	School of Engineering & Technology	
	Department: SOET	**Session: 2025-26**
	Program: BCA (AI&DS)	**Semester: V**
	Course Code: ENCA351	**Number of students: 188**
	Course Name: Design and Analysis of Algorithms Lab	**Faculty: Dr. Aarti**

Lab Assignment -2: Algorithmic Strategies in Real-World Problems

Instructions:

- Assignment must be submitted within the deadline communicated by the instructor at the time of release.
- Assignment must be submitted on https://lms.krmangalam.edu.in/
- You must provide a link to your GitHub repository with your submission on LMS.
- Use of ChatGPT and similar tools is strictly prohibited.
- The assignment needs to be submitted by each individual.
- This assignment carries a total of 10 marks.
- Assignment will be assessed based on the evaluation rubrics.
- Estimated Duration: 10-12 hours

Assignment Title: Solving Real-World Problems Using Algorithmic Strategies

---

Real-World Problem Context

Choosing the right algorithmic strategy is essential for solving real-life problems efficiently. This project explores four practical problem scenarios and applies suitable algorithmic strategies to design and evaluate solutions. Through implementation, profiling, and visualization, you'll gain insights into how theory translates into practice.

Learning Objectives

By completing this project, you will:

- Apply key algorithmic characteristics (finiteness, input/output, effectiveness, etc.).
- Implement and analyze algorithmic strategies in real-world contexts.

- Visualize time and space trade-offs using plots and profiling tools.
- Communicate algorithm behavior and trade-offs effectively through documentation.

Assignment Tasks Task 1: Workspace Bootstrap

- Create a private GitHub repository named: algo-strategies-mini-project<yourname>
- Add README.md and .gitignore.
- Create a Python virtual environment.
- Install necessary packages: matplotlib, numpy, memory_profiler, time, jupyter.
- Validate your environment using a basic Jupyter notebook with imports and test plots.

Task 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.

Task 4: Final Summary and Documentation Include

the following in your submission:

- A summary table comparing all four problems:

Problem	Strategy	Time Complexity	Domain	Notes
TV Commercial Scheduling	Greedy	?	Media & Advertisement	Schedule ads to maximize revenue before deadline
Knapsack	Dynamic Programming	?	Budget Planning	Maximize value without exceeding budget
Sudoku Solver	Backtracking	?	Gaming	Fill grid under constraint satisfaction
Password Cracking	Brute-Force	?	Cybersecurity	Try all combinations to guess password

- **Insights:** Observed vs. theoretical performance
- **README:** Problem overview, how to run the notebook, and citations
- **Notebook:** Structured Jupyter Notebook with markdown explanations and wellcommented code

---

Evaluation Rubric (Total: 10 Marks)

Criteria	Marks	What is Expected
1. GitHub Setup & Organization	1	Structured repo with README, .gitignore, and environment setup
2. Algorithm Implementation	3	Correct and clear code for all four problems
3. Performance Profiling & Plots	2.5	Time/memory measurements and plots for each problem

4. Analysis & Insights	2.5	Observations on complexity, trade-offs, and suitability
5. Code Quality & Documentation	1	Clean code with comments, markdown cells, and citations

---

Submission Instructions

Push the following to your GitHub repository:

- README.md: project overview and usage instructions
- algo_strategies_notebook.ipynb: all code, plots, and analysis
- images/: folder with exported plots (optional)
- requirements.txt: list of required packages
- .gitignore and virtual environment files (no need to upload the full env folder) Submit the **GitHub repository link** via LMS.

Academic Integrity & Plagiarism Policy

- This is an individual project.
- Code, plots, or writeups copied from peers or external sources will result in zero marks.
- Cite external references properly in the README.

Support Resources

- CLRS – *Introduction to Algorithms*
- Python Time and Memory Profiling Docs
- matplotlib and memory_profiler tutorials

Contact for Queries: Dr. Aarti, aarti.sangwan@krmangalam.edu.in

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-: