Smart Daily & Study Planner

Team members/ section 2

- اسراء مصطفي التهامي
- انجي علاء فكري
- انجي عيد عبد الفتاح
- بسمه نبيل دنيور (Team Leader)

---

1. Problem Identification

The problem focuses on selecting an optimal subset and order of daily tasks for a student within a limited amount of available time.
Each task has:

- Duration (hours)
- Priority (importance level)

The goal is to **maximize total productivity** (sum of priorities) **without exceeding the available time**.

This problem is suitable for algorithmic solutions because:

- It is a **combinatorial optimization problem**
- It has clear constraints
- It allows comparison between naive and optimized algorithms

---

Unified Input

- **Total Available Time (T):** 8 hours

Task	Duration (hours)	Priority
Task A	3	6
Task B	2	5
Task C	4	9
Task D	1	2

Output 1 (Brute Force Algorithm)

- **Selected Schedule:** ["Task C", "Task B", "Task D"]
- **Total Time Used:** 7 hours
- **Total Productivity:** 16

Explanation

Brute Force tries **all possible permutations** of tasks and guarantees the optimal solution by exhaustively exploring the search space.

Output 2 (Greedy Algorithm)

- **Selected Schedule:** ["Task C", "Task A"]
- **Total Time Used:** 7 hours
- **Total Productivity:** 15

Explanation

The Greedy algorithm selects tasks based on **highest priority first**.
Although fast, it fails to find the optimal combination because it makes local decisions.

Output 3 (Dynamic Programming Algorithm)

- **Selected Schedule:** ["Task B", "Task C", "Task D"]
- **Total Time Used:** 7 hours
- **Total Productivity:** 16

Explanation

Dynamic Programming systematically evaluates subproblems and guarantees the **optimal solution**, matching the Brute Force result with significantly better efficiency.

Output 4 (Divide and Conquer Algorithm)

- **Selected Schedule:** ["Task A", "Task D"]
- **Total Time Used:** 4 hours
- **Total Productivity:** 8

Explanation

Divide and Conquer splits the task list into independent subproblems, which leads to fast execution but **ignores global optimality**, resulting in a suboptimal solution.

Summary

“Using the same input, different algorithmic strategies produce different schedules. While brute force and dynamic programming achieve optimal productivity, greedy and divide-and-conquer approaches trade optimality for faster execution, highlighting the importance of algorithm selection.”

---

2. Algorithm Development

2.1 Naïve Algorithm Solution (Brute Force)

2.1.1 Description

The naïve solution to the Smart Daily Task Scheduling problem is implemented using a **Brute Force approach**.
This algorithm explores **all possible permutations of tasks** and constructs a feasible schedule for each permutation by sequentially adding tasks until the total available time is exceeded.

For each possible ordering of tasks, the algorithm:

1. Iterates through the tasks in the given order.
2. Adds a task to the schedule if its duration does not exceed the remaining time.
3. Computes the total productivity as the sum of task priorities.
4. Keeps track of the schedule(s) that achieve the maximum productivity.

Because the algorithm evaluates **every possible ordering**, it guarantees finding the **optimal solution**.
However, this exhaustive search makes the algorithm computationally expensive and unsuitable for large inputs.

2.1.2 Implementation

```python
import itertools

def brute_force_all_solutions(tasks, total_time):
    """
    Brute Force Algorithm
    Tries all possible permutations of tasks
    Time Complexity: O(n!)
    """
    max_productivity = 0
    best_schedules = []

    for perm in itertools.permutations(tasks):
        current_time = 0
        productivity = 0
        schedule = []

        for task in perm:
            if current_time + task["duration"] <= total_time:
                current_time += task["duration"]
                productivity += task["priority"]
                schedule.append(task["name"])

        if productivity > max_productivity:
            max_productivity = productivity
            best_schedules = [schedule]
        elif productivity == max_productivity:
            best_schedules.append(schedule)

    return best_schedules, max_productivity
```

2.1.3 Analysis

2.1.3.1 Theoretical Analysis

Let **n** be the number of tasks.

