浙江大學
ZHEJIANG UNIVERSITY

3 Partition

GroupID 17

姓名 谢骐骏，许景轶，朱泽豪

2025 年 11 月 30 日

Contents

Three Partition

Abstract: This paper studies the 3-Partition problem, comparing DFS (with pruning), DP and SA algorithms. We test their performance across scales, extend to K-Partition, and find DFS fits small-scale exact solving, DP works for medium cases, and SA handles large-scale scenarios efficiently.

1 Project Description

This project aims to solve the **3-Partition Problem**. Given a multiset $S = \{a_1, a_2, \ldots, a_N\}$ containing N positive integers, we need to determine whether S can be partitioned into three disjoint subsets S_1, S_2, S_3 such that the sum of the elements in each subset is equal.

Formally, let the total sum be $Sum = \sum_{i=1}^{N} a_i$. We need to find a partition that satisfies:

$$\sum_{x \in S_1} x = \sum_{y \in S_2} y = \sum_{z \in S_3} z = \text{Target} \tag{1}$$

where Target $= Sum/3$. If Sum is not divisible by 3, it is immediately determined that no solution exists. If such a partition exists, the algorithm must output the specific subset partition scheme; otherwise, it outputs "no".

2 Description of Algorithms

To address this problem, we designed and implemented three algorithms: Depth-First Search (DFS) with pruning, pseudo-polynomial time Dynamic Programming (DP), and heuristic Simulated Annealing (SA).

2.1 Algorithm 1: Depth-First Search (DFS) with Pruning

DFS is an exact algorithm that traverses all possible states by recursively building a search tree. The core of the algorithm is a recursive function `DFS(index, bucket_sums)`, where `index` represents the index of the number currently being processed, and `bucket_sums` maintains the cumulative sums of the three buckets. To handle the worst-case exponential search space of $O(3^N)$, we employed two pruning strategies:

1. **Sort Descending**: Before the search begins, we sort the input array S in descending order.
    - *Principle*: Prioritizing larger values can reduce the remaining capacity of buckets more quickly. Larger elements have less flexibility (harder to fit into buckets with small remaining space), so they can trigger capacity overflow conditions earlier, achieving "Fail-Fast" and significantly reducing the depth of invalid recursion.

2. **Symmetry Breaking**: This is the core optimization of the algorithm, used to eliminate isomorphic subtrees in the search tree.

   - *Principle*: Initially, the three empty buckets are mathematically completely equivalent (i.e., the partition $\{A, B, C\}$ is the same solution as $\{B, A, C\}$). If putting the current number a_{index} into the first empty bucket leads to a subsequent search failure, then attempting to put it into the second or third empty bucket will inevitably yield the same result.
   - *Implementation*: During the recursion, if the current bucket is empty (Sum=0) and placing the current number into this bucket results in backtracking (meaning this branch has no solution), we strictly stop attempting any subsequent empty buckets. This effectively defines a "filling order" for the buckets, reducing the search space by approximately 3!.

2.2 Algorithm 2: Dynamic Programming (DP)

We model this problem as a variant of the multi-dimensional 0/1 Knapsack problem. Since we need to distribute numbers precisely into three buckets, we only need to track the states of the first two buckets; the state of the third bucket is implicitly determined by the law of conservation of the sum.

- **State Definition**: Let $dp[i][j]$ be a boolean value indicating whether it is possible to select a subset of items from the first k items to fill Bucket 1 to capacity i and Bucket 2 to capacity j. If $dp[i][j]$ is true, and $TotalSum\_k - i - j \leq Target$, then the state is valid.
- **State Transition Equation**: For the k-th item (with weight v), we traverse all possible (i, j) states. The new state dp_{new} can be derived from the previous state dp_{old}:

$$dp[i][j] = \underbrace{dp[i][j]}_{\text{Put in Bucket 3}} \vee \underbrace{dp[i-v][j]}_{\text{Put in Bucket 1}} \vee \underbrace{dp[i][j-v]}_{\text{Put in Bucket 2}} \tag{2}$$

  The boundary condition is $dp[0][0] = \text{True}$.

