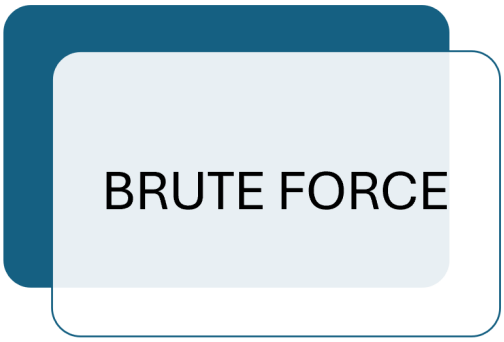
Society Event Planner

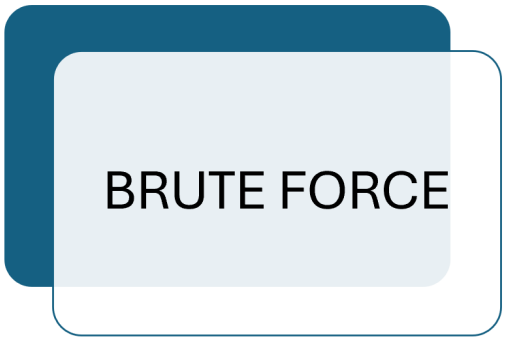
Howard Alix, Henry Beedham, Stephen Dorey, Boran emin, Theo Swan


Introduction to
the problem

Algorithmic Explanation



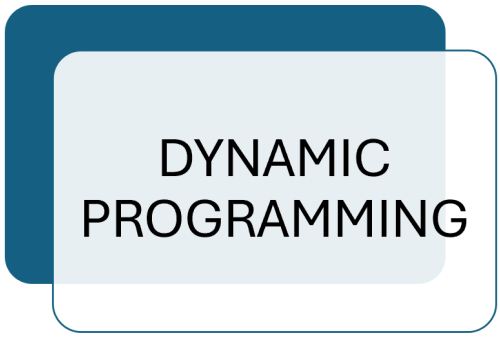
Algorithmic Explanation




```
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
```

For the Brute Force Algorithm all the possible combinations are generated of the current length and each combination is individually when a better solution (that meets the current constraints and produces a higher enjoyment score) is found then best solution is overwritten with the new combination and the associated cost, duration and enjoyment. This process is repeated for each combination for each length from the maximum length to the minimum length of 0(no possible combination). After this entire process is complete, a list of the combination and the cost time and enjoyment are returned to the main program.

Algorithmic Explanation



The dynamic algorithm is the classic 1D Knapsack problem. As we have chosen to do the extension to take into account both time and budget constraints, it's the 2D knapsack problem.

The algorithm builds a grid.
Rows represent every possible hour you could spent from (0 to max time)
Columns represent every pound of the budged from (0 to total)

We start with a table full of zeros

For each activity on the list the algorithm looks at every time and budged combination to see if the activity fits. It does this in reverse to prevent picking the same activity twice

If adding the activity causes the total enjoyment to increase for that cell, the table gets updated with the new value

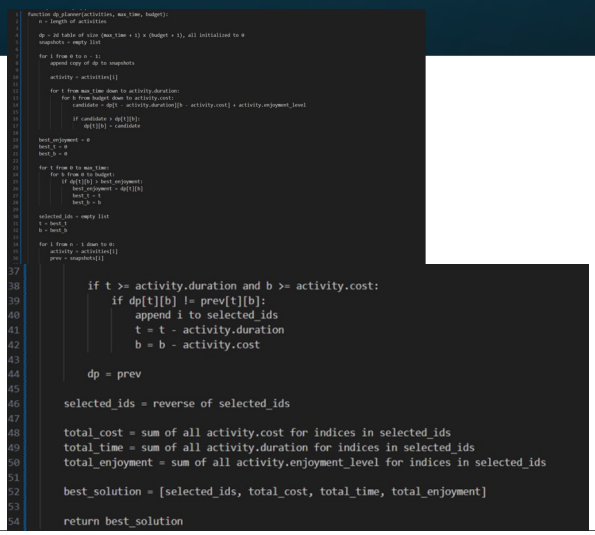
Before processing each activity, it takes a snapshot of the table in its current state.