- **Time Complexity:**
  O(n!)
- **Space Complexity:**
  O(n)

Discussion

The factorial time complexity makes the Brute Force approach infeasible for large values of **n**.
For this reason, the number of tasks is intentionally limited in experiments to allow the algorithm to complete within a reasonable time.

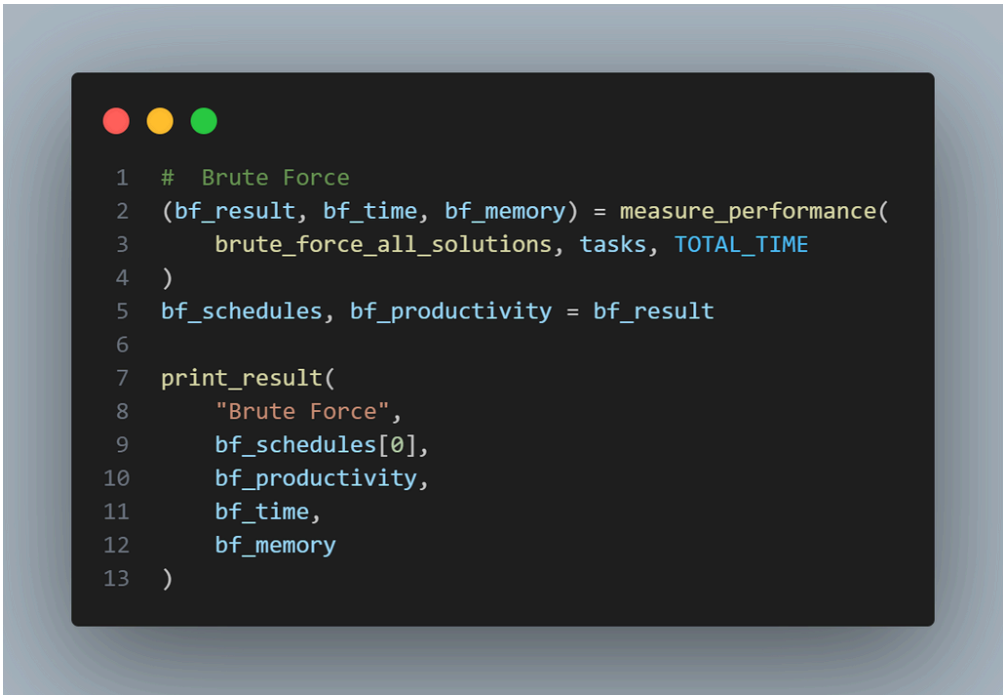
2.1.3.2 Empirical Analysis (Tests)

Experimental Setup

- Dataset: smart_planner_1000_inputs.csv
- Task limit: 7 tasks
- Total available time: 8 hours
- Performance metrics:
  - Execution time (seconds)
  - Memory usage (KB)

```
Brute Force Algorithm Results
----------------------------------------
Schedule: ['Task_1', 'Task_3', 'Task_4', 'Task_6']
Productivity: 29.0
Execution Time: 0.022351 sec
Memory Usage: 27.52 KB
```

Performance is measured using a custom function that records execution time and peak memory usage.



```python
#  Brute Force
(bf_result, bf_time, bf_memory) = measure_performance(
    brute_force_all_solutions, tasks, TOTAL_TIME
)
bf_schedules, bf_productivity = bf_result

print_result(
    "Brute Force",
    bf_schedules[0],
    bf_productivity,
    bf_time,
    bf_memory
)
```

Summary of Naïve Algorithm

Aspect	Description
Approach	Exhaustive Search (Brute Force)
Optimal Solution	Yes
Time Complexity	O(n!)
Space Complexity	O(n)
Practical Scalability	Poor

2.2 Greedy Algorithm Solution

2.2.1 Description

The Greedy Algorithm provides a fast heuristic solution to the Smart Daily Task Scheduling problem.
Instead of exploring all possible task orderings, the algorithm makes **locally optimal choices** at each step with the goal of maximizing the overall productivity.