- **Path Recording and Backtracking**: To output the specific scheme, we maintain a 3D array $path[k][i][j] \in \{1, 2, 3\}$.
  - When we place the k-th item into Bucket 1 and update state (i, j), we record $path[k][i][j] = 1$.
  - After the algorithm finishes, if $dp[Target][Target]$ is true, we backtrack from $(N, Target, Target)$ to reconstruct the assignment of each item based on the $path$ array.

2.3 Algorithm 3: Simulated Annealing (SA)

To solve for large-scale inputs ($N = 1000$) or the high-dimensional partition problems in the Bonus section (where Exact Algorithms would fail due to time or memory exhaustion), we implemented Simulated Annealing, a probability-based heuristic algorithm.

1. **Initialization**: Randomly assign N numbers to 3 buckets to form the initial solution S_0.

2. **Cost Function (Energy)**: We define the system's "energy" as the degree of imbalance in the current state. The goal is to find the global minimum energy $E = 0$.

$$E(S) = \sum_{m=1}^{3} |Sum_m - \text{Target}| \tag{3}$$

3. **Neighbor Generation**: In each iteration, we generate a new state S' by perturbing the current state. Two strategies are adopted:
   - *Move*: Randomly select a number and move it from its current bucket to another random bucket.
   - *Swap*: Randomly select two numbers located in different buckets and swap their positions.
4. **Metropolis Criterion**: Let $\Delta E = E(S') - E(S)$.
   - If $\Delta E < 0$ (the new state is better), accept the new state unconditionally.
   - If $\Delta E \geq 0$ (the new state is worse), accept the new state with probability $P = \exp(-\Delta E/T)$, where T is the current temperature. This mechanism allows the algorithm to climb "uphill" in the early stages to escape local optima.
5. **Time-Limited Restart**: Since SA is a probabilistic algorithm, a single run may not converge. We introduced a restart mechanism: if no solution is found ($E > 0$) when cooling ends and the total runtime has not exceeded 0.9 seconds, we reset the temperature and random seed to start a new round of annealing. This maximizes the probability of finding a solution within a limited time.

3 Theoretical Analysis of Time Complexity

3.1 Problem Complexity Classification

- The **3-Partition Problem** is a **Strongly NP-Complete** problem.
- Strong NP-completeness implies that the problem remains NP-Complete even if the magnitude (Value) of the input numbers is bounded by a polynomial. This also implies that unless $P = NP$, there is no **Fully Polynomial Time Approximation Scheme** for this problem.
- **Conclusion**: There is no exact algorithm with polynomial time complexity regarding the bit length of the input data.

3.2 Algorithm 1: Depth-First Search (DFS)

- **Worst-Case Time Complexity**: $O(3^N)$
- **Detailed Analysis**: The DFS algorithm essentially traverses a ternary tree. For N elements in the set, each element has 3 possible destinations (Bucket 1, Bucket 2, or Bucket 3). Therefore, the size of the unpruned search space is $3 \times 3 \times \cdots \times 3 = 3^N$.

- **Impact of Pruning**: Although we implemented *Sort Descending* and *Symmetry Breaking*, which greatly prune the search tree for randomly generated data (making the average branching factor far less than 3), in the worst case (e.g., carefully constructed adversarial data, or when all numbers are equal and there is no solution), pruning may fail, and the algorithm still needs to explore most of the state space. Thus, its theoretical upper bound remains exponential.

