Problem 1: Scheduling TV Commercials to Maximize Impact

Source Code:

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

# Greedy Job Sequencing
def job_sequencing(ads):
    # Sort ads by profit in descending order
    ads.sort(key=lambda x: x[2], reverse=True)
    max_deadline = max(ad[1] for ad in ads)
    slots = [None] * (max_deadline + 1)
    total_profit = 0
    selected = []

    for ad in ads:
        ad_id, deadline, profit = ad
        for d in range(min(max_deadline, deadline), 0, -1):
            if slots[d] is None:
                slots[d] = ad
                total_profit += profit
                selected.append((d, ad))
                break

    return selected, total_profit
```

```python
# Example ads
ads = [("A1", 2, 100), ("A2", 1, 19), ("A3", 2, 27), ("A4", 1, 25), ("A5", 3, 15)]
selected, total = job_sequencing(ads)

print("Selected Ads (Slot, Ad):", selected)
print("Total Revenue:", total)

# Visualization: number of ads vs revenue
sizes = [10, 50, 100, 200]
revenues = []

for n in sizes:
    ads = [(f"Ad{i}", random.randint(1, 10), random.randint(10, 1000)) for i in range(n)]
    _, rev = job_sequencing(ads)
    revenues.append(rev)

plt.plot(sizes, revenues, marker='o')
plt.title("Number of Ads vs Revenue (Greedy Algorithm)")
plt.xlabel("Number of Ads")
plt.ylabel("Total Revenue")
plt.grid(True)
plt.show()
```

1. Input

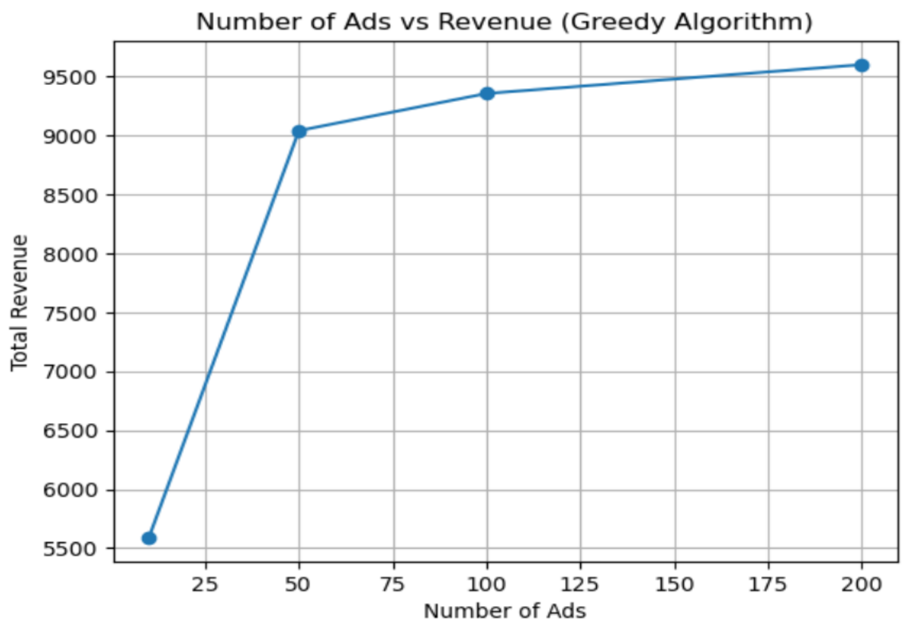
List of advertisements in the form:
(Ad ID, Deadline, Profit)
Example:
[('A1', 2, 100), ('A2', 1, 19), ('A3', 2, 27), ('A4', 1, 25), ('A5', 3, 15)]

2. Approach

- Sort all ads by descending profit.
- For each ad, find the latest available slot before its deadline.
- Assign it if the slot is free (Greedy choice).
- Continue until all slots are filled or all ads checked.

3. Output

