Data Structures and Algorithms Coursework

Group 26

Introduction **1**
Design and Approach **2**
Program Structure 2
Data Structures Used 2
Assumptions & Simplifications 3
Algorithms **4**
Brute-force pseudocode 4
Brute-force logic and approach 5
Dynamic pseudocode: 6
Dynamic programming logic and approach 7
Time and Space Complexity 8
Testing and Results **9**
How We Tested Our Program 9
Results 10
Test 1 10
Test 2 11
Test 3 12
Limitations/Edge Cases: 13
Test 4 13
Conclusion from testing 14
Optional Extensions **15**
Extension 1 - Multivariable Dynamic Programming Algorithm 15
Extension 2 - Advanced Performance Analysis and Visualisation 16
Extension 3 - The Greedy Heuristic Algorithm 17
Generative AI Statement **18**
References **19**

Introduction

The ‘Event Planner Scenario’ is a decision-making problem faced by a committee for a university student society organising a weekend event. The committee has a fixed amount of time, a fixed budget and a list of activities all with unique time required, cost and enjoyment levels. Each activity can be fully included or excluded, no inbetween. The challenge is to implement a solution that finds the highest enjoyment level of combined activities while staying within the constraints of either maximum budget or maximum time. In our solution we chose to do this as well as finding a solution that finds the combination of activities that has the highest enjoyment level while following both constraints.

The objectives of the coursework are to implement an event planner system that takes a list of activities, applies the chosen constraints and outputs the optimal set of activities that maximises enjoyment. We are required to implement and use two different algorithms to do so. A brute-force approach that generates all possible combinations and checks them, guaranteeing to find the optimal solution. Then a dynamic programming approach was also implemented, solving the same problem more efficiently demonstrating scalability with different activity list input sizes.

Design and Approach

Program Structure

The program is structured by separating each large responsibility into its own function in event_planner.py. Input handling is done in the get_input function which reads the input file and returns the number of activities, time and budget constraints and the actual list of activities. The get_input function is designed to handle inputs of varying sizes.

The algorithms are assigned their own respective functions, brute_force_optimal_plan (brute-force algorithm) and dp (dynamic programming algorithm). By keeping the algorithms separated it made it easier to compare results and performance.

To run the program there are also helper functions (run_time_constraint and run_budget_constraint) which are used to decide whether time or budget is the primary constraint. Output formatting is handled by the respective functions print_dp_output for dynamic programming and print_bf_output for the brute force algorithm.

To coordinate the entire program, a main execution block (if __name__ = “__main__”:) is used to run both algorithms, time their execution and print results.

Data Structures Used

In the **brute-force algorithm** the following data structures were used:

- *Dataclass (Activity): used to group all attributes together; cleaner than using a dictionary or tuple; improves readability; frozen=True prevents accidental modification*
- *List (Activity): stores all activities; easy iteration*
- *Bitmasking (integers): time complexity; guarantees every subset combination checked; recursion avoided*
- *Dictionaries (for output): used for structured output; results easy to interpret, more readable than tuple*
- *Local accumulators (primitive types): fast arithmetic operations; constraint checking*

In the **dynamic algorithm** the following data structures were used:

- *List of lists (activities): stores all activities in a simple format, allowing fast access while also being compatible with dynamic programming table indexing*
- *2D list (dp_grid): allows for solutions to subproblems to be stored and comparison of decisions efficiently*
- *2D list (keep_grid): allows backtracking to reconstruct which activities were chosen, separating value calculation from reconstruction, improving overall clarity*
- *Scalars for constraints and indexing (capacity, enjoyment_index): makes function flexible for both constraints*

Assumptions & Simplifications

Brute-force

- Activities are independent and do not depend on each other
- Time and cost values are non negative
- Each activity can only be selected once
- Constraints may be optional if set to None

Dynamic programming

- The problem follows the zero one selection model
- Capacity values are treated as integers
- Only one main constraint is used as the table dimension at a time
- Each activity can only be chosen once

