**Grandma’s Pokémon Card Distribution**

1. **Goals**

This assignment may seem like a family story, but it is a dynamic programming problem about distributing cards. Grandma Rosa owns a collection of Pokémon cards, each with a monetary value, and she wants to pass them on to her granddaughters while following specific rules of fairness. If only one granddaughter is chosen, she receives the entire deck; if two are chosen, the favorite must get the higher-valued set, and if the favorite is not included, cards may need to be discarded so both grandchildren end up with exactly equal totals; and if all three are chosen, the favorite must receive the largest share, followed by a second favorite, while the remaining grandchild gets the rest.

In every case, the program must output which cards were given to whom, the total value each received, and whether any cards were discarded. In my case, the favorite was determined from my initials “PDC,” where P is the 16th letter of the alphabet, D is the 4th, and C is the 3rd, summing to 23, and taking the modulus of 3 gives 2, which corresponds to Melanie as my favorite grandchild.

I implemented both the main function (PDC\_ca3) and the extra credit function (PDC\_ec\_ca3). The main function distributes the cards fairly for up to three grandchildren while ensuring that the favorite rules are respected, while the extra credit function extends the solution to larger groups of grandchildren (up to 16 in my tests) to demonstrate the scalability of the approach.

For my computing environment, I used a MacBook Pro (Apple Silicon M4, base model) with a 512-GB SSD and the base unified-memory configuration, running macOS with Python 3.13 (CPU only) to implement the program and to measure timing and memory usage during experiments.

1. **Dynamic Programming Algorithms**

Dynamic Programming (DP) is a problem-solving method that helps us solve big problems by breaking them into smaller pieces. Many of these smaller problems repeat, so instead of solving the same problem repeatedly, DP solves each one once, saves the answer, and reuses it later. This makes the program much faster and more efficient. DP is often used in situations where we want to find the best or most fair solution, like splitting items evenly, finding the shortest path, or planning resources.

A diagram of a problem

AI-generated content may be incorrect.

*Fig 1: Dynamic Programming Data flow diagram*

Memoization is a top-down way of solving problems in Dynamic Programming. It works by solving the big problem with recursion, breaking it into smaller parts. Each time a small problem is solved, the result is saved, so if it comes up again, the program just looks it up instead of solving it again.

A diagram of a computer error

AI-generated content may be incorrect.

*Fig 2: Memoization Data flow diagram*

Tabulation is a bottom-up approach to Dynamic Programming. In this method, we first solve the simplest subproblems (called base cases) and store their answers in a DP table. Then, step by step, we use those results to solve slightly bigger problems, until we reach the final solution. The answer to the original problem is always stored in the last cell of the DP table.

A diagram of a computer hardware process

AI-generated content may be incorrect.

*Fig 3: Tabulation Data block diagram*

1. **Program**

My solution works by splitting the card values into groups that are as equal as possible. In order to achieve this, I use a **subset-sum dynamic programming algorithm with tabulation (bottom-up)**. This method builds a DP table of reachable sums and uses the recurrence relation , which means a sum is possible if it was already possible before or if it can be formed by adding the current card value to a smaller sum. In this way, the algorithm finds the subset of cards whose total is closest to a target value (for example, half or a third of the deck’s total). Depending on whether there are 1, 2, or 3 grandchildren, the program applies this DP routine to divide the deck fairly while following the rules about favorites and discards. For the extra credit, the same tabulation idea is extended to handle up to grandchildren by repeatedly running the DP step on the remaining cards until all groups are assigned.

A screenshot of a computer program

AI-generated content may be incorrect.

*Fig 4: Pseudocode for subset-sum (Tabulation)*

The time complexity of the subset-sum tabulation is O(n·T), where *n* is the number of cards and *T* is the target sum (typically half or one-third of the total value). The space complexity is O(T) because we only store which sums are achievable up to the target. This keeps memory usage manageable even when the number of cards grows into the thousands, making the approach scalable for the experiment sizes in this assignment.