```
Selected Ads (Slot, Ad): [(2, ('A1', 2, 100)), (1, ('A3', 2, 27)), (3, ('A5', 3, 15))]
Total Revenue: 142
```



4. Analysis

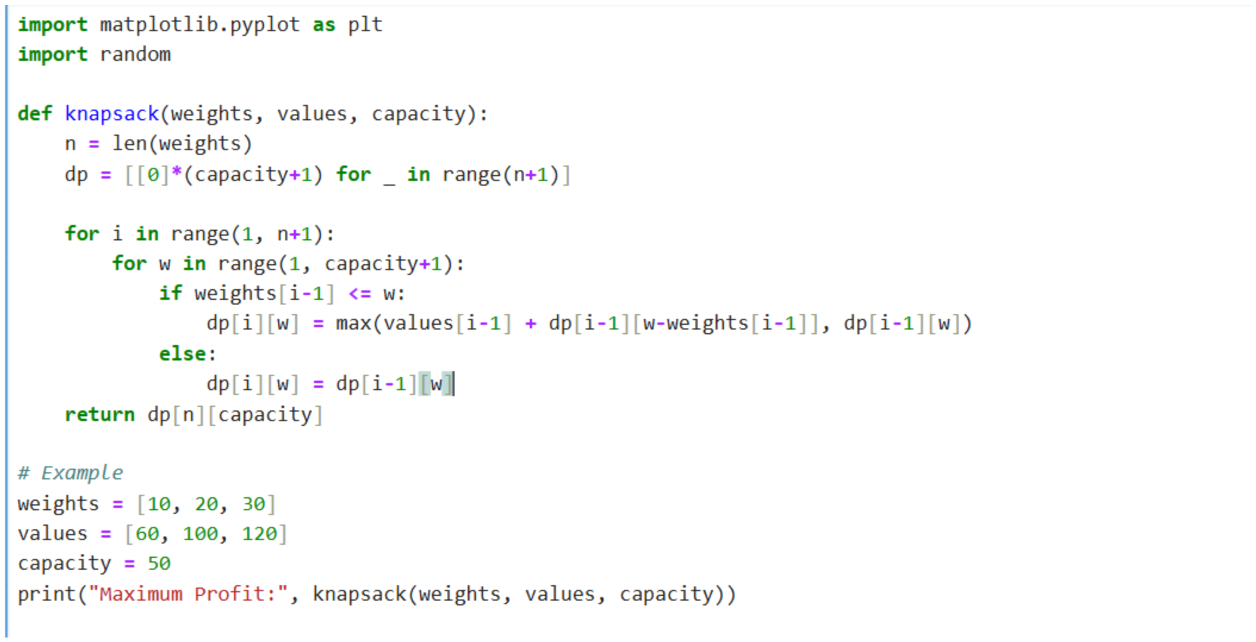
- **Time Complexity:** $O(n \log n + n^2)$ → due to sorting and slot checking
- **Space Complexity:** $O(n)$ for slot tracking
- **Real-World Constraints:**
    - Limited ad slots during peak viewing hours
    - Different durations and audience reach per slot

5. Visualization

A bar/line chart showing **Number of Ads vs. Revenue Generated**.
As ads increase, total profit rises but plateaus when all slots fill.

Problem 2: Maximizing Profit with Limited Budget

Source Code:

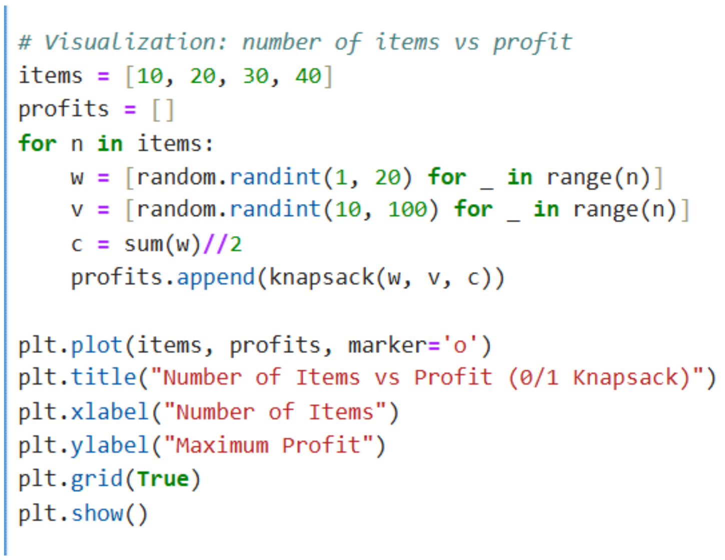


```python
import matplotlib.pyplot as plt
import random

def knapsack(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(1, capacity+1):
            if weights[i-1] <= w:
                dp[i][w] = max(values[i-1] + dp[i-1][w-weights[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][capacity]

# Example
weights = [10, 20, 30]
values = [60, 100, 120]
capacity = 50
print("Maximum Profit:", knapsack(weights, values, capacity))
```