Algorithms

Brute-force pseudocode

```
BRUTE_FORCE_ACTIVITIES(activities, max_time, max_budget):
        n <- number of activities
        best_subset <- empty
        best_enjoyment <- - infinite
        best_time <- 0
        best_cost <- 0

        for mask from 0 to (2^n – 1):
                subset <- empty
                total_time <- 0
                total_cost <- 0
                total_enjoyment <- 0

                  for i from 0 to(n - 1):
                  if (mask has bit i set):
                  add activities[i] to subset
                          total_time += activities[i].time
                          total_cost += activities[i].cost
                          total_enjoyment += activities[i].enjoyment

                  feasible <- TRUE
                  if max_time is not None and total_time > max_time;
                          feasible <- FALSE
                  if max_budget is not None and total_cost > max_budget;
                          feasible <- FALSE

                  if feasible and total_enjoyment . best_enjoyment:
                          best subset <- subset
                          best_ enjoyment <- total_enjoymeny
                          best__time <- total_time
                          best_cost <- total_cost

        return (best_subset, best_enjoyment, best_time, best_cost)
```

Brute-force logic and approach

The brute-force algorithm tries every possible combination of activities to find the highest enjoyment value. Starting with a list of all activities, the algorithm generates all combinations, whether it is just one activity, a few, all of them or none of them.

For each combination the total time, money and enjoyment value is calculated. Each combination is then checked to see if it follows the time or budget constraint chosen, otherwise that combination is rejected.

If the combination is valid, its total enjoyment value is compared to that of the best one found so far, if it's better then it becomes the new best option.

The algorithm continues this process until every combination created has been checked. As no combinations are skipped its guaranteed that the best solution will be found. If no combinations fit the constraints then the algorithm returns an empty result. If a combination that follows the constraints is found then the set of activities that gives the highest enjoyment is returned.

Dynamic pseudocode:

```
 FOR each activity i from 1 to number_of_activites

        weight = activity i time or cost
        value = activity I enjoyment
        best_enjoyment[i][capacity] = maximum enjoyment using i activities

        FOR each capacity from 0 to max_capacity

                skip = best_enjoyment [i - 1][capacity] #what to do if we skip the current activity

                IF activity fits (weight is less than or equal to capacity) THEN
                        take = best_enjoyment[i - 1][c - weight] + value
                ELSE
                        take = - 1 #don't take it because it doesn't fit
                END IF

                Best_enjoyment[i][c] = max(skip, take) #take the better option

                IF take > skip THEN #remember how the best enjoyment was chosen
                        chosen[i][capacity] = TRUE
                ELSE
                        chosen[i][capacity] = FALSE
                END IF

        END FOR
END FOR
```

The backtracking algorithm (so we can reconstruct the chosen activities)

```
selected_activities = empty list

current_capacity = max_capacity
i = number_of_activities

WHILE i > 0 DO

        IF chosen[i][current_capacity] == TRUE THEN

                ADD activity I to selected_activities #activity I was taken so we include it

                weight = activity I time or cost
```

```
        current_capacity = current_capacity - weight
    END IF

    i = i - 1 #go to the last activity

END WHILE

REVERSE selected_activities #put the activities in the original order
```

Dynamic programming logic and approach

The dynamic programming algorithm is implemented through the function “dp”. It implements a dynamic programming solution to the problem, which is modelled as a 0/1 Knapsack optimisation problem[1]. The objective is the maximise total enjoyment while keeping to one constraint, from the list of activities each activity may be either included or excluded from the schedule.

The mode parameter allows for the algorithm to use either the time or budget constraint. Depending on which constraint is chosen, the limit is treated as the knapsack capacity[2] and the chosen constraint from each activity is used as the weight.

Two tables are then created, “dp_grid” which stores the maximum enjoyment possible for both a given number of activities and a given remaining capacity. The table “keep_grid” stores True or False values that record whether an activity was chosen and is then later used to reconstruct the final list of included activities.

The main logic of the algorithm is that it considers activities one at a time, and for each respective activity and each possible capacity value decides whether to skip the activity or add the activity’s enjoyment value to the best enjoyment possible with the remaining capacity. Whichever option gives higher total enjoyment is chosen by the algorithm and the choice is recorded in the tables.

