ECM1414 – Data Structures and Algorithms – Group Project for Event Planner

1. Introduction:

This report presents our design and implementation of the Student Society Event Planner system. The planner addresses the problem of selecting an optimal subset of activities for a society's event weekend under limited resources.

Each activity is characterized by:

- Duration (hours)
- Cost (£)
- Estimated enjoyment value

Given fixed constraints on time and budget, the objective is to determine the right combination of activities that maximise total enjoyment but also ensure that that said constraints are not exceeded. Since each activity may be selected at most once, this problem can be modelled as a constrained optimisation task which requires careful decision-making under limited resources.

The objective of this coursework is to design, implement, and compare two algorithmic strategies: a naïve brute-force exhaustive search and a dynamic approach. While brute force guarantees an optimal solution by evaluating all possible subsets, it's expected to become computationally expensive as the number of activities increases. In contrast, the dynamic programming approach is hypothesised to achieve the same optimal result more efficiently by avoiding redundant computation.

By analysing their correctness and performance across varying input sizes, we aim to highlight the differences in efficiency, scalability, and practicality.

Moreover, optional extensions were implemented to further enhance the planner and explore additional improvements to the algorithms.

2. Design and Approach:

2.1 Program Structure

The program is organised into separate modules to ensure a clear separation between inputs, data representation, and algorithmic implementation. This modular structure improves maintainability, simplifies independent testing, and allows for different algorithms to be compared under identical input conditions.

- **Event_planner.py**
  This serves as the main driver of the program: it reads input files, constructs the list of Activity objects, passes the activities and constraints to the selected planner function, and prints the resulting solution. By centralising these responsibilities, the program can ensure that both algorithms can be evaluated consistently.
- **Brute_force.py**
  This contains the baseline exhaustive search algorithm. This module generates and evaluates combinations of activities to return the best feasible solution found.
- **Enhanced_brute_force.py (Extension 1)**
  Extends said baseline brute-force approach to check both time and budget constraints simultaneously. This was separated from the baseline to preserve a clean comparison between core functions and optional extended ones.
- **Dynamic.py**
  Contains the dynamic programming implementation. This module computes the optimal solution using a more efficient strategy than exhaustive search. Separating it ensures clear modularity and allows for core logic to be isolated for independent testing, debugging, and maintenance.
- **Enhanced_dynamic.py (Extension 1)**
  Extends the baseline dynamic programming approach to support both constraints simultaneously. Unlike the single constraint version, this maintains a two-dimensional state representation to ensure that selected activities satisfy both limits. Separating this module from the baseline dynamic implementation preserves clarity, enables direct comparison between single- and multi- constraint versions, and keeps optional extensions isolated from core functionality.
- **Input_Files**
  Contains provided and additional test inputs of varying sizes. These are used to evaluate correctness and observe performance as the number of activities increases.

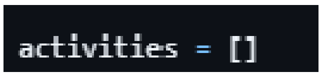
This modular structure allows code and algorithms to be modified, replaced, and extended without the need to alter how inputs are processed or represented.

2.2 Data Structures Used

This implementation makes use of the following core data structures:

Traditional Data Structures

1. **Lists:**

   ```
   activities = []
   ```

   Lists serve as the primary data structure in the implementation. Storing all activities in a Python list preserves input order and enables efficient sequential traversal. This is important because the algorithms depend on processing activities through a consistent and predictable sequence. Moreover, lists support direct indexing, which is essential for generating and evaluating subsets in the brute-force algorithm.
2. **Tuples**
   In the brute-force implementation, subsets of activities are represented as tuples (generated through itertools.combinations). Tuples are immutable, making them appropriate for representing fixed combinations of activity indices. The final selected solution is also returned as a tuple of activity indices to ensure the solution is stable and cannot be accidentally modified.
3. **2D dynamic programming table**
   For the multi-constraint version, a two-dimensional DP table is used to track optimal enjoyment values across both time and budget capacities.
   Each state represents the best achievable enjoyment for a given combination of constraints. Thus, this table enables systematic optimisation while also avoiding redundant recalculation of subproblems.
   Although this increases memory usage, it preserves polynomial-time complexity and remains significantly more efficient than exhaustive searches.

Less Traditional / Implementation-Specific Structures

4. **Snapshot storage**
   During dynamic programming, intermediate states (snapshots of best achievable enjoyment values) are stored within the DP table. This avoids redundant recalculation of overlapping subproblems and ensure that previously computed results can be reused efficiently.
5. **Activity Objects (Class instances)**
   Each activity is represented as an instance of an 'Activity' class containing: cost, duration, enjoyment level, name. Encapsulating these attributes within a class improves modularity and reduces indexing errors. This 'object-oriented structure' also makes the program easier to extend.

2.3 Assumptions and Simplifications

The implementation makes the following assumptions:

- Each activity may be selected at most once.
- Total enjoyment is calculated as the sum of individual enjoyment values
- Duration, cost, and enjoyment values are treated as non-negative integers
- Constraints are treated as strict limits – no exceeding.
- The baseline algorithms focus on budget while the enhanced ones check both time and budget.

These simplifications keep the problem well-defined and allow the focus to remain on algorithmic correctness, efficiency, and scalability.

3. Algorithms:

Documentation requirement for Brute Force algorithm:

This brute force function explores every possible subset of activities whose total cost fits within the budget. The algorithm creates a list of integers 0, 1, 2, ..., n-1 where n is the number of activities. These IDs are used to index into the activities list. By computing: max_length = budget/ cheapest activity cost, an upper bound on how many activities could possibly fit within the budget can be found. If this exceeds the number of activities, then it is capped at n. By doing this, the algorithm avoids generating combinations that are guaranteed to exceed the budget.

Combinations are generated in the following for loop:

for length in range(max_length, -1, -1):

possible_solutions = combinations(ids, length)

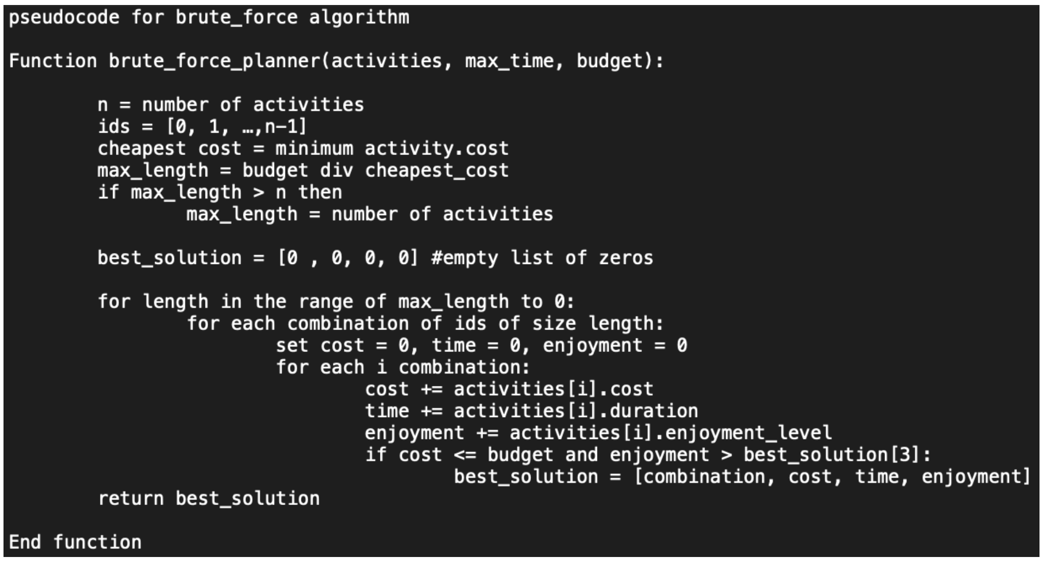
For each length, itertools.combinations generates all subsets of that size. For example, if length = 3 and you gave 6 activities, you generate 20 combinations (6C3 = 20). Each combination is a tuple like (0, 2,5)

For every combination, the cost, time and enjoyment are summed then the algorithm checks if cost <= budget, then it's a valid holiday plan. If its enjoyment is better than the current best, store it. Essentially, they can be broken down into 3 simple steps: generate everything → evaluate everything → keep the best.