```python
# Visualization: number of items vs profit
items = [10, 20, 30, 40]
profits = []
for n in items:
    w = [random.randint(1, 20) for _ in range(n)]
    v = [random.randint(10, 100) for _ in range(n)]
    c = sum(w)//2
    profits.append(knapsack(w, v, c))

plt.plot(items, profits, marker='o')
plt.title("Number of Items vs Profit (0/1 Knapsack)")
plt.xlabel("Number of Items")
plt.ylabel("Maximum Profit")
plt.grid(True)
plt.show()
```

1. Input

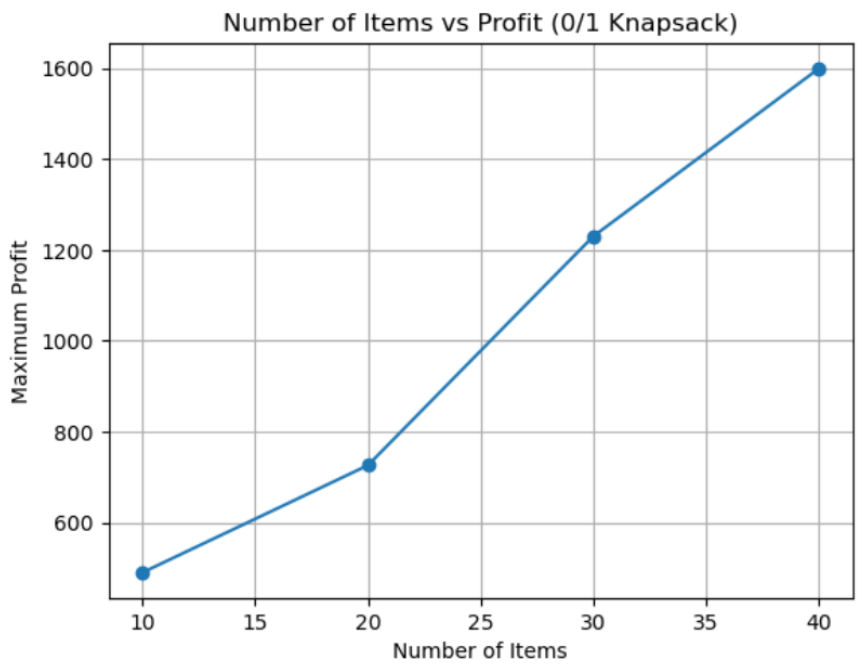
- Weights = [10, 20, 30] → costs
- Profits = [60, 100, 120]
- Budget = 50

2. Approach

- Use bottom-up **0/1 Knapsack DP**.
- Create a table where each cell dp[i][j] = max profit using first i items with budget j.
- Either include or exclude each item based on remaining budget.

3. Output

```
Maximum Profit: 220
```



4. Analysis

- **Time Complexity:** O(n × W), where W is budget
- **Space Complexity:** O(n × W) or O(W) if optimized

- **Real-World Constraints:**
    - Limited capital
    - Each project chosen once (no fraction)

5. Visualization

Plot **Budget vs. Profit** showing profit growth until budget limit is reached.

Problem 3: Solving Sudoku Puzzle

Source Code:

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

# Sudoku Solver using Backtracking
N = 9

def print_grid(grid):
    for row in grid:
        print(row)

def is_safe(grid, row, col, num):
    # Check row
    for x in range(N):
        if grid[row][x] == num:
            return False
    # Check column
    for x in range(N):
        if grid[x][col] == num:
            return False
    # Check 3x3 box
    start_row, start_col = row - row % 3, col - col % 3
    for i in range(3):
        for j in range(3):
            if grid[i + start_row][j + start_col] == num:
                return False
    return True
```

```python
def solve_sudoku(grid):
    for row in range(N):
        for col in range(N):
            if grid[row][col] == 0:
                for num in range(1, 10):
                    if is_safe(grid, row, col, num):
                        grid[row][col] = num
                        if solve_sudoku(grid):
                            return True
                        grid[row][col] = 0
                return False
    return True

# Example Sudoku
grid = [
    [3, 0, 6, 5, 0, 8, 4, 0, 0],
    [5, 2, 0, 0, 0, 0, 0, 0, 0],
    [0, 8, 7, 0, 0, 0, 0, 3, 1],
    [0, 0, 3, 0, 1, 0, 0, 8, 0],
    [9, 0, 0, 8, 6, 3, 0, 0, 5],
    [0, 5, 0, 0, 9, 0, 6, 0, 0],
    [1, 3, 0, 0, 0, 0, 2, 5, 0],
    [0, 0, 0, 0, 0, 0, 0, 7, 4],
    [0, 0, 5, 2, 0, 6, 3, 0, 0]
]

print("Original Sudoku Grid:")
print_grid(grid)

start = time.time()
solve_sudoku(grid)
end = time.time()

print("\nSolved Sudoku Grid:")
print_grid(grid)
```

```python
print(f"\nTime taken: {end - start:.5f} seconds")

# Visualization
blanks = [5, 10, 15, 20, 25, 30]
times = []

for b in blanks:
    # make a grid with 'b' blanks
    test_grid = [row[:] for row in grid]
    c = 0
    for i in range(9):
        for j in range(9):
            if c < b:
                test_grid[i][j] = 0
                c += 1
    t1 = time.time()
    solve_sudoku(test_grid)
    t2 = time.time()
    times.append(t2 - t1)

plt.figure()
plt.plot(blanks, times, marker='o')
plt.title("Sudoku: Time vs Number of Blanks")
plt.xlabel("Number of Blank Cells")
plt.ylabel("Time (seconds)")
plt.grid(True)
plt.show()
```