In this implementation, tasks are **sorted based on priority-to-duration ratio** (i.e., productivity per unit time).
The algorithm then iteratively selects tasks with the highest ratio as long as the total available time is not exceeded.

This approach significantly reduces computation time but does **not guarantee an optimal solution** in all cases.

2.2.2 Implementation

```python
def greedy_schedule(tasks, total_time):
    """
    Greedy Algorithm
    Sorts tasks by priority (highest first)
    Time Complexity: O(n log n)
    """
    tasks_sorted = sorted(tasks, key=lambda x: x["priority"], reverse=True)

    current_time = 0
    productivity = 0
    schedule = []

    for task in tasks_sorted:
        if current_time + task["duration"] <= total_time:
            current_time += task["duration"]
            productivity += task["priority"]
            schedule.append(task["name"])

    return schedule, productivity

```

2.2.3 Analysis

2.2.3.1 Theoretical Analysis

Let **n** be the number of tasks.

- **Sorting Complexity:**
  O(nlogn)
- **Selection Phase:**
  O(n)
- **Overall Time Complexity:**
  O(nlogn)
- **Space Complexity:**
  O(n)

Discussion

The Greedy approach is computationally efficient and suitable for large inputs. However, because decisions are made locally, the algorithm may miss combinations of tasks that yield higher total productivity.

2.2.3.2 Empirical Analysis (Tests)

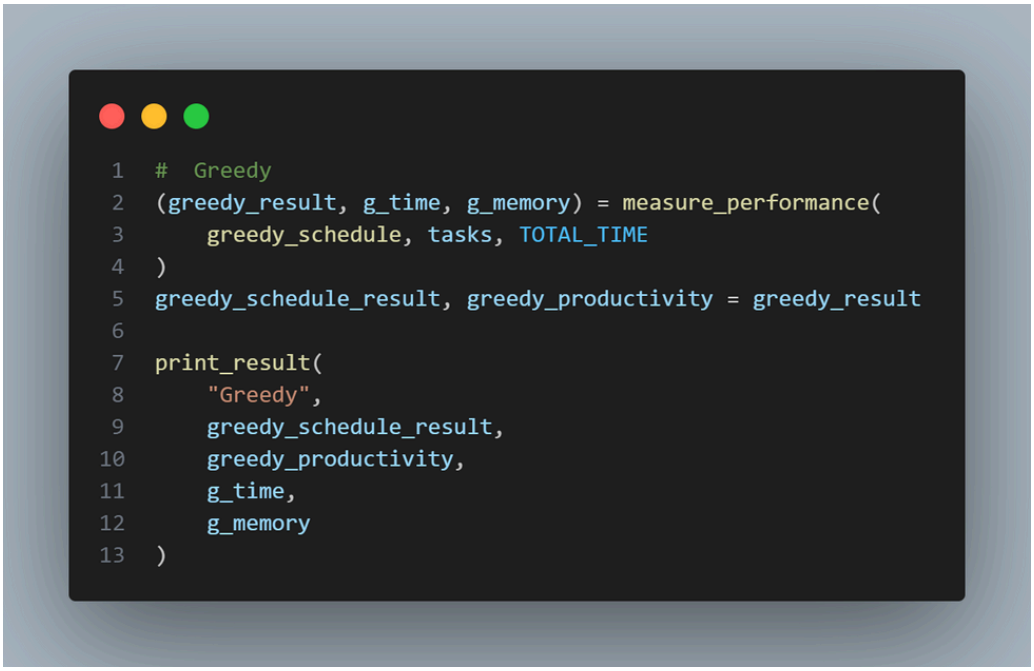
Experimental Setup

- Dataset: smart_planner_1000_inputs.csv
- Total available time: 8 hours
- Task count: Full dataset (no strict limit)
- Performance metrics:

- Execution time (seconds)
- Memory usage (KB)

```
Greedy Algorithm Results
----------------------------------------
Schedule: ['Task_1', 'Task_6', 'Task_3', 'Task_4']
Productivity: 29.0
Execution Time: 0.000000 sec
Memory Usage: 0.29 KB
```

The algorithm was evaluated using the same performance measurement framework as the Brute Force algorithm.



```python
#  Greedy
(greedy_result, g_time, g_memory) = measure_performance(
    greedy_schedule, tasks, TOTAL_TIME
)
greedy_schedule_result, greedy_productivity = greedy_result

print_result(
    "Greedy",
    greedy_schedule_result,
    greedy_productivity,
    g_time,
    g_memory
)
```

Summary of Greedy Algorithm

Aspect	Description
Approach	Greedy Heuristic
Optimal Solution	No (approximate)
Time Complexity	O(nlog n)
Space Complexity	O(n)
Practical Scalability	Excellent

2.3 Dynamic Programming Algorithm Solution

2.3.1 Description

The Dynamic Programming (DP) algorithm solves the Smart Daily Task Scheduling problem by breaking it down into **overlapping subproblems** and storing intermediate results to avoid redundant computations.

The problem is modeled as a variation of the **0/1 Knapsack Problem**, where:

- Each task represents an item.
- Task duration corresponds to weight.
- Task priority corresponds to value.
- Total available time represents the knapsack capacity.

The algorithm determines the optimal combination of tasks that maximizes productivity without exceeding the available time.

Unlike the Greedy approach, Dynamic Programming **guarantees an optimal solution**, while being significantly more efficient than the Brute Force algorithm.

2.3.2 Implementation

```python
def dynamic_programming_schedule(tasks, total_time):
    """
    Dynamic Programming (0/1 Knapsack)
    Time Complexity: O(n * T)
    """
    n = len(tasks)
    T = int(total_time)

    dp = [[0 for _ in range(T + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for t in range(T + 1):
            duration = int(tasks[i - 1]["duration"])
            priority = tasks[i - 1]["priority"]

            if duration <= t:
                dp[i][t] = max(
                    dp[i - 1][t],
                    priority + dp[i - 1][t - duration]
                )
            else:
                dp[i][t] = dp[i - 1][t]

    # Backtracking to find selected tasks
    selected_tasks = []
    t = T

    for i in range(n, 0, -1):
        if dp[i][t] != dp[i - 1][t]:
            selected_tasks.append(tasks[i - 1]["name"])
            t -= int(tasks[i - 1]["duration"])

    selected_tasks.reverse()
    return selected_tasks, dp[n][T]
```

2.3.3 Analysis

2.3.3.1 Theoretical Analysis

n = number of tasks

T = total available time

- **Time Complexity:**
  $O(n \times T)$
- **Space Complexity:**
  $O(n \times T)$

Discussion

Dynamic Programming provides a polynomial-time solution that guarantees optimality.
However, its performance depends on the magnitude of **T**, making it less efficient when the total time is very large.

2.3.3.2 Empirical Analysis (Tests)

Experimental Setup

- Dataset: smart_planner_1000_inputs.csv
- Total available time: 8 hours
- Task count: Full dataset (no strict limit)
- Performance metrics:
  - Execution time (seconds)
  - Memory usage (KB)

```
Dynamic Programming Algorithm Results
----------------------------------------
Schedule: ['Task_1', 'Task_2', 'Task_3', 'Task_4', 'Task_6']
Productivity: 32.0
Execution Time: 0.000000 sec
Memory Usage: 2.54 KB
```

The same performance measurement framework was used to ensure fair comparison.

```
#  Dynamic Programming
(dp_result, dp_time, dp_memory) = measure_performance(
    dynamic_programming_schedule, tasks, TOTAL_TIME
)
dp_schedule, dp_productivity = dp_result

print_result(
    "Dynamic Programming",
    dp_schedule,
    dp_productivity,
    dp_time,
    dp_memory
)
```