Time and space complexity of the algorithm

The time complexity grows exponentially because the number of possible subsets of n items is 2^n even though the algorithm restricts the subset sizes, the number of combinations still grows exponentially. Each activity has two possible states (included or excluded, resulting in 2^n possible subsets. This exponential growth explains the rapid increase in runtime as n increase.

Pseudocode for brute force algorithm:



```
pseudocode for brute_force algorithm

Function brute_force_planner(activities, max_time, budget):

        n = number of activities
        ids = [0, 1, …,n-1]
        cheapest cost = minimum activity.cost
        max_length = budget div cheapest_cost
        if max_length > n then
                max_length = number of activities

        best_solution = [0 , 0, 0, 0] #empty list of zeros

        for length in the range of max_length to 0:
                for each combination of ids of size length:
                        set cost = 0, time = 0, enjoyment = 0
                        for each i combination:
                                cost += activities[i].cost
                                time += activities[i].duration
                                enjoyment += activities[i].enjoyment_level
                                if cost <= budget and enjoyment > best_solution[3]:
                                        best_solution = [combination, cost, time, enjoyment]
        return best_solution

End function
```

Documentation for Dynamic programming

Logic and approach

The algorithm creates a table where each row represents a possible amount of time you could spend (from 0 up to max_time) and each column represents a possible amount of money you could spend (from 0 up to budget). At the start, the table is full of zeros because no activities have been considered yet.

For each activity, the algorithm checks if the activity improves the enjoyment for any time and budget combination. The algorithm performs this check by looking at the table backwards (so it doesn't accidentally reuse the same activity twice). For each cell (time,budget), algorithm checks whether you could fit this activity into that time and budget. If yes, it calculates what the enjoyment would be if you included it and if that enjoyment is better than what's currently in the bale, it updates the cell.

Before processing each activity, the algorithm takes a snapshot of the table. This is done because later, when the algorithm wants to reconstruct which activities were chosen, it needs to know whether adding activity "i" changed the table or not. These snapshots let the algorithm trace backwards and figure out exactly which activities contributed to the final best enjoyment.

Once all activities have been processed, the table contains the best enjoyment values for every possible time and budget combination. The algorithm scans the whole table to find the cell with the highest enjoyment. That cell tells you: how much time was used; how much money was used and what the maximum enjoyment is.

The algorithm works backwards to figure out which activities were chosen and this is where the snapshots are used. Starting from the best cell, the algorithm goes backwards through the activities: it compares the current table with the snapshot from before the activity was added, If the value in the cell changed that means the activity was included, if the vale stayed the same the activity was not included. Each time it finds and included activity, it subtracts the activity's time and cost and continues tracing back. By the end, the algorithm has reconstructed the exact set of activities that produced the optimal enjoyment.

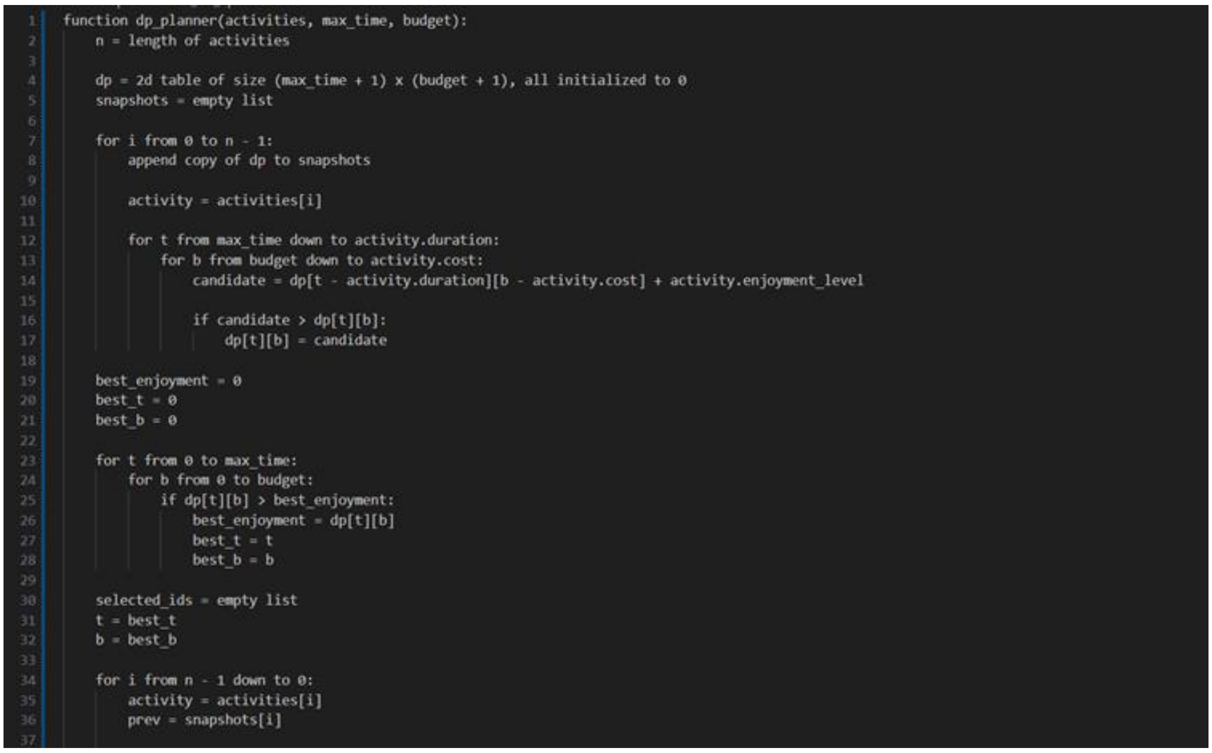
Time complexity

The algorithm fills in a 2D table where: one dimension is all possible time values (0 → max_time), the other is all possible budget values (0 → budget). For each activity, it scans through the entire table (backwards) and updates cells where the activity fits. This means that the work done grows with: the number of activities, multiplied by all possible time values, multiplied by all possible budget values. This gives this dynamic algorithm a polynomial time complexity.