Once the table is full, the algorithm backtracks through “keep_grid”. If an activity was chosen by the algorithm, it is added to the results list. The remaining capacity is then reduced accordingly and the algorithm moves on to the previous activity.

After the set of activities that produced the maximum enjoyment is reconstructed, the code calculates total enjoyment, total time used and total cost.

---

[1]ByteQuest. *0/1 Knapsack Problem Explained Visually*. March 2025, https://www.youtube.com/watch?v=qxWu-SeAqe4. Accessed 23 Feb 2026.
[2] “DSA The 0/1 Knapsack Problem.” *W3Schools*, https://www.w3schools.com/dsa/dsa_ref_knapsack.php. Accessed 23 February 2026.

Time and Space Complexity

The approaches to solving the Event Planner problem differ in efficiency.

The **brute-force** algorithm checks every possible combination of activities. As the number of activities increases the number of combinations grows exponentially. When the input size of activities is small it remains manageable, however even a medium increase in input size leads to a significant increase of combinations to evaluate. Because of this the running time of the algorithm increases rapidly becoming impractical for larger inputs. The space usage of the brute-force algorithm is relatively small, as the only memory it uses is to store the current subset being evaluated and the current best solution.

On the other hand, the **dynamic programming** algorithm comes up with a solution to the problem in parts, storing theoretical final results in tables. The running table of the algorithm grows in proportion to the number of activities multiplied by the size of the chosen constraint[3]. Because of this the algorithm runs in polynomial time and deals with scaling size of inputs much better than the brute-force algorithm. But the improvement of time efficiency results in the cost of space efficiency. The memory usage is much higher, especially with larger inputs, as the dynamic programming algorithm uses two two dimensional tables, of which the sizes.

To conclude, both algorithms guarantee finding the optimal solution, however the brute-force algorithm is simpler but only suitable for smaller input sizes. The dynamic programming algorithm is more time efficient for larger inputs and shows the trade-off between time and space efficiency.

[3] “Knapsack Problem.” *Open Courser*, 01 May 2024, https://opencourser.com/topic/q1uxs1/knapsack-problem. Accessed 24 Feb 2026.

Testing and Results

How We Tested Our Program

To ensure accurate testing, we picked two testing strategies when testing the input_small.txt. Manual calculations and running the program and recording the results for the file. For the rest of the test files, we ran the program and recorded their results for the test file specified. Using manual calculations ensured that both the Brute Force and Dynamic Programming algorithms were logically correct. We used the three provided input files(small, medium, and large) to evaluate the scalability of our algorithms, measuring how execution time grows as the number of activities(n) increases. Then, we performed edge case testing by modifying the input_small.txt file and changing the budget to £0 to confirm that the program handles "no feasible solution" scenarios without crashing. This approach allowed us to confirm that our system consistently identifies the optimal solution while highlighting the efficiency gains of the Dynamic Programming approach over the Brute Force method.

Results

Test 1

Input File: input_small.txt Number of activities: 5 Available Time: 10 Hours
Available Budget: £200

	Dynamic Programming algorithm	**Brute Force Algorithm**
Selected Activities	Campus-Tour (2 hours, £20, enjoyment 50) Game-Night (3 hours, £80, enjoyment 120) Pizza-Workshop (2 hours, £60, enjoyment 100) Hiking (5 hours, £30, enjoyment 140)	Campus-Tour (2 hours, £20, enjoyment 50) Game-Night (3 hours, £80, enjoyment 120) Pizza-Workshop (2 hours, £60, enjoyment 100) Hiking (5 hours, £30, enjoyment 140)
Total Enjoyment	410	410
Total Time Used	12 hours	12 hours
Total Cost	£190	£190
Execution Time	9.03330510482192e-05 seconds	1.6249949112534523e-05 seconds

Manual Calculations for Test 1:
There are 2^5 total combinations. 2^5=32. However, some combinations do not fit within the budget and are over £200. We have listed the top 5 results from our manual calculations.