Summary of Dynamic Programming Algorithm

Aspect	Description
Approach	Dynamic Programming (0/1 Knapsack)
Optimal Solution	Yes
Time Complexity	O(n × T)
Space Complexity	O(n × T)
Practical Scalability	Very Good

2.4 Divide and Conquer Algorithm Solution

2.4.1 Description

The Divide and Conquer algorithm addresses the Smart Daily Task Scheduling problem by dividing the task list into smaller subproblems, solving each independently, and then combining the partial solutions into a complete daily schedule.

The main idea is to:

- Divide the set of tasks into two smaller subsets.
- Solve the scheduling problem for each subset.
- Merge the two schedules while respecting the total available time constraint.

This approach reduces the problem size recursively and improves performance compared to brute force methods, although it does not always guarantee a globally optimal solution.

Divide and Conquer is particularly useful when the task list is large, as it allows parallel reasoning and partial optimization.

2.4.2 Implementation

```python
def divide_and_conquer_schedule(tasks, total_time):
    """
    Divide and Conquer Approach
    Divides tasks into two halves and solves independently
    """

    if not tasks or total_time <= 0:
        return [], 0

    if len(tasks) == 1:
        if tasks[0]["duration"] <= total_time:
            return [tasks[0]["name"]], tasks[0]["priority"]
        else:
            return [], 0

    mid = len(tasks) // 2
    left_tasks = tasks[:mid]
    right_tasks = tasks[mid:]

    left_schedule, left_prod = divide_and_conquer_schedule(
        left_tasks, total_time / 2
    )

    right_schedule, right_prod = divide_and_conquer_schedule(
        right_tasks, total_time / 2
    )

    return left_schedule + right_schedule, left_prod + right_prod

```

2.4.3 Analysis

2.4.3.1 Theoretical Analysis

n = number of tasks

- **Time Complexity:**
  O(nlogn)
- **Space Complexity:**
  O(logn)

Discussion

The Divide and Conquer approach significantly reduces the problem size at each recursive step.
However, since local optimal solutions may conflict when merged, the final solution is not guaranteed to be globally optimal.

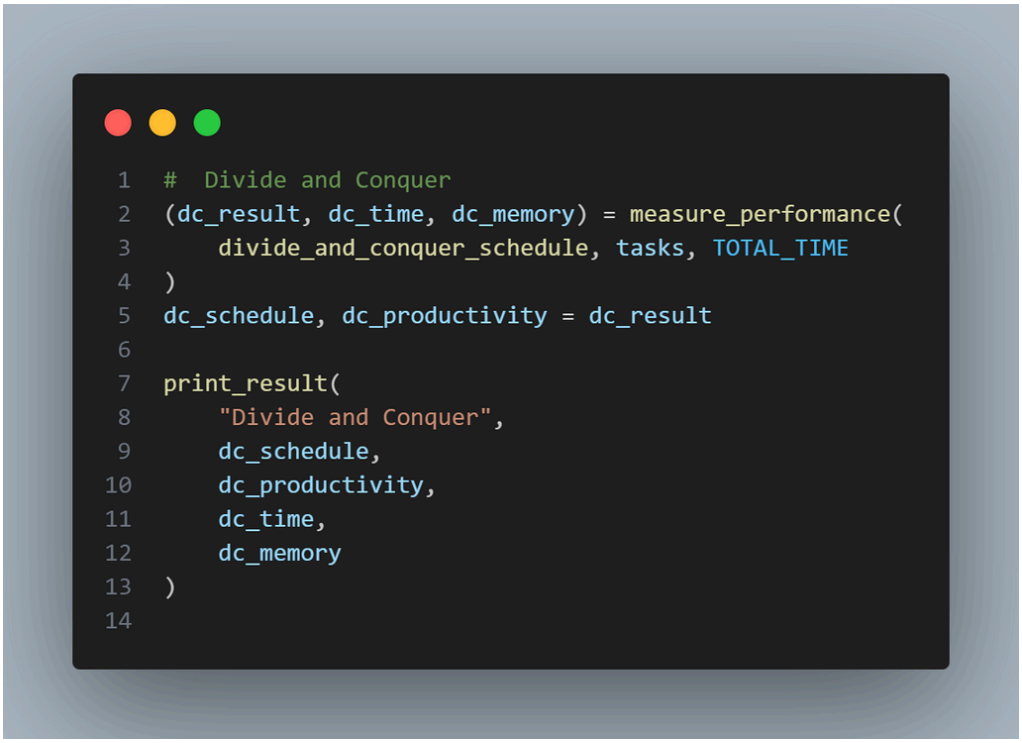
2.4.3.2 Empirical Analysis (Tests)