Its like a save point in a game.

We use these because at the end the table only tells us total enjoyment and to findout out what we picked, we compare the table before and after each activity was considered.

After processing all of the activities, we scan the table for the highest enjoyment value. That cell contains our perfect plan (the total time used, money spent, and max enjoyment.
To get the list of our chosen activities we traceback that best cell using the snapshots we took before processing each activity. If the value in the cell changed after adding an activity, we know that is one of the chosen activites. Then, we subtract the time and cost of that activity and move back until we've found all of the chosen activities.

Algorithmic Explanation



```
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
```

DYNAMIC PROGRAMMING

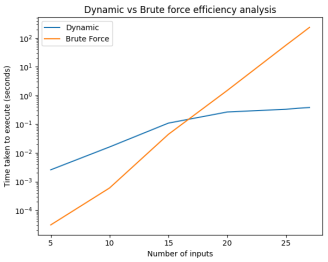
Algorithmic Explanation

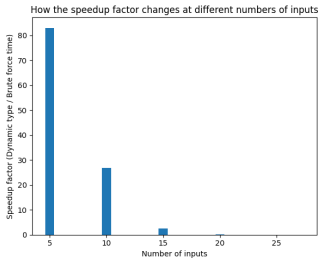
BRUTE FORCE

DYNAMIC PROGRAMMING

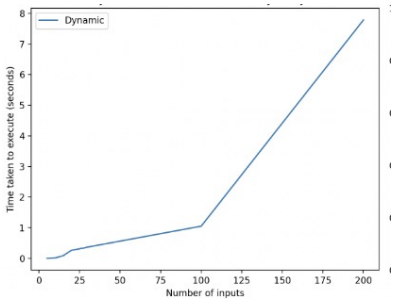

Demonstration

Results and Discussion

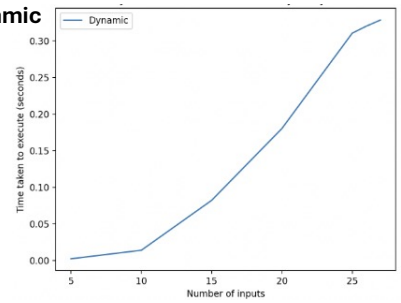




Dynamic time taken to execute based on higher number of inputs



Graph for more detailed analysis specifically on dynamic



The program runs both dynamic and brute force timing at intevals, 5,10,15,20,25,26 and 27. I cut off at 27because I noticed a pattern of it doubling each time and after 27 took ~250 seconds, I'd expect 28 to take nearly 10 minutes compared to dynamic not even taking a second I decided I didn't want to have that run.

The rate of increase in time for brute force, is exponentially growing as the number of inputs increase. Whilst the rate of increase for dynamic seems to increase much more gradually. With the time taken for 200 inputs to be computed not even reaching 10 seconds, compared to brute force taking over 50 seconds for 25 inputs.

I believe there is no set time where brute force is useless, depends on requirements. For organising only 12 hours on the same day of work it could be difficult if there are 100 inputs which could take days to complete. Which would be useless, since it would not be completed in the required time frame, especially since dynamic only a few seconds. However if they are organising a whole month of activities which could have 150 or so hours.This would be more

reasonable to complete although I'd choose dynamic over brute force at any time over 20 activites. – (That’s the conclusion I came to but the coursework states over 10 minutes as an example of when you might state it being useless which I estimate to be 28 being the first one to take over 10 minutes)

