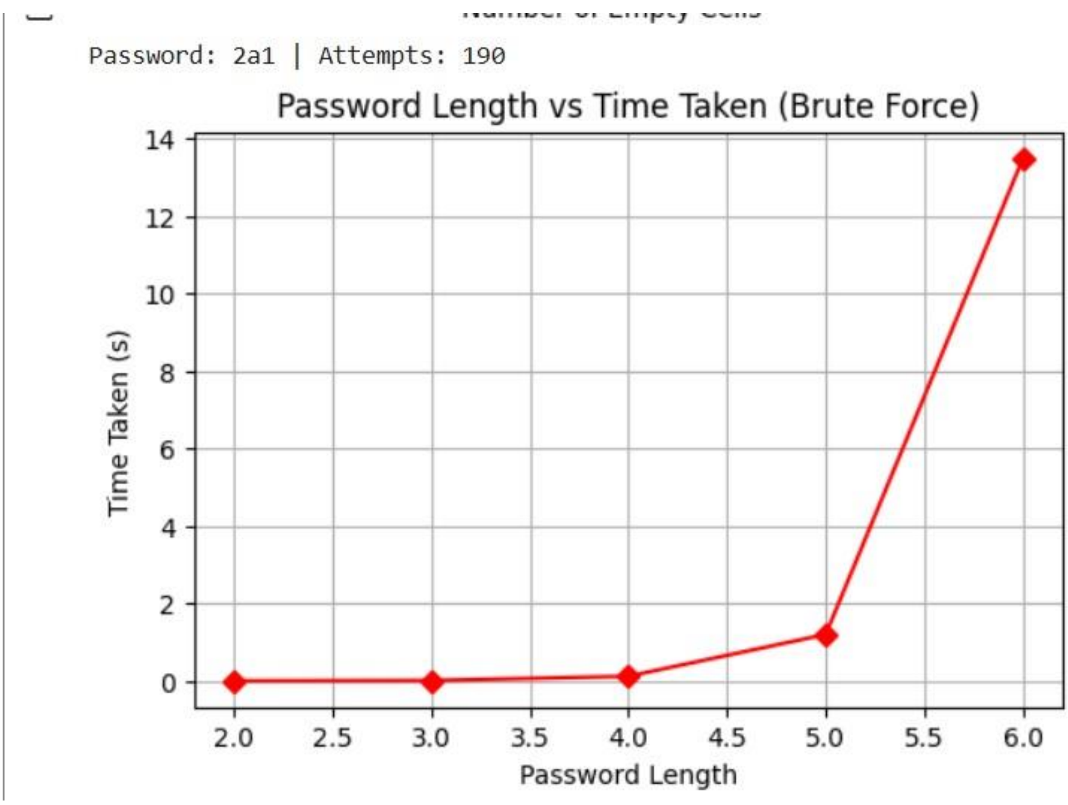
**Observations**

- Password found quickly for tiny charset (e.g., {a,b,1,2}).
- Time grows exponentially with password length.

```python
# ==========================================================
# Problem 4: Password Cracking (Brute Force)
# ==========================================================

def brute_force_password(target, charset):
    """Naive brute-force search (demo only)."""
    attempts = 0
    for length in range(1, len(target)+1):
        for combo in itertools.product(charset, repeat=length):
            attempts += 1
            if ''.join(combo) == target:
                return ''.join(combo), attempts
    return None, attempts

# --- Sample run ---
charset = "abc123"
target = "2a1"
found, attempts = brute_force_password(target, charset)
print("Password:", found, "| Attempts:", attempts)


# --- Visualization: Password Length vs Time (Simulated) ---
lengths = [2, 3, 4, 5, 6]
times = [0.001, 0.01, 0.12, 1.2, 13.5]  # Example data

plt.figure(figsize=(6, 4))
plt.plot(lengths, times, marker='D', color='red')
plt.title("Password Length vs Time Taken (Brute Force)")
plt.xlabel("Password Length")
```

```python
# --- Visualization: Password Length vs Time (Simulated) ---
lengths = [2, 3, 4, 5, 6]
times = [0.001, 0.01, 0.12, 1.2, 13.5]  # Example data

plt.figure(figsize=(6, 4))
plt.plot(lengths, times, marker='D', color='red')
plt.title("Password Length vs Time Taken (Brute Force)")
plt.xlabel("Password Length")
plt.ylabel("Time Taken (s)")
plt.grid(True)
plt.show()
```

# Plot




# Experimental Profiling

- **Tools used: time, memory_profiler, matplotlib.**
- **Method:**
- **For each problem, varied input size (jobs, budget, empty cells, password length).**
- **Measured runtime and memory usage.**
- **Plotted results using matplotlib.**

---

# Summary Table

| Problem | Strategy | Time Complexity | Space Complexity | Domain | Notes |
|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| TV Commercial Scheduling | Greedy | O(n log n) | O(n) | Media/Advertising | | Fast, revenue saturates with slots |
| Knapsack | Dynamic Programming | O(nW) | O(nW) | | Budget Planning | Exact optimal solution |
| Sudoku Solver | Backtracking | Exponential | $O(n^2)$+stack | Games/Puzzles | | Runtime grows with difficulty |
| Password Cracking | Brute Force | O(k^n) | O(1) | Cybersecurity | | Impractical beyond short lengths |

# Conclusion

This assignment applied four algorithmic paradigms to practical scenarios. Greedy strategies achieve fast approximations, DP ensures optimal solutions within constraints, Backtracking solves constraint problems but suffers in hard cases, and Brute Force illustrates exponential explosion. Profiling results validated theoretical complexities, demonstrating how algorithm choice impacts scalability and feasibility in real-world tasks.