The core DP recurrence used in the pseudocode is:

This means that if a sum *s* is already achievable, then adding a new card value *v* allows us to achieve a new sum *s+v*, provided it does not exceed the target. This recurrence avoids recomputing and is what makes tabulation efficient.

**Main Function (PDC\_ca3)**

A screenshot of a computer program

AI-generated content may be incorrect.

*Fig 5: Pseudocode for Main function*

The pseudocode outlines how Grandma Rosa’s cards are distributed using dynamic programming while respecting fairness and special rules. First, if a random seed is given, it ensures reproducibility of the random choice of grandchildren. The program then identifies the favorite grandchild (based on initials i.e. Melaine) and, if three grandchildren are present, also determines a second favorite. A random scenario of G grandchildren is chosen, and allocations are prepared.

If **G=1**, the single grandchild simply receives all cards. For **G=2**, the deck is split into two nearly equal piles using subset-sum DP. If the favorite is included, they automatically receive the higher-valued pile if not, the program forces exact equality between the two piles, either by discarding a subset from the larger pile or, if that fails, by a global fallback that guarantees equality with some cards discarded. For **G=3,** the deck is divided into three near-equal groups via two DP subset-sum calls.

If the favorite is in the scenario, priority order is favorite → second favorite → remaining, and the largest pile goes to the favorite. If the favorite is absent, it follows the listed order of the chosen grandchildren. This combination of rules enforces the fairness policy favorites get priority, absent favorites trigger strict equality, and all cases use DP-based splitting for balanced distribution.

**Extra Credit (PDC\_ec\_ca3)**

**A screenshot of a computer code

AI-generated content may be incorrect.**

*Fig 6: Extra credit function*

The function PDC\_ec\_ca3 generalizes the card distribution problem to any number of grandchildren (G ≥ 1) without considering favorites. It starts by tracking all remaining card indices and creating empty allocations for each grandchild. For each grandchild in sequence, the algorithm calculates the total value of the remaining cards and divides it by the number of groups still left, giving a target that represents an equal share. Using the subset-sum tabulation algorithm, it then selects a subset of cards whose total is as close as possible to this target and assigns those cards to the current grandchild. The selected indices are removed from the pool of remaining cards, and the process continues until all grandchildren have received their share or no cards remain. This repeated use of subset-sum ensures that the distribution is balanced and fair across all grandchildren.

The time complexity of PDC\_ec\_ca3 is O (G · n · T), where n is the number of cards and T is the average target value in each round (approximately sum(A)/G). Each iteration applies the subset-sum tabulation on the remaining cards, and this is repeated G times. The space complexity is O(T), since the subset-sum routine only needs to keep track of achievable sums up to the current target. In practice, this design makes the approach efficient for the assignment’s input sizes and allows it to scale reasonably well as both n and G grow.

1. **Experimental Design**

Examining how my algorithm performs for the case with three grandchildren, as well as the extra credit function for larger numbers of grandchildren, these experiments evaluate how time and space usage change depending on the input values provided. All experiments were conducted on the same computing environment described earlier.

* 1. **Experiment 1: Main Program**

The first experiment, conducted with the main program, was used to test the time efficiency and space efficiency of the algorithm across different values of n (number of cards) and G (number of grandchildren), while keeping in mind the distribution rules. The experiments were carried out for deck sizes ranging from 1 to 4096 and for different distributions of grandchildren (G = 1, 2, 3). This helped us evaluate how well the algorithm scales with increasing input size and how effectively the rules are followed for varying numbers of grandchildren. The metrics recorded during this experiment were: n (the number of cards in the deck), G (the number of grandchildren involved in the distribution), time\_sec (the runtime of the algorithm in seconds, measuring computational speed), and peak\_kb (the peak memory used during execution in kilobytes, measuring storage space).