Activities	**Total Cost**	**Total Enjoyment**
Campus-Tour, Game-Night, Pizza-Workshop, Hiking	£190	410
Museum-Trip, Pizza-Workshop, Hiking	£190	390
Game-Night, Pizza-Workshop, Hiking	£170	360
Campus-Tour, Museum-Trip, Hiking	£150	340
Campus-Tour, Museum-Trip, Game-Night	£200	320

Test 2

Input File: input_medium.txt Number of activities: 12 Available Time: 15 Hours
Available Budget: £300

	Dynamic Programming algorithm	**Brute Force Algorithm**
Selected Activities	Welcome-BBQ (3 hours, £50, enjoyment 80) Karaoke-Night (2 hours, £40, enjoyment 70) Film-Screening (3 hours, £30, enjoyment 90) Sports-Tournament (4 hours, £60, enjoyment 110) Pub-Quiz (2 hours, £25, enjoyment 60) Cooking-Class (3 hours, £75, enjoyment 105) Open-Mic (2 hours, £20, enjoyment 50)	Welcome-BBQ (3 hours, £50, enjoyment 80) Karaoke-Night (2 hours, £40, enjoyment 70) Film-Screening (3 hours, £30, enjoyment 90) Sports-Tournament (4 hours, £60, enjoyment 110) Pub-Quiz (2 hours, £25, enjoyment 60) Cooking-Class (3 hours, £75, enjoyment 105) Open-Mic (2 hours, £20, enjoyment 50)
Total Enjoyment	565	565
Total Time Used	19 hours	19 hours
Total Cost	£300	£300
Execution Time	0.0002705829683691263 seconds	0.002095125033520162 seconds

Test 3

Input File: input_large.txt Number of activities: 25 Available Time: 20 Hours
Available Budget: £500

	Dynamic Programming algorithm	**Brute Force Algorithm**
Selected Activities	Orientation-Walk (1 hours, £10, enjoyment 30) Ice-Breaker-Games (2 hours, £20, enjoyment 50) Movie-Marathon (5 hours, £60, enjoyment 140) Charity-Run (3 hours, £15, enjoyment 70) Trivia-Night (2 hours, £30, enjoyment 65) Campus-Scavenger-Hunt (3 hours, £25, enjoyment 75) Photography-Walk (3 hours, £40, enjoyment 85) Poetry-Slam (2 hours, £20, enjoyment 55) Dance-Workshop (2 hours, £50, enjoyment 90) Baking-Competition (3 hours, £55, enjoyment 95) Board-Game-Cafe (3 hours, £45, enjoyment 80) Volunteering-Event (4 hours, £10, enjoyment 60) Park-Picnic (3 hours, £30, enjoyment 65) Stargazing-Trip (4 hours, £50, enjoyment 100) Crafts-Fair (2 hours, £40, enjoyment 75)	Orientation-Walk (1 hours, £10, enjoyment 30) Ice-Breaker-Games (2 hours, £20, enjoyment 50) Movie-Marathon (5 hours, £60, enjoyment 140) Charity-Run (3 hours, £15, enjoyment 70) Trivia-Night (2 hours, £30, enjoyment 65) Campus-Scavenger-Hunt (3 hours, £25, enjoyment 75) Photography-Walk (3 hours, £40, enjoyment 85) Poetry-Slam (2 hours, £20, enjoyment 55) Dance-Workshop (2 hours, £50, enjoyment 90) Baking-Competition (3 hours, £55, enjoyment 95) Board-Game-Cafe (3 hours, £45, enjoyment 80) Volunteering-Event (4 hours, £10, enjoyment 60) Park-Picnic (3 hours, £30, enjoyment 65) Stargazing-Trip (4 hours, £50, enjoyment 100) Crafts-Fair (2 hours, £40, enjoyment 75)
Total Enjoyment	1135	1135
Total Time Used	42 hours	42 hours
Total Cost	£500	£500.0
Execution Time	0.0009689159924164414 seconds	36.40540541696828 seconds

Limitations/Edge Cases:

input_edge.txt is the same as input_medium.txt
The only difference is the budget constraint. It has been changed to 0 to test for an edge case.