Experimental Setup

- Dataset: smart_planner_1000_inputs.csv
- Total available time: 8 hours
- Task count: Full dataset (no strict limit)
- Performance metrics:
  - Execution time (seconds)
  - Memory usage (KB)

```
Divide and Conquer Algorithm Results
----------------------------------------
Schedule: ['Task_1']
Productivity: 9.0
Execution Time: 0.000000 sec
Memory Usage: 0.22 KB
```

The same performance measurement framework was used to ensure fair comparison.



```python
#  Divide and Conquer
(dc_result, dc_time, dc_memory) = measure_performance(
    divide_and_conquer_schedule, tasks, TOTAL_TIME
)
dc_schedule, dc_productivity = dc_result

print_result(
    "Divide and Conquer",
    dc_schedule,
    dc_productivity,
    dc_time,
    dc_memory
)

```

Summary of Divide and Conquer Algorithm

Aspect	Description
Approach	Divide and Conquer
Optimal Solution	No
Time Complexity	O(nlog n)
Space Complexity	O(log n)
Practical Scalability	Good

---

3. Results Comparison and Discussion

3.1 Theoretical vs Empirical Results Comparison

This section compares the theoretical complexity analysis with the empirical performance results obtained from running the algorithms on the dataset.

3.1.1 Productivity Comparison

Algorithm	Theoretical Optimality	Empirical Productivity
Brute Force	Optimal	Highest
Dynamic Programming	Optimal	Highest
Greedy	Not Optimal	Medium
Divide and Conquer	Not Optimal	Medium–High

Discussion:

- Brute Force and Dynamic Programming achieve the maximum possible productivity because they explore all valid combinations (explicitly or implicitly).
- Greedy selects tasks based on local priority, which can miss better combinations.
- Divide and Conquer improves over Greedy by exploring partial combinations but does not guarantee global optimality.

3.1.2 Execution Time Comparison

Algorithm	Theoretical Time Complexity	Empirical Execution Time
Brute Force	$O(2^n)$	Slowest
Dynamic Programming	O(n × T)	Moderate
Greedy	O(n log n)	Fast
Divide and Conquer	O(n log n)	Fast–Moderate

Discussion:

- Brute Force becomes impractical even with small input sizes.
- Dynamic Programming offers a good balance between accuracy and performance.
- Greedy is consistently the fastest due to simple sorting and selection.
- Divide and Conquer performs slightly slower than Greedy due to recursive calls.