* 1. **Experiment 2: Extra Credit**

The second experiment extends the analysis through the extra credit function, which generalizes the distribution to any number of grandchildren (G ≥ 1). The purpose is to evaluate how the algorithm performs in terms of time efficiency and space efficiency when both the deck size (n) and the number of grandchildren (G) vary. This experiment was conducted for deck sizes ranging from 1 to 4096 and for different values of G up to 16, thereby testing the scalability of the algorithm beyond the original constraint of three grandchildren. By increasing G beyond 3, this experiment demonstrates how the solution adapts to more complex distribution scenarios, while still aiming to balance the card values fairly among grandchildren. The metrics recorded during this experiment were the same as in the first: n (deck size), G (number of grandchildren), time\_sec (runtime of the algorithm in seconds), and peak\_kb (peak memory usage in kilobytes). These results highlight both the scalability and the limitations of the algorithm when handling larger and more flexible problem sizes.

1. **Results**

The experiments were carried out according to the defined metrics, and the results were plotted in graphs to visualize how the algorithm adapts and responds under different conditions.

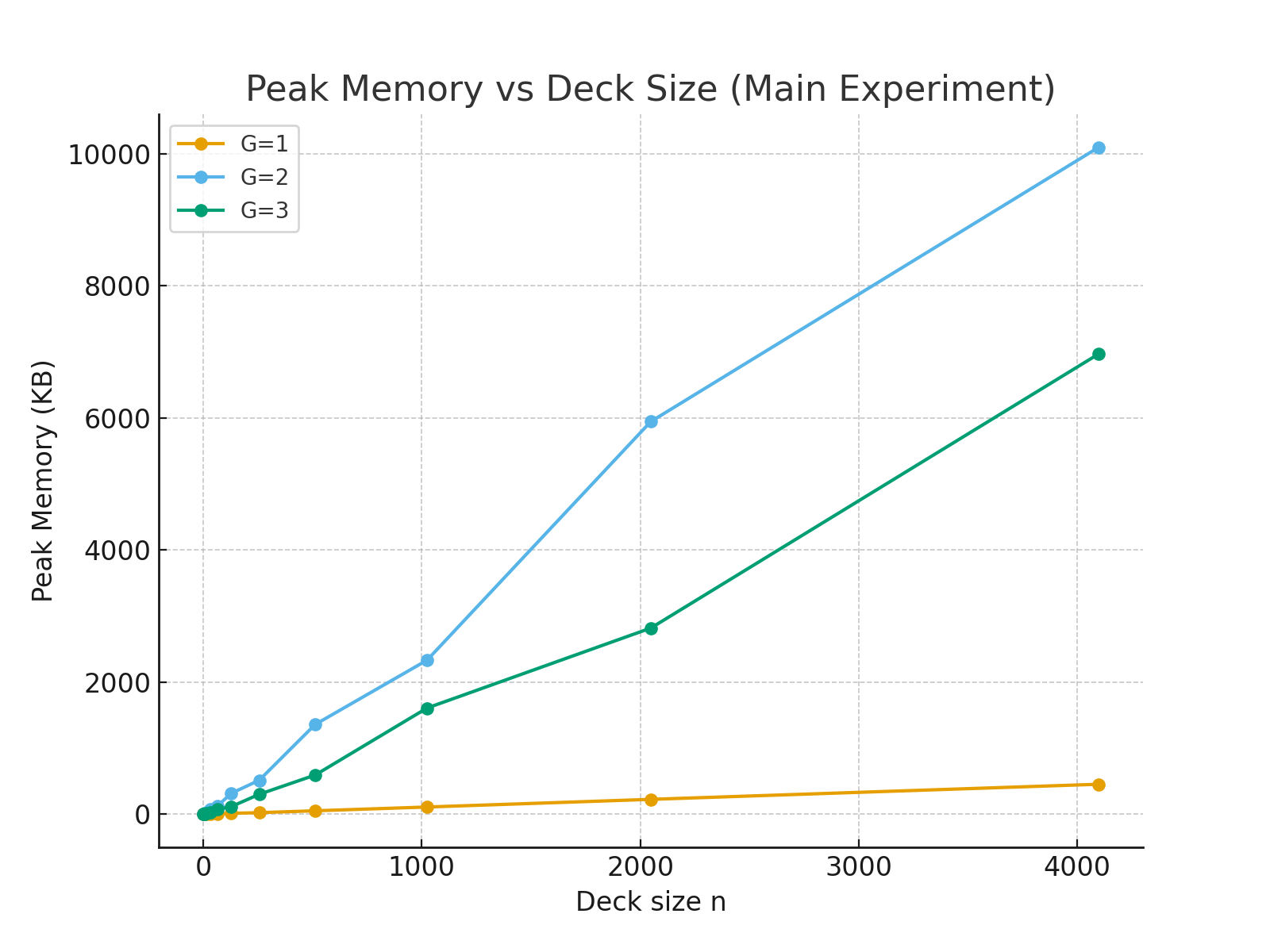
* 1. **Experiment 1: Main Program**

**A graph with different colored lines

AI-generated content may be incorrect.**

*Fig 7: Graph for runtime as deck size increases*

In the runtime graph, execution time is negligible for G = 1 but increases sharply for G = 2 and G = 3 as deck size grows, with G = 2 being the most expensive because of the need for precise equalization. Overall, the results confirm that the algorithm is trivial for G = 1 but scales poorly in both time and space for G = 2 and G = 3 due to the computational cost of subset-sum.



*Fig 8: Graph for Memory as deck size increases*

In the peak memory graph, memory usage stays minimal for G = 1 since all cards are directly assigned, while G = 2 shows the steepest growth due to the dynamic programming subset-sum used for exact balancing, and G = 3 grows moderately as it uses repeated subset-sum partitions

* 1. **Experiment 2: Extra Credit**

**A graph of different colored lines

AI-generated content may be incorrect.**

*Fig 9: Graph for memory with deck size for various G*