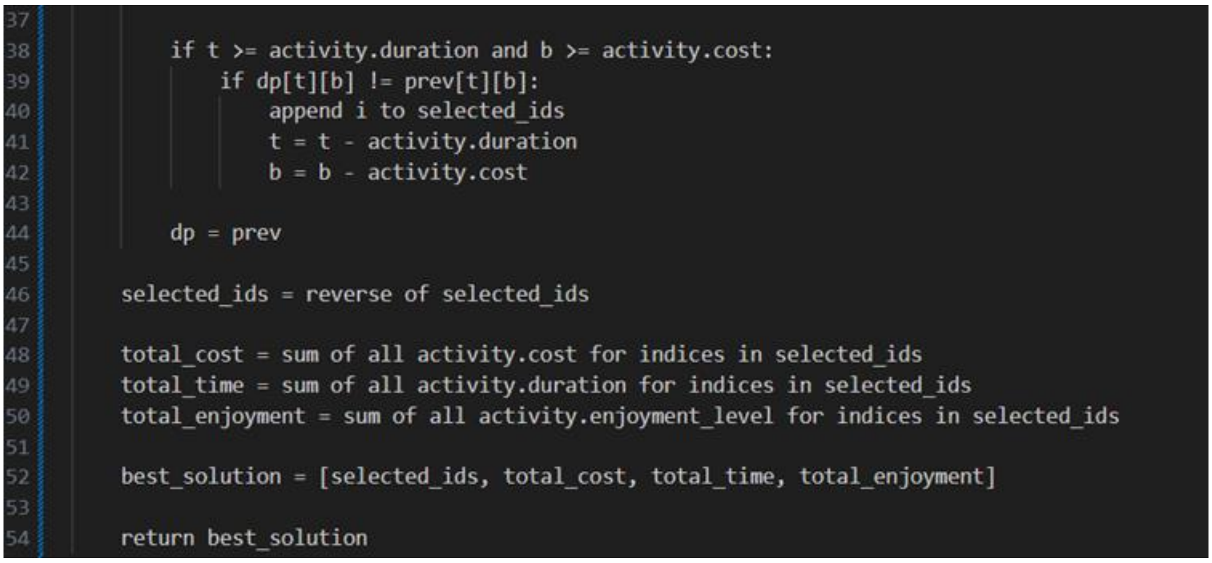
Space complexity

The algorithm stores a large table with size (max_time + 1) * (budget + 1). This table requires a lot of memory to store but the algorithm also stores a snapshot of the entire table before each activity is processed. So, if there are n activities then there are n copies of the table. This means the memory grows with n * max_time * budget. This gives the algorithm a polynomial space complexity.

Pseudocode for Dynamic Programming:



```
function dp_planner(activities, max_time, budget):
    n = length of activities

    dp = 2d table of size (max_time + 1) x (budget + 1), all initialized to 0
    snapshots = empty list

    for i from 0 to n - 1:
        append copy of dp to snapshots

        activity = activities[i]

        for t from max_time down to activity.duration:
            for b from budget down to activity.cost:
                candidate = dp[t - activity.duration][b - activity.cost] + activity.enjoyment_level

                if candidate > dp[t][b]:
                    dp[t][b] = candidate

    best_enjoyment = 0
    best_t = 0
    best_b = 0

    for t from 0 to max_time:
        for b from 0 to budget:
            if dp[t][b] > best_enjoyment:
                best_enjoyment = dp[t][b]
                best_t = t
                best_b = b

    selected_ids = empty list
    t = best_t
    b = best_b

    for i from n - 1 down to 0:
        activity = activities[i]
        prev = snapshots[i]

```



```
        if t >= activity.duration and b >= activity.cost:
            if dp[t][b] != prev[t][b]:
                append i to selected_ids
                t = t - activity.duration
                b = b - activity.cost

        dp = prev

    selected_ids = reverse of selected_ids

    total_cost = sum of all activity.cost for indices in selected_ids
    total_time = sum of all activity.duration for indices in selected_ids
    total_enjoyment = sum of all activity.enjoyment_level for indices in selected_ids

    best_solution = [selected_ids, total_cost, total_time, total_enjoyment]

    return best_solution
```

Testing and Results:

Constraints for all tests

Test	Time	Budget
Small	10 hours	£200
Medium	15 hours	£300
Large	20 hours	£500

Introduction:

For testing we tested the program by using the provided small, medium and large txt files and we used the small file to verify the program was working as intended. As the small file only contains 5 activities there is only 32 possible combinations, meaning we can manually verify which is best.

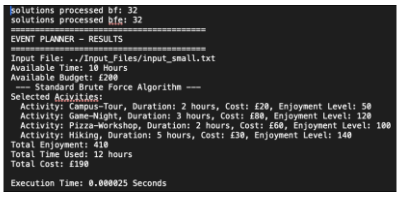
Once we know the algorithms are correct for the smaller file size it means that if they run and go through the correct number of expected combinations then the final output should be correct, presuming the solution provided in the output for the larger sizes doesn't exceed the constraint(s) for that algorithm.

To find out the number of combinations that are processed we added a temporary count variable which increments by 1 every time a solution is processed.

Brute Force:

Small Test:

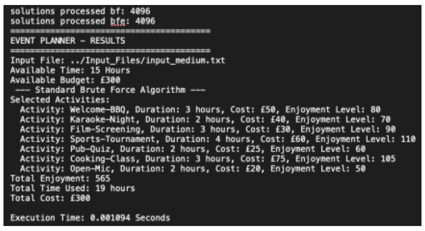
- 32 solutions processed, as expected
- Within constraint, budget, as expected
- Selected expected combination



```
solutions processed bf: 32
solutions processed bfe: 32
========================================
EVENT PLANNER - RESULTS
========================================
Input File: ../Input_Files/input_small.txt
Available Time: 10 Hours
Available Budget: £200
 --- Standard Brute Force Algorithm ---
Selected Acivities:
  Activity: Campus-Tour, Duration: 2 hours, Cost: £20, Enjoyment Level: 50
  Activity: Game-Night, Duration: 3 hours, Cost: £80, Enjoyment Level: 120
  Activity: Pizza-Workshop, Duration: 2 hours, Cost: £60, Enjoyment Level: 100
  Activity: Hiking, Duration: 5 hours, Cost: £30, Enjoyment Level: 140
Total Enjoyment: 410
Total Time Used: 12 hours
Total Cost: £190

Execution Time: 0.000025 Seconds
```

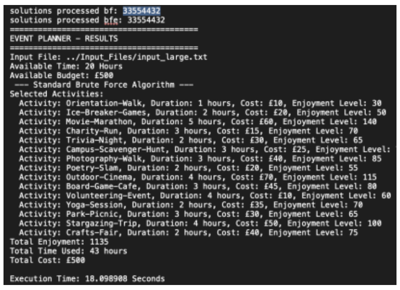
Medium and Large Tests:

- 4096 solutions processed, as expected
- Within constraint, budget, as expected



```
solutions processed bf: 4096
solutions processed bfe: 4096
========================================
EVENT PLANNER - RESULTS
========================================
Input File: ../Input_Files/input_medium.txt
Available Time: 15 Hours
Available Budget: £300
 --- Standard Brute Force Algorithm ---
Selected Activities:
  Activity: Welcome-BBQ, Duration: 3 hours, Cost: £50, Enjoyment Level: 80
  Activity: Karaoke-Night, Duration: 2 hours, Cost: £40, Enjoyment Level: 70
  Activity: Film-Screening, Duration: 3 hours, Cost: £30, Enjoyment Level: 90
  Activity: Sports-Tournament, Duration: 4 hours, Cost: £60, Enjoyment Level: 110
  Activity: Pub-Quiz, Duration: 2 hours, Cost: £25, Enjoyment Level: 60
  Activity: Cooking-Class, Duration: 3 hours, Cost: £75, Enjoyment Level: 105
  Activity: Open-Mic, Duration: 2 hours, Cost: £20, Enjoyment Level: 50
Total Enjoyment: 565
Total Time Used: 19 hours
Total Cost: £300

Execution Time: 0.001094 Seconds
```