Test 4

Input File: input_edge.txt
Number of activities: 12
Available Time: 15 Hours
Available Budget: £0

	Dynamic Programming algorithm	**Brute Force Algorithm**
Selected Activities		
Total Enjoyment	0	0
Total Time Used	0 hours	0 hours
Total Cost	£0	£0
Execution Time	7.958034984767437e-06 seconds	0.0022045420482754707 seconds

Conclusion from testing

The testing results confirm that both the Brute Force and Dynamic Programming algorithms successfully identified the same optimal enjoyment values for all testing scenarios. This shows that the algorithms were correctly implemented. Manual calculations for the small input file confirmed that the algorithms correctly identify the best case scenario, which in this case is the enjoyment value of 410.

The execution time shows a significant gap in the scalability of the two approaches. The Brute Force algorithms execution time grew from approximately 1.6*10^-5 for the small input file to approximately 36 seconds for the large input file. This shows how impractical the Brute Force algorithm is as the number of activities increases. However, the Dynamic Programming algorithm was pretty consistent overall, staying under 0.001 seconds for all tests. This shows that the Dynamic programming approach is better in real word scenarios as it can handle scenarios where there is a larger number of activities better than the Brute Force approach.

The last thing to test was an edge case. We modified the input_medium.txt and changed the budget available to 0. This file was then renamed to input_edge.txt and we ran the tests. The program handled this and returned an empty set of activities and 0 enjoyment without facing an error. This shows that our program is accurate and robust.

Optional Extensions

Extension 1 - Multivariable Dynamic Programming Algorithm

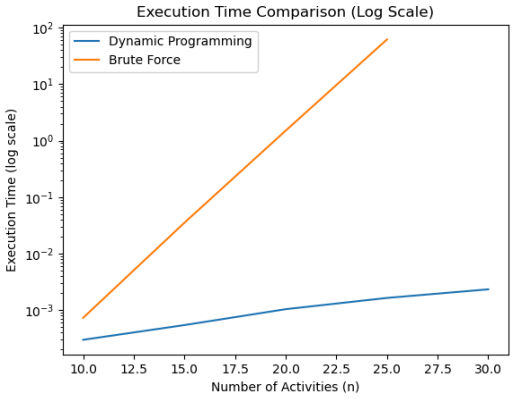
An adapted version of the dynamic programming algorithm has been created (dynamic_programming_multivariable_extension.py) so that instead of holding budget or time as a constraint, it creates a 3D array that holds both budget and time as constraints at the same time. Testing has been done for input files of 10, 15 and 20 activities, and the results are contained in the file: “multivariable_test_results.txt”.

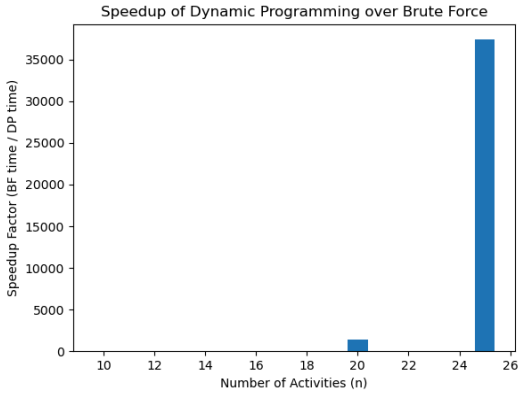
The time and space complexity also increase due to the addition of another dimension in the dp table. This means that it goes from O(n * T) to O(n * T * B), where n is the number of activities, T is the time constraint, and B is the budget constraint.