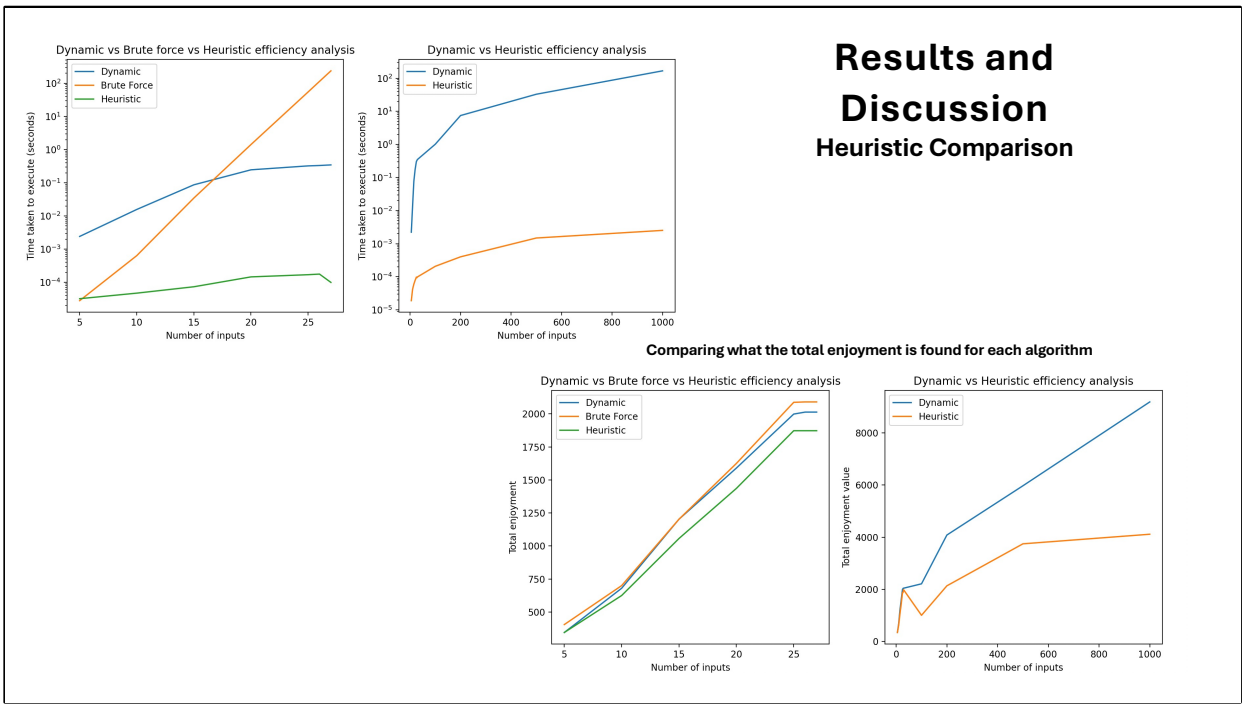
For the speedup factor the bar graph shows what percentage is the speed the dynamic is of the brute force. Such as the first point for 5 activites, the dynamic planner completes in 80% of the time of the brute force. This reduces dramatically at each increase, to a point of at 20 you can barely even see the bar at all. Beyond that the bar not being visible at all. This shows that the increase in performance from choosing the dynamic planner rather than the brute force planner increases dramatically as the number of activities increases.

Greedy Heuristic

--- Greedy Heuristic Algorithm ---
Selected Activities:
Activity: Museum-Trip, Duration: 4 hours, Cost: £100, Enjoyment Level: 150
Activity: Hiking, Duration: 5 hours, Cost: £30, Enjoyment Level: 140
Total Enjoyment: 290
Total Time Used: 9 hours
Total Cost: £130

Execution Time: 0.000019 Seconds

The greedy heuristic algorithm first calculates the enjoyment per hour and enjoyment per cost of every activity. It creates two initial lists by working out the maximum length of a solution and adding the highest enjoyment per cost or enjoyment per hour activity to the list until it reaches the maximum length Now it check between these two lists to see which has the highest total enjoyment value sets that as the current best solution. Finally, it removes the lowest enjoyment items until the selection fits within the budget and time constraints. Then it submits the final



The greedy heuristic algorithm finishes the fastest out of all of the algorithms as well as increasing time taken at the slowest rate. However it doesn't find the exact best solution every time

When checking for what total enjoyment level we get for each value, the heuristic is always a little bit lower than the dynamic for lower input levels. The brute force always being higher. However, the level of how good the heuristic is drops off heavily the more inputs are added, with there being a significant difference at 1000 inputs.


Questions