- 33,554,432 solutions processed, as expected
- Within constraint, budget, as expected

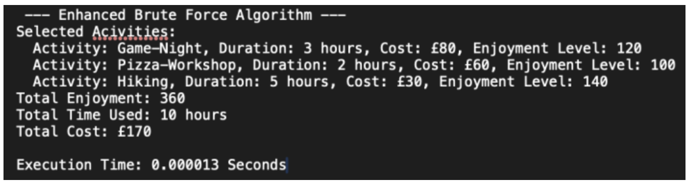


```
solutions processed bf: 33554432
solutions processed bfe: 33554432
========================================
EVENT PLANNER - RESULTS
========================================
Input File: ../Input_Files/input_large.txt
Available Time: 20 Hours
Available Budget: £500
 --- Standard Brute Force Algorithm ---
Selected Activities:
  Activity: Orientation-Walk, Duration: 1 hours, Cost: £10, Enjoyment Level: 30
  Activity: Ice-Breaker-Games, Duration: 2 hours, Cost: £20, Enjoyment Level: 50
  Activity: Movie-Marathon, Duration: 5 hours, Cost: £60, Enjoyment Level: 140
  Activity: Charity-Run, Duration: 3 hours, Cost: £15, Enjoyment Level: 70
  Activity: Trivia-Night, Duration: 2 hours, Cost: £30, Enjoyment Level: 65
  Activity: Campus-Scavenger-Hunt, Duration: 3 hours, Cost: £25, Enjoyment Level:
  Activity: Photography-Walk, Duration: 3 hours, Cost: £40, Enjoyment Level: 85
  Activity: Poetry-Slam, Duration: 2 hours, Cost: £20, Enjoyment Level: 55
  Activity: Outdoor-Cinema, Duration: 4 hours, Cost: £70, Enjoyment Level: 115
  Activity: Board-Game-Cafe, Duration: 3 hours, Cost: £45, Enjoyment Level: 80
  Activity: Volunteering-Event, Duration: 4 hours, Cost: £10, Enjoyment Level: 60
  Activity: Yoga-Session, Duration: 2 hours, Cost: £35, Enjoyment Level: 70
  Activity: Park-Picnic, Duration: 3 hours, Cost: £30, Enjoyment Level: 65
  Activity: Stargazing-Trip, Duration: 4 hours, Cost: £50, Enjoyment Level: 100
  Activity: Crafts-Fair, Duration: 2 hours, Cost: £40, Enjoyment Level: 75
Total Enjoyment: 1135
Total Time Used: 43 hours
Total Cost: £500

Execution Time: 18.098908 Seconds
```

Dual Constraint Brute Force:

Small Test:

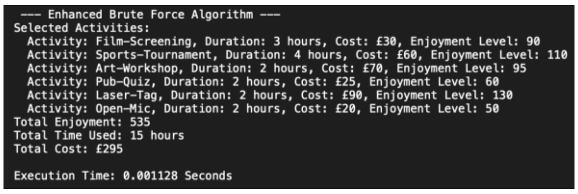
- 32 solutions processed, as expected (see above screenshot)
- Within both Budget and Time Constraints, as expected
- Selected the expected combination



```
 --- Enhanced Brute Force Algorithm ---
Selected Acivities:
  Activity: Game-Night, Duration: 3 hours, Cost: £80, Enjoyment Level: 120
  Activity: Pizza-Workshop, Duration: 2 hours, Cost: £60, Enjoyment Level: 100
  Activity: Hiking, Duration: 5 hours, Cost: £30, Enjoyment Level: 140
Total Enjoyment: 360
Total Time Used: 10 hours
Total Cost: £170

Execution Time: 0.000013 Seconds
```

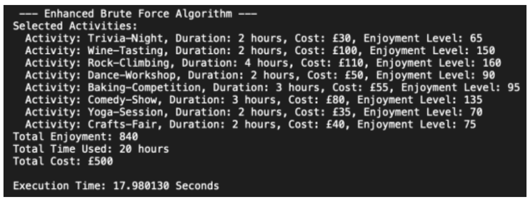
Medium and Large Tests:

- 4096 solutions processed, as expected
- Within constraint, budget, as expected



```
 --- Enhanced Brute Force Algorithm ---
Selected Activities:
  Activity: Film-Screening, Duration: 3 hours, Cost: £30, Enjoyment Level: 90
  Activity: Sports-Tournament, Duration: 4 hours, Cost: £60, Enjoyment Level: 110
  Activity: Art-Workshop, Duration: 2 hours, Cost: £70, Enjoyment Level: 95
  Activity: Pub-Quiz, Duration: 2 hours, Cost: £25, Enjoyment Level: 60
  Activity: Laser-Tag, Duration: 2 hours, Cost: £90, Enjoyment Level: 130
  Activity: Open-Mic, Duration: 2 hours, Cost: £20, Enjoyment Level: 50
Total Enjoyment: 535
Total Time Used: 15 hours
Total Cost: £295

Execution Time: 0.001128 Seconds
```

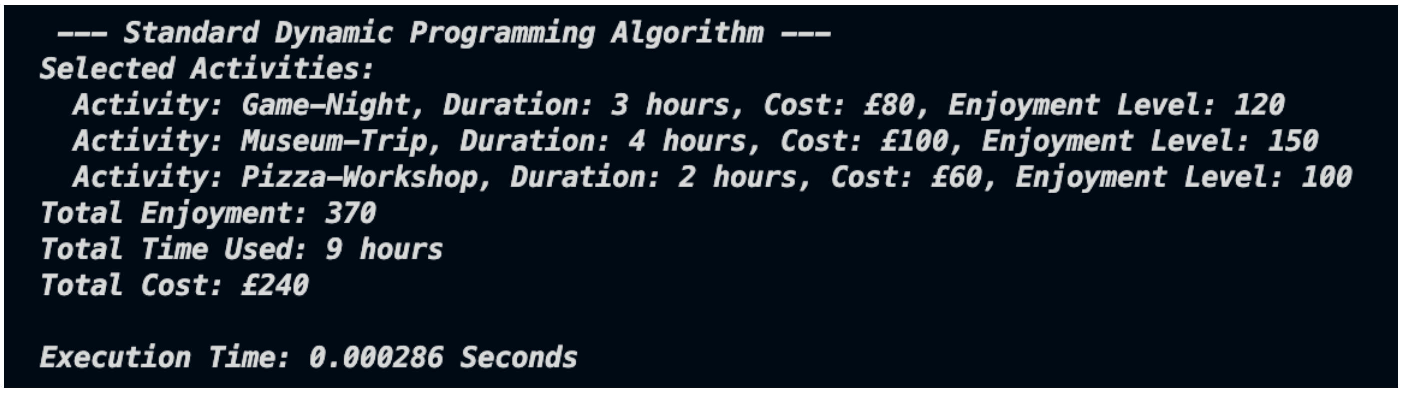
- 33,554,432 solutions processed, as expected
- Within constraint, budget, as expected



```
 --- Enhanced Brute Force Algorithm ---
Selected Activities:
  Activity: Trivia-Night, Duration: 2 hours, Cost: £30, Enjoyment Level: 65
  Activity: Wine-Tasting, Duration: 2 hours, Cost: £100, Enjoyment Level: 150
  Activity: Rock-Climbing, Duration: 4 hours, Cost: £110, Enjoyment Level: 160
  Activity: Dance-Workshop, Duration: 2 hours, Cost: £50, Enjoyment Level: 90
  Activity: Baking-Competition, Duration: 3 hours, Cost: £55, Enjoyment Level: 95
  Activity: Comedy-Show, Duration: 3 hours, Cost: £80, Enjoyment Level: 135
  Activity: Yoga-Session, Duration: 2 hours, Cost: £35, Enjoyment Level: 70
  Activity: Crafts-Fair, Duration: 2 hours, Cost: £40, Enjoyment Level: 75
Total Enjoyment: 840
Total Time Used: 20 hours
Total Cost: £500

Execution Time: 17.980130 Seconds
```