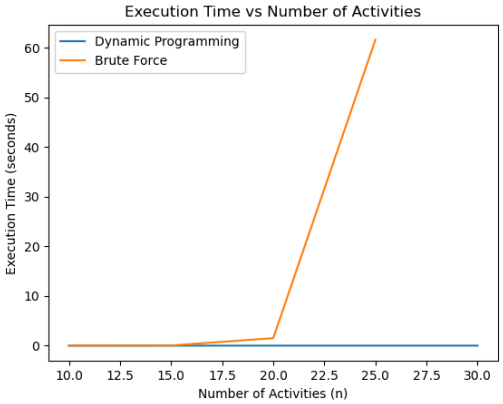
The brute force algorithm follows its original logic, using bitmasking to check every 2n combination. The only major change is the addition of a second requirement to ensure each selection stays within both the time and budget limits. While these extra checks do not change the O(2n) time complexity, the n=30 results show a major failure. This shows that the calculations eventually become too high for the system to manage. Therefore, we believe that this method is not suitable for larger tasks when compared to the efficiency of the dynamic programming version.

The results of testing for 10, 15, 20, 25 and 30 activities for both algorithms are contained in the file: “multivariable_test_results.txt”.

Extension 2 - **Advanced Performance Analysis and Visualisation**







These are the results for running both algorithms with the test files: input_10, input_15, input_20, input_25 and input_30. We had to skip the input_30 file with the brute force algorithm due to the runtime being much too high. We can draw some interesting conclusions from the graphs. We found that after n = 20 for the brute force algorithm, the time skyrocketed exponentially and was extremely far from the runtime of the dynamic programming algorithm. Also, it is not practical at all to use a speedup bar graph due to the ratio becoming incredibly high at n = 25; we are not even able to see the n = 10 and n = 15 values on the chart, as they are proportionally tiny in comparison to the n = 20 and n = 25 values.

As n increases, the dynamic programming execution time really doesn't grow that much, but we can see that it is following a log(n) graph.

The results do match the theoretical complexity that we would expect due to the structure of the 2 algorithms. The dynamic programming runtime scales polynomially with the number of activities and constraints. The brute force algorithm however, scales exponentially with the number of activities and the constraints. This leads to huge performance penalties at high values of n, as it is 2^n.

Extension 3 - The Greedy Heuristic Algorithm

The Greedy Heuristic Algorithm[4] that we chose to implement uses the ratio of enjoyment divided by either time or cost. The nature of this algorithm is such that it picks the first good enjoyment/time/cost ratio without looking at the other ones. This leads to it sometimes being close to the optimal solution if it happens to find the correct solution first, but it will often stray as it is just looking at the local maxima, opposed to the global maximum, hence why it's called a ‘greedy’ algorithm.

A time where the algorithm would perform well would be when the enjoyment per time/budget ratios are all very similar, as the greedy choice would naturally align with the optimal combination.

A time where the algorithm would perform poorly would be when enjoyment is high but maybe time and cost are also high as well, leading to it potentially choosing one high enjoyment, high time/cost activity opposed to a few lower enjoyment and lower time/cost activities, which together would give a higher total enjoyment but are overlooked because the greedy algorithm commits too early to the large activity due to its nature.

This means that the greedy algorithm is fast and simple, but by definition short sighted which explains why it performs well in some cases and poorly in others.

[4] (“DSA Greedy Algorithms”)

Generative AI Statement

No AI was used

References

Works Cited

ByteQuest. *0/1 Knapsack Problem Explained Visually*. March 2025, https://www.youtube.com/watch?v=qxWu-SeAqe4. Accessed 23 Feb 2026.

“Knapsack Problem.” *Open Courser*, 01 May 2024, https://opencourser.com/topic/q1uxs1/knapsack-problem. Accessed 24 Feb 2026.

“DSA The 0/1 Knapsack Problem.” *W3Schools*, https://www.w3schools.com/dsa/dsa_ref_knapsack.php. Accessed 23 February 2026.

Greedy Heuristic Algorithm

GeeksforGeeks. “0/1 Knapsack Problem.” 29 January 2026, https://www.geeksforgeeks.org/dsa/0-1-knapsack-problem-dp-10/. Accessed 23 February 2026.

Dynamic Programming Algorithm

“DSA Greedy Algorithms.” *W3Schools*, https://www.w3schools.com/dsa/dsa_ref_greedy.php. Accessed 23 February 2026.