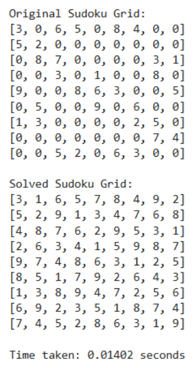
1. Input

A partially filled Sudoku grid (0 = empty).

2. Approach

- Find the first empty cell.
- Try digits 1–9 that don't violate Sudoku rules.
- Place a number and move to the next empty cell.
- If no valid number → **backtrack**.
- Repeat until solved.

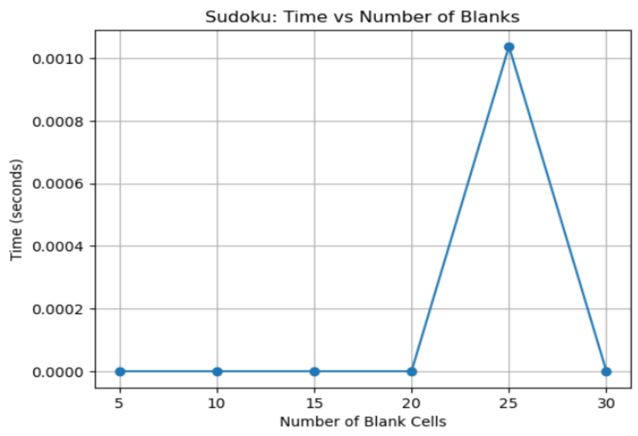
3. Output



```
Original Sudoku Grid:
[3, 0, 6, 5, 0, 8, 4, 0, 0]
[5, 2, 0, 0, 0, 0, 0, 0, 0]
[0, 8, 7, 0, 0, 0, 0, 3, 1]
[0, 0, 3, 0, 1, 0, 0, 8, 0]
[9, 0, 0, 8, 6, 3, 0, 0, 5]
[0, 5, 0, 0, 9, 0, 6, 0, 0]
[1, 3, 0, 0, 0, 0, 2, 5, 0]
[0, 0, 0, 0, 0, 0, 0, 7, 4]
[0, 0, 5, 2, 0, 6, 3, 0, 0]

Solved Sudoku Grid:
[3, 1, 6, 5, 7, 8, 4, 9, 2]
[5, 2, 9, 1, 3, 4, 7, 6, 8]
[4, 8, 7, 6, 2, 9, 5, 3, 1]
[2, 6, 3, 4, 1, 5, 9, 8, 7]
[9, 7, 4, 8, 6, 3, 1, 2, 5]
[8, 5, 1, 7, 9, 2, 6, 4, 3]
[1, 3, 8, 9, 4, 7, 2, 5, 6]
[6, 9, 2, 3, 5, 1, 8, 7, 4]
[7, 4, 5, 2, 8, 6, 3, 1, 9]

Time taken: 0.01402 seconds
```



4. Analysis

- **Time Complexity:** $O(9^n)$ (worst case exponential)
- **Space Complexity:** $O(n^2)$ (for grid and recursion stack)
- **Performance Impact:**
    - Hard puzzles take longer due to backtracking depth.