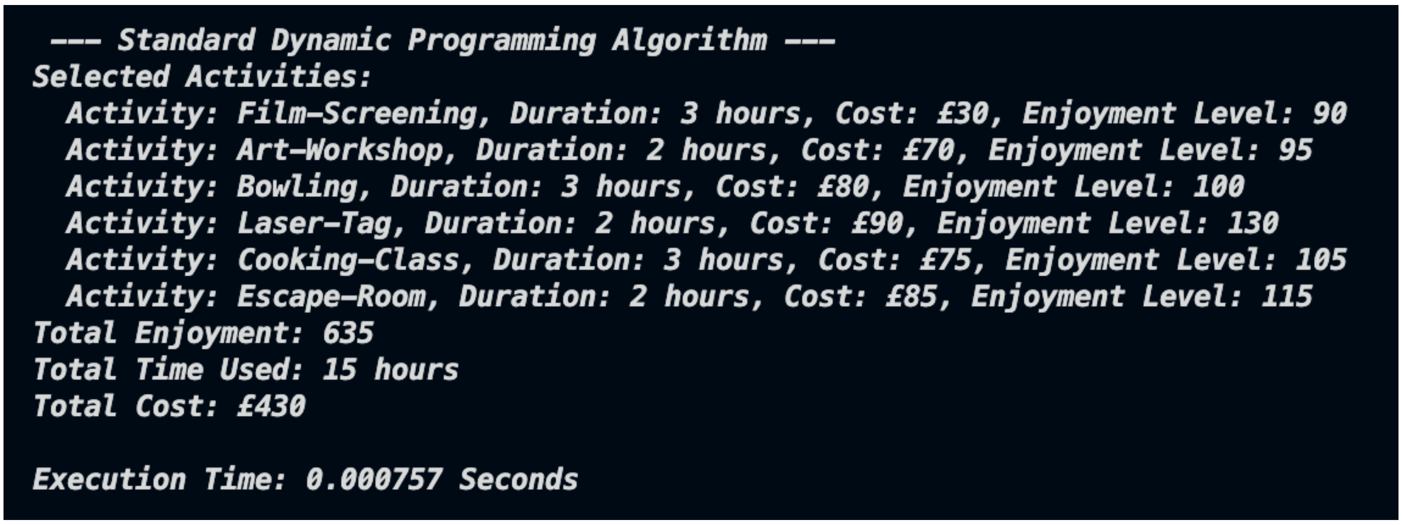
Dynamic Programming Implementation

Small Test:

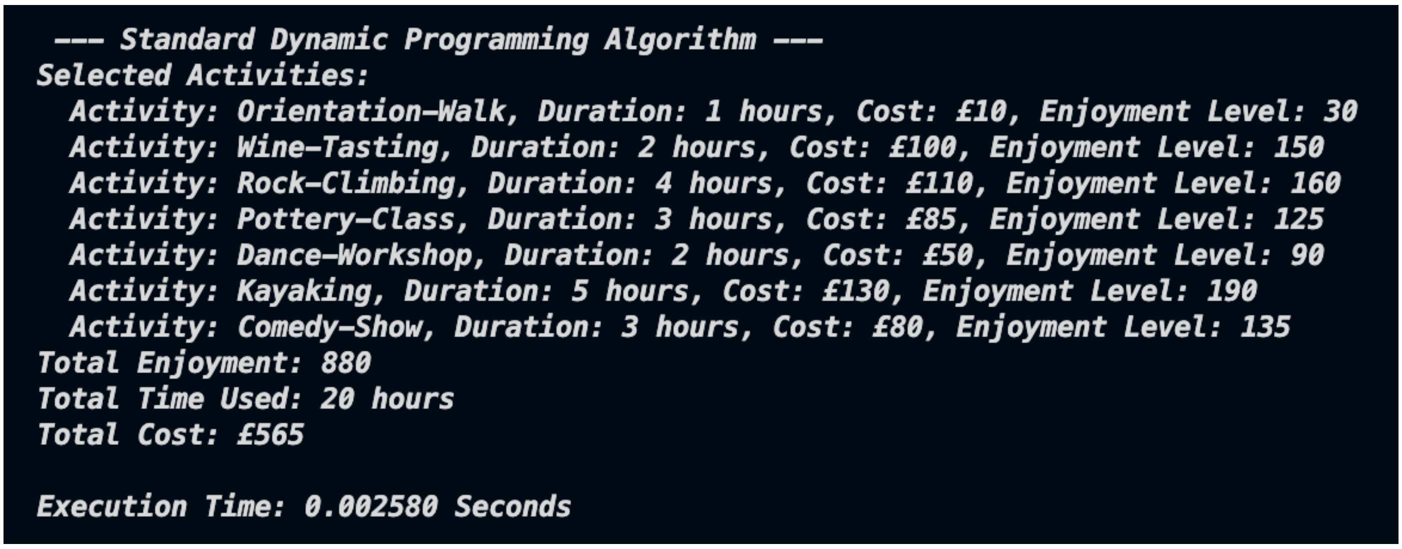


```
 --- Standard Dynamic Programming Algorithm ---
Selected Activities:
  Activity: Game-Night, Duration: 3 hours, Cost: £80, Enjoyment Level: 120
  Activity: Museum-Trip, Duration: 4 hours, Cost: £100, Enjoyment Level: 150
  Activity: Pizza-Workshop, Duration: 2 hours, Cost: £60, Enjoyment Level: 100
Total Enjoyment: 370
Total Time Used: 9 hours
Total Cost: £240

Execution Time: 0.000286 Seconds
```

Medium and Large Tests:



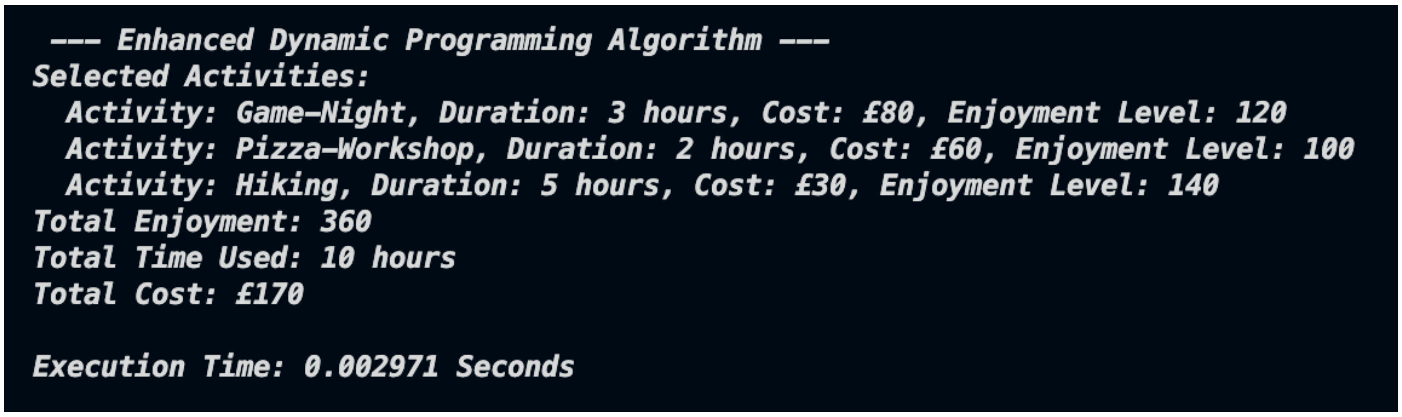
```
 --- Standard Dynamic Programming Algorithm ---
Selected Activities:
  Activity: Film-Screening, Duration: 3 hours, Cost: £30, Enjoyment Level: 90
  Activity: Art-Workshop, Duration: 2 hours, Cost: £70, Enjoyment Level: 95
  Activity: Bowling, Duration: 3 hours, Cost: £80, Enjoyment Level: 100
  Activity: Laser-Tag, Duration: 2 hours, Cost: £90, Enjoyment Level: 130
  Activity: Cooking-Class, Duration: 3 hours, Cost: £75, Enjoyment Level: 105
  Activity: Escape-Room, Duration: 2 hours, Cost: £85, Enjoyment Level: 115
Total Enjoyment: 635
Total Time Used: 15 hours
Total Cost: £430

Execution Time: 0.000757 Seconds
```



```
 --- Standard Dynamic Programming Algorithm ---
Selected Activities:
  Activity: Orientation-Walk, Duration: 1 hours, Cost: £10, Enjoyment Level: 30
  Activity: Wine-Tasting, Duration: 2 hours, Cost: £100, Enjoyment Level: 150
  Activity: Rock-Climbing, Duration: 4 hours, Cost: £110, Enjoyment Level: 160
  Activity: Pottery-Class, Duration: 3 hours, Cost: £85, Enjoyment Level: 125
  Activity: Dance-Workshop, Duration: 2 hours, Cost: £50, Enjoyment Level: 90
  Activity: Kayaking, Duration: 5 hours, Cost: £130, Enjoyment Level: 190
  Activity: Comedy-Show, Duration: 3 hours, Cost: £80, Enjoyment Level: 135
Total Enjoyment: 880
Total Time Used: 20 hours
Total Cost: £565

Execution Time: 0.002580 Seconds
```

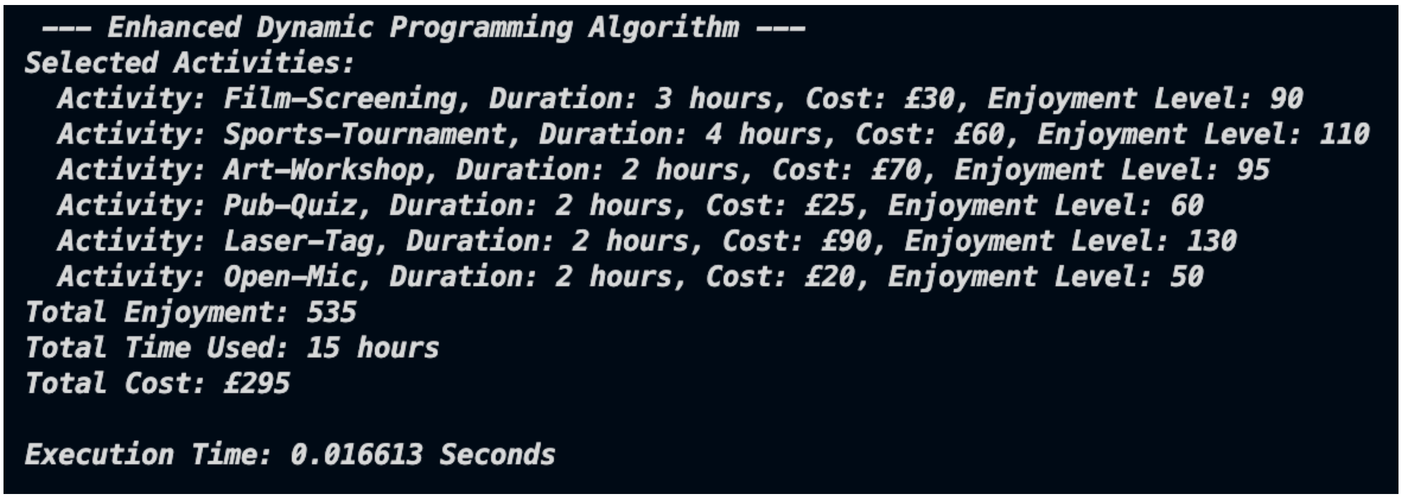
Dual Constraint Dynamic Programming Implementation

Small Test:

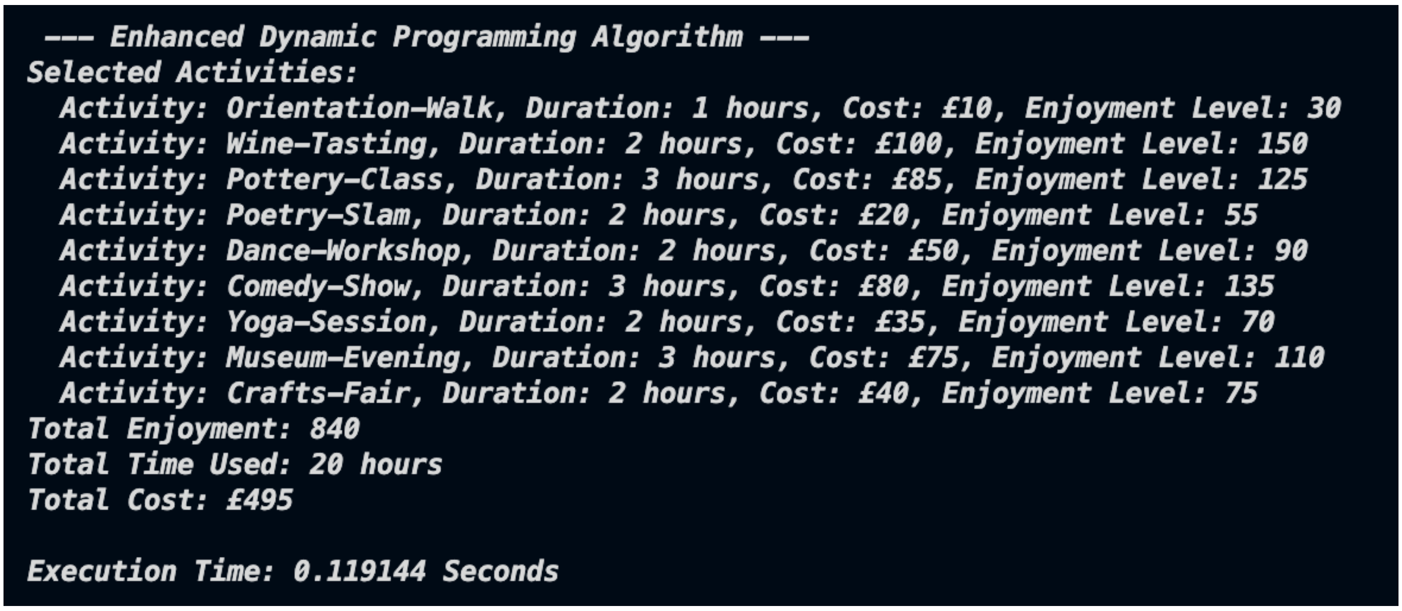


```
 --- Enhanced Dynamic Programming Algorithm ---
Selected Activities:
  Activity: Game-Night, Duration: 3 hours, Cost: £80, Enjoyment Level: 120
  Activity: Pizza-Workshop, Duration: 2 hours, Cost: £60, Enjoyment Level: 100
  Activity: Hiking, Duration: 5 hours, Cost: £30, Enjoyment Level: 140
Total Enjoyment: 360
Total Time Used: 10 hours
Total Cost: £170

Execution Time: 0.002971 Seconds
```