This graph, representing the extra credit experiment, shows how memory usage changes as the deck size increases for up to 16 grandchildren. As expected, memory consumption grows with deck size, since larger decks require storing more states and subsets during dynamic programming partitioning. The case of G=1 stands out, consuming the largest amount of memory for larger decks. This occurs because, with only one grandchild, the algorithm processes the entire deck through tabulation without splitting. For G=2 and G=3, the memory usage is moderate, but as the value of G increases further, the memory demand decreases. This reduction happens because the partitioning work is distributed across more groups, and each group receives fewer cards, leading to smaller state spaces and more efficient memory usage.

**A graph of different colored lines

AI-generated content may be incorrect.**

*Fig 10: Graph for runtime with deck size for various G*

This figure illustrates the runtime performance of the extra credit algorithm as the deck size increases for values of G ranging from 1 to 16. As expected, runtime grows with larger deck sizes because more cards require additional processing during dynamic programming partitioning. Like the memory results, G=1 takes the longest time, since the algorithm processes the entire deck without splitting it among grandchildren. G=2 and G=3 also show higher runtimes compared to larger G, as the deck is still divided into relatively large partitions. However, as G increases beyond 3, runtime becomes more efficient, with significantly lower growth. This occurs because the workload is distributed across more grandchildren, meaning each group receives fewer cards, leading to smaller subset-sum computations and faster execution overall.

1. **Conclusion**

The experiments for both the main program and the extra credit extension show that the algorithm achieves fairness while scaling reasonably with deck size and number of grandchildren. In the main program, distributing to one to three grandchildren highlighted that exact equalization with two grandchildren is most demanding, while one and three scale more smoothly. In the extra credit experiments, extending up to sixteen grandchildren showed that runtime and memory rise with larger decks but the workload is spread more evenly across groups.

My approach is effective, but the code could be improved in clarity, formatting, and reusability. Functions can be refactored for better design, and experiments could be expanded to test more edge cases. Overall, I am confident in the correctness of the analysis and the algorithm’s ability to solve the problem, while acknowledging opportunities to refine style, structure, and experimentation.