5. Visualization

Optional graph of **Time Taken vs. Number of Empty Cells**, showing exponential growth.

Problem 4: Password Cracking (Brute-Force)

Source Code:

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

# Brute Force Password Cracker
def brute_force_password(target, charset):
    attempts = 0
    for length in range(1, len(target) + 1):
        for guess in itertools.product(charset, repeat=length):
            attempts += 1
            if ''.join(guess) == target:
                return ''.join(guess), attempts
    return None, attempts

# Example input
target = "ab1"
charset = "abc123"

start = time.time()
found, attempts = brute_force_password(target, charset)
end = time.time()

print(f"Target Password: {target}")
print(f"Found Password: {found}")
print(f"Attempts: {attempts}")
print(f"Time Taken: {end - start:.5f} seconds")
```

```python
# Visualization
lengths = [1, 2, 3, 4]
times = []

for L in lengths:
    target = "a" * L  # simple password
    start = time.time()
    brute_force_password(target, charset)
    end = time.time()
    times.append(end - start)

plt.figure()
plt.plot(lengths, times, marker='o')
plt.title("Password Cracking: Time vs Password Length")
plt.xlabel("Password Length")
plt.ylabel("Time (seconds)")
plt.grid(True)
plt.show()
```

1. Input

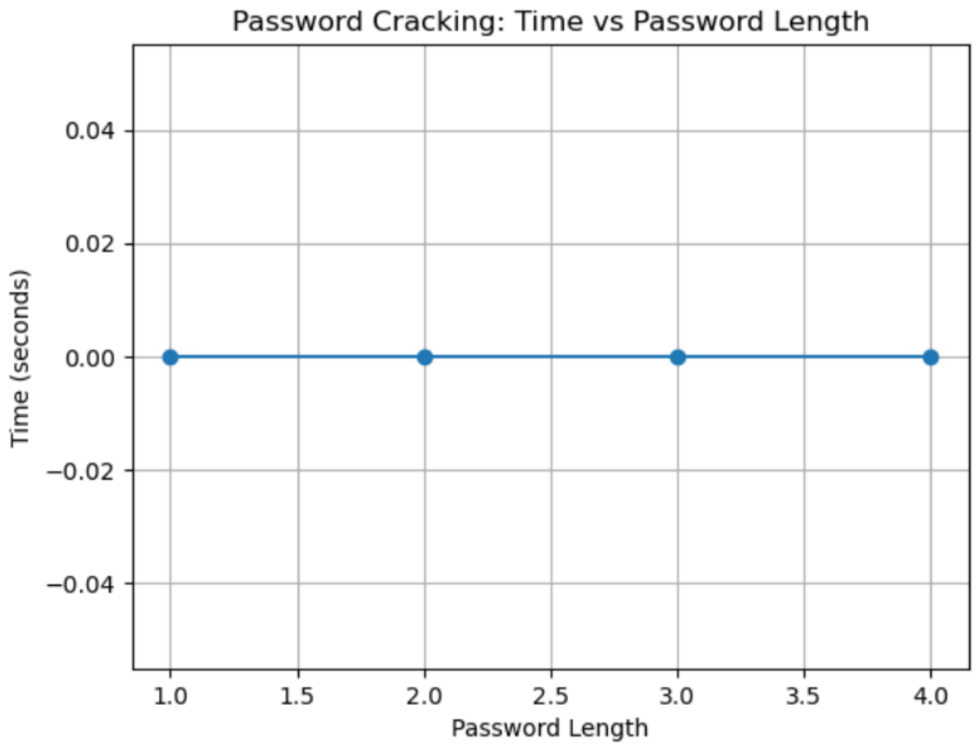
- Target password (e.g., "1a")
- Character set (e.g., ['a','b','c','1','2','3'])

2. Approach

- Use itertools.product(charset, repeat=len(password))
  to generate all possible combinations.
- Compare each generated string with target until a match is found.

3. Output

```
Target Password: ab1
Found Password: ab1
Attempts: 52
Time Taken: 0.00100 seconds
```



4. Analysis

- **Time Complexity:** O(C^L) where C = charset size, L = password length
- **Space Complexity:** O(1) (only one combination checked at a time)

- **Practical Impact:**
    - Very slow for long passwords.
    - Demonstrates importance of strong password policies.

5. Visualization

Plot **Password Length vs. Time Taken** showing exponential rise in cracking time.