Medium and Large Tests:



```
 --- Enhanced Dynamic Programming Algorithm ---
Selected Activities:
  Activity: Film-Screening, Duration: 3 hours, Cost: £30, Enjoyment Level: 90
  Activity: Sports-Tournament, Duration: 4 hours, Cost: £60, Enjoyment Level: 110
  Activity: Art-Workshop, Duration: 2 hours, Cost: £70, Enjoyment Level: 95
  Activity: Pub-Quiz, Duration: 2 hours, Cost: £25, Enjoyment Level: 60
  Activity: Laser-Tag, Duration: 2 hours, Cost: £90, Enjoyment Level: 130
  Activity: Open-Mic, Duration: 2 hours, Cost: £20, Enjoyment Level: 50
Total Enjoyment: 535
Total Time Used: 15 hours
Total Cost: £295

Execution Time: 0.016613 Seconds
```



```
 --- Enhanced Dynamic Programming Algorithm ---
Selected Activities:
  Activity: Orientation-Walk, Duration: 1 hours, Cost: £10, Enjoyment Level: 30
  Activity: Wine-Tasting, Duration: 2 hours, Cost: £100, Enjoyment Level: 150
  Activity: Pottery-Class, Duration: 3 hours, Cost: £85, Enjoyment Level: 125
  Activity: Poetry-Slam, Duration: 2 hours, Cost: £20, Enjoyment Level: 55
  Activity: Dance-Workshop, Duration: 2 hours, Cost: £50, Enjoyment Level: 90
  Activity: Comedy-Show, Duration: 3 hours, Cost: £80, Enjoyment Level: 135
  Activity: Yoga-Session, Duration: 2 hours, Cost: £35, Enjoyment Level: 70
  Activity: Museum-Evening, Duration: 3 hours, Cost: £75, Enjoyment Level: 110
  Activity: Crafts-Fair, Duration: 2 hours, Cost: £40, Enjoyment Level: 75
Total Enjoyment: 840
Total Time Used: 20 hours
Total Cost: £495

Execution Time: 0.119144 Seconds
```

4. Optional Extensions:

5.1 Extension 1: Multiple constraints (Time + Budget)

To extend the planner beyond a single constraint, we modified the algorithms to enforce both total time and total budget simultaneously. This means a solution is only feasible if:' total_cost' <= budget and 'total_time' <= max_time

Changes to Brute Force:

The brute-force implementation was updated to evaluate both constraints when processing each subset. After summing cost, duration, and enjoyment, the algorithm verifies that both limits are satisfied before comparing against the best solution.

The time complexity remains O(2^n), as all subsets are still evaluated.

Changes to Dynamic Programming:

The dynamic programming implementation was extended from a single-capacity table to a two-dimensional table indexed by time and budget. Each state stores the maximum achievable enjoyment for a given combination of constraints.

Impact on Complexity:

While brute force remains exponential, the dynamic programming approach increases in dimensionality when given a second constraint.

For a single constraint, the time complexity is: **O(n x C)**
*where n is the number of activities and C is the capacity (time or budget)

With two constraints, the complexity becomes: **O(n x C1 x C2)**

*where C1 represents the time capacity, C2 represents the budget capacity

This is because the algorithm evaluates states across both dimensions of the DP table. Space complexity is also similar due to storage of the DP table.

Although this increases memory usage and runtime compared to the single constraint version, the approach still remains polynomial and is significantly more scalable than the exponential brute-force method for larger n.

Test Scenarios:

Test cases were created where:

- Budget is the limiting factor
- Time is the limiting factor
- Both constraints are binding

Thus, confirming the correctness within multi-constraint optimisation.

5.2 Advanced performance analysis and visualisation

Performance Testing setup

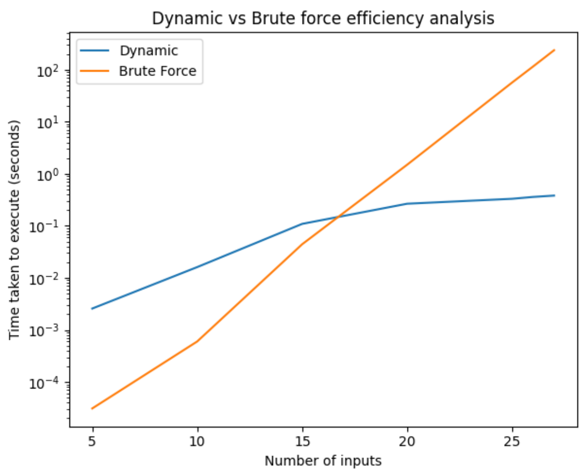
Additional test inputs were generated for increasing values of n (5,10,15,20,25,26, and 27 activities). Each input maintained realistic time and budget constraints.

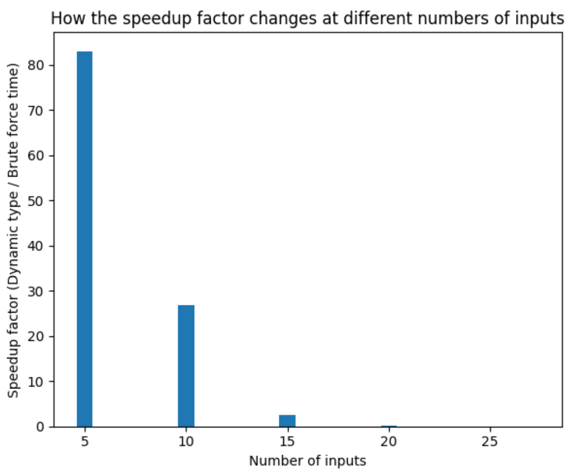
Execution times were recorded for both the brute-force and dynamic programming algorithms.

Testing for brute force was capped at n=27, as runtime reached approximately 250 seconds. Based on the observed exponential growth pattern, n=28 was estimated to exceed 10 minutes. Therefore, running any further tests was deemed impractical.

Dynamic programming was tested up to 200 inputs to observe long term scalability.