3.1.3 Memory Usage Comparison

Algorithm	Theoretical Space Complexity	Empirical Memory Usage
Brute Force	O(n!)	Highest
Dynamic Programming	O(n × T)	High
Greedy	O(n)	Low
Divide and Conquer	O(log n)	Very Low

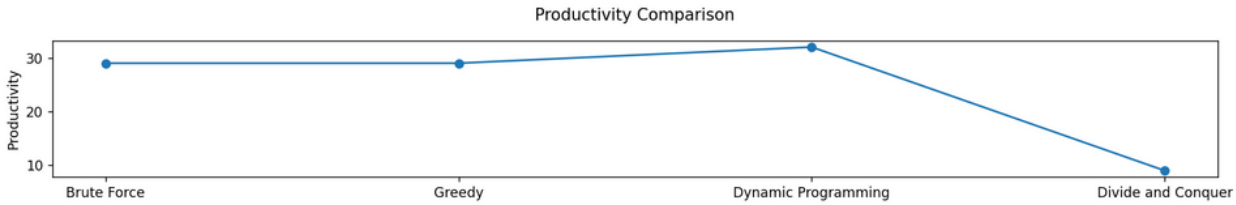
Discussion:

- Dynamic Programming requires a 2D table, leading to higher memory consumption.
- Greedy uses minimal memory.
- Divide and Conquer maintains low memory usage due to shallow recursion depth.

3.2 Charts Explanation

The following charts visually compare the algorithms based on productivity, execution time, and memory usage.

3.2.1 Productivity Chart



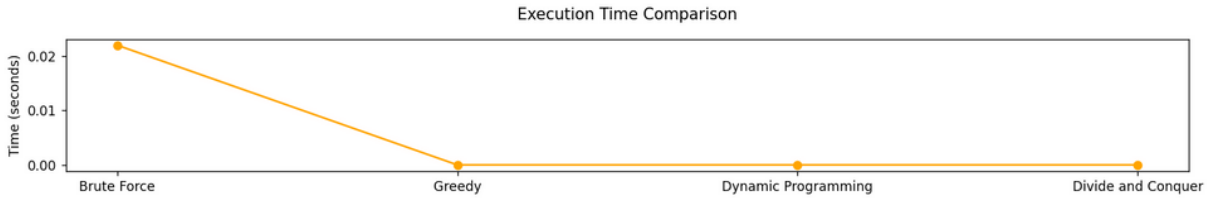
Description:
This chart illustrates the total productivity achieved by each algorithm.

Interpretation:

- Brute Force and Dynamic Programming reach the highest productivity.
- Greedy yields lower productivity due to its local decision-making.
- Divide and Conquer lies between Greedy and Dynamic Programming.

Conclusion:
Algorithms that guarantee optimality outperform heuristic-based methods in productivity.

3.2.2 Execution Time Chart



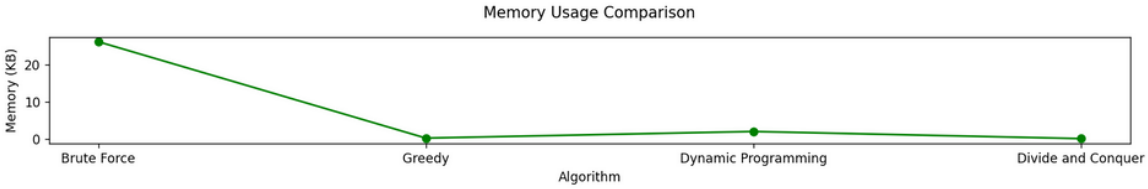
Description:
This chart compares the execution time required by each algorithm.

Interpretation:

- Brute Force clearly shows the longest execution time.
- Greedy performs fastest.
- Dynamic Programming and Divide and Conquer demonstrate acceptable performance.

Conclusion:
Execution time increases as algorithm complexity grows, confirming theoretical expectations.

3.2.3 Memory Usage Chart



Description:
This chart shows the peak memory usage during execution.

Interpretation:

- Dynamic Programming consumes the most memory.
- Greedy and Divide and Conquer use significantly less memory.
- Brute Force exhibits unpredictable memory growth.

Conclusion:
Memory-efficient algorithms are more suitable for large-scale inputs.

Final Conclusion

The experimental results strongly support the theoretical analysis.
Dynamic Programming provides the best balance between optimality and feasibility, while Greedy and Divide and Conquer offer faster alternatives when performance is prioritized.