Execution Time comparison



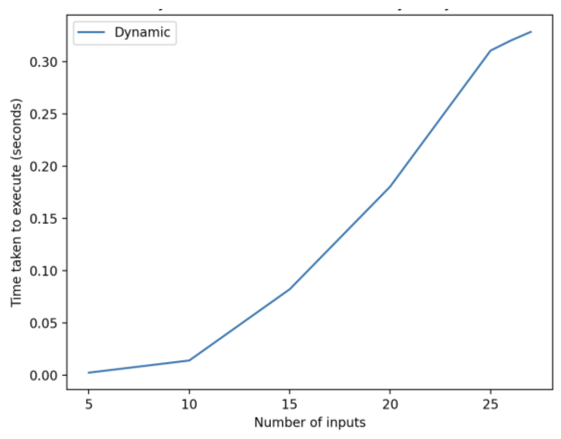


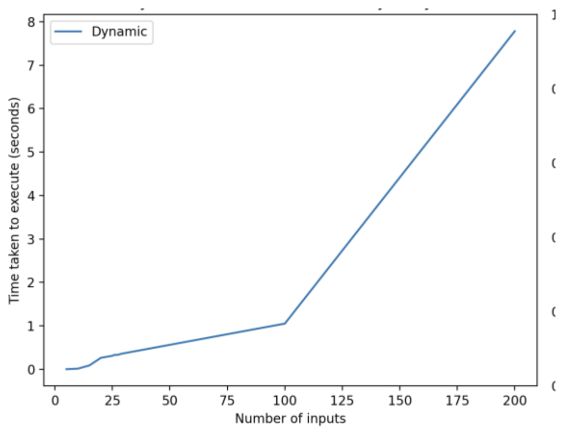
The graph clearly shows two different growth behaviours:

- **Brute force exhibits** exponential growth
  While it performs quickly at small input sizes, runtime is clearly seen to drastically increase as n grows. By n=27, execution time has reached that of approximately 250 seconds. Since the algorithm evaluates every possible subset of activities, the theoretical time complexity of O(2^n) is reflected through the algorithm's rapid escalation. As the number of activities increases, the number of possible combinations grow exponentially. This leads to a significant increase in computation time.
  In practice, brute force may be acceptable for small datasets (approximately 15 activities). However, beyond this point, dynamic programming becomes clearly preferable due to its significantly lower runtime.
- **Dynamic programming exhibits** more graduate growth
  Even when extended to 200 inputs, execution time still remains under 10 seconds. The growth pattern appears approximately polynomial rather than exponential, which aligns with theoretical expectations.

This supports the original hypothesis stated in the introduction: '*while brute force guarantees optimality at the cost of exponential growth, dynamic programming achieves the same optimal result more efficiently by avoiding redundant computation*'.

These results confirm that although both methods produce optimal solutions, their scalability differs significantly.





Speedup Factor Analysis (refer to speedup factor bar)

This was calculated as **Speedup = Brute Force time / Dynamic time**

At n=5, dynamic programming is already approximately 80x faster than brute force showing how the speedup grows significantly as n increases.
By 15-20 activities, the performance becomes substantial. Beyond this point, the relative difference is so large that the dynamic programming runtime becomes negligible compared to the brute force on the plotted scale.

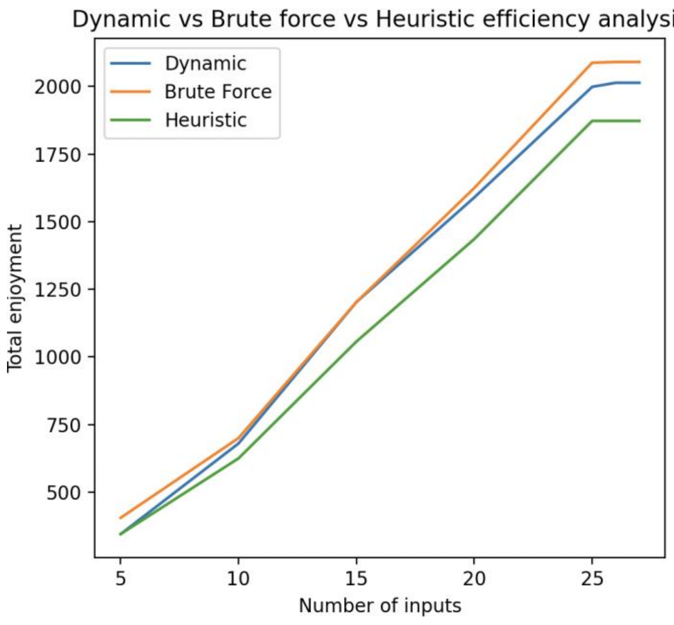
This demonstrates that the performance advantage of dynamic programming increases significantly as input size grows.

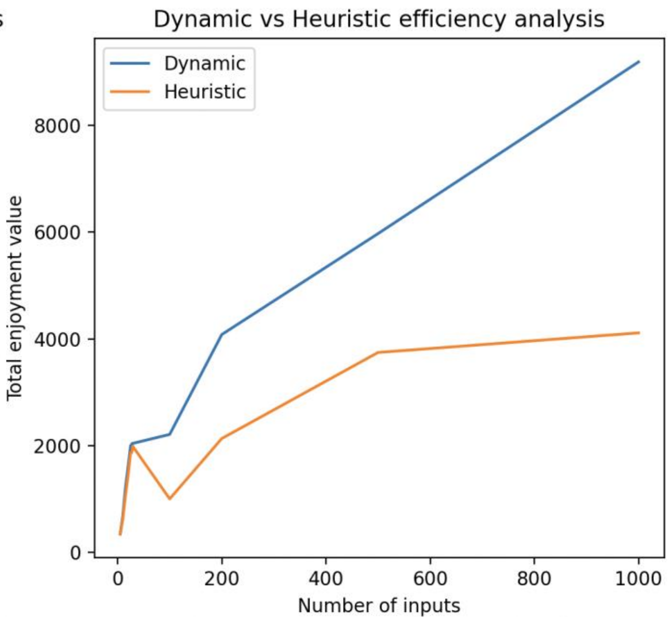
5.2 Extension 3: Greedy heuristic comparison

Greedy heuristic

The greedy heuristic algorithm first begins by calculating the two-ranking metrics for each activity: enjoyment per hour and enjoyment per cost. By using these ratios, two candidate solution lists are constructed through repeated selection of the highest-ranked activity until a maximum-length solution is reached.

The algorithm then compares the total enjoyment values of both lists and selects the higher-performing list to be the current best solution. If the selected activities exceed the time or budget constraints, the lowest-contributing activities are removed until feasibility is achieved.





Performance comparison

When comparing total enjoyment values, the greedy heuristic consistently produces slightly lower results compared to the dynamic programming solution for smaller input sizes. Since both brute-force and dynamic programming approaches compute the optimal solution, their outputs are identical.

However, as the number of inputs increase, the gap between the greedy heuristic and the optimal solution significantly widens.

When we examine larger inputs (400,600,1000), the greedy heuristic achieves approximately 58%, 50%, 46 of the optimal solution respectively. This downward trend illustrates the idea that as the problem size increases, the heuristic captures decreasing proportions of the optimal value. While this may be computationally efficient, the greedy method becomes less reliable for large datasets.

Why this happens?

The greedy heuristic performs best when the selection of high-ratio activities does not prevent the formation of the optimal overall combination. Meaning that if the activities that appear best individually are also the ones which belong in the best total schedule, then the greedy method achieves relatively optimal results.

This situation is more likely to occur with smaller inputs because there are fewer combinations which exist. Thus, there is less opportunity for a sub-optimal early choice to block better overall solutions.

However, as the number of inputs increases, the number of possible combinations grow exponentially; it becomes more likely that greedy selections prevent said problem to happen and deny more favourable combinations. As a result, the heuristic becomes less reliable at larger input sizes, therefore trading solution quality for computational efficiency (ScienceDirect, n.d).

5. Generative AI statement

Theo Swan: SID: 750041957: No LLMs have been used in my contribution to the coursework.

Henry Beedham: SID 750028696: No LLMs have been used in my contribution to the coursework.

Howard Alix: SID 750025583: No LLMs have been used in my contribution to the coursework.

Boran Emin: SID 750025583: No LLMs have been used in my contribution to the coursework.

Stephen Dorey: SID 750039082: No LLMs have been used in my contribution to the coursework.

6. References

ScienceDirect (n.d.) Greedy heuristic. Available at: https://www.sciencedirect.com/topics/computer-science/greedy-heuristic