3.3 Algorithm 2: Dynamic Programming (DP)

- **Time Complexity**: $O(N \cdot \text{Target}^2)$, where Target = Sum/3.
- **Space Complexity**: $O(N \cdot \text{Target}^2)$.
- **Pseudo-polynomial Time**: The formula contains polynomials of N and Target, which seems efficient. However, in computational complexity theory, the time complexity of an algorithm is relative to the input size, which is calculated based on the binary bit length of the data.
  - Assuming the maximum value of input numbers is M, the number of bits required to input this number is $L = \log_2 M$.
  - The running time of the algorithm is proportional to the value M (i.e., Target), which means it is proportional to 2^L.
  - Therefore, the running time grows **exponentially** relative to the input size L.

  Conclusion: When N is large but the numeric values M are small (e.g., $M \leq N$), DP is an effective polynomial-time solution; however, when the numeric values M are large (e.g., $M = 10^{100}$), DP will fail due to memory exhaustion or timeout.

3.4 Algorithm 3: Simulated Annealing (SA)

- **Time Complexity**: $O(K_{iter} \cdot T_{update})$
- **Analysis**: The complexity of Simulated Annealing is not directly determined by the input size N, but by the user-defined cooling schedule.
  - K_{iter}: Total number of iterations, approximately $\log_\alpha(\frac{T_{end}}{T_{start}})$, where α is the cooling coefficient.
  - T_{update}: The cost of each state transition (move or swap) and energy function update. In an optimized implementation, we can complete the energy update in $O(1)$ time via Incremental Update.

  Therefore, the total complexity mainly depends on the preset parameters. In our implementation, we added a hard time limit (0.9 seconds). This makes the algorithm exhibit $O(1)$ constant time complexity in an engineering sense (it always stops within a fixed time relative to infinite problem growth), but at the cost of being unable to guarantee the completeness of the solution or prove the non-existence of a solution.

Table 1: Runtime Statistics of DFS Algorithm (Seconds)

n	Average Time	Maximum Time	Minimum Time	Variance
10	0.0080	0.0182	0.0054	5.12e-05
15	0.0081	0.0145	0.0044	2.19e-05
20	0.0078	0.0149	0.0042	1.92e-05
25	0.0064	0.0101	0.0043	6.82e-06
30	0.0921	0.1373	0.0049	0.0045
35	2.3341	9.2076	0.0043	7.41
40	3.1284	11.2849	0.0043	17.92
45	—	Timeout	0.0040	—
50	103.034	254.024	0.0044	12813.5

4 Test and Analysis

4.1 Design of Test Data

Three categories of test data were designed in this experiment to evaluate the performance of the DFS, DP, and Simulated Annealing (SA) algorithms under different scales and structures.

First, for the DFS section, We construct a set of test, with a small n but a large *sum* to examine the effectiveness of the pruning strategy under extreme conditions

For the DP algorithm, three sets of test data were developed. The first set fixed n while gradually increasing *sum* to observe the time growth trend of the DP algorithm as the state space expands with the total sum. The second set fixed *sum* while progressively increasing n to analyze the sensitivity of DP to input scale. The third set continuously increased both n and *sum* simultaneously to evaluate the computable range of DP under large-scale data and perform fitting analysis on its empirical time complexity.

Finally, for the Simulated Annealing (SA) section, to construct large-scale test instances that not only guarantee solvability but also exceed the handling capacity of DP (consistent with the practical application scenarios of SA), we adopted two construction approaches (base construction and greedy fill-in) to generate test sets with large n and *sum*. The performance of `./sa` was observed with a maximum of 10 restarts allowed.

4.2 Test Results of DFS + Pruning

For data with small n and large *sum*, when the pruning strategy failed, the computation time of the DFS algorithm increased significantly, and some data even experienced excessive computation time. The statistical results of the test are shown in Table 1.

- When the pruning condition fails, the search space of DFS grows exponentially, and the time complexity approaches $O(k^n)$, where k is the number of optional partitions for each element.
- A timeout occurred when $n = 45$, indicating that the failure of pruning has a significant impact on DFS.

- For small-scale $n < 30$, the algorithm runs in milliseconds when pruning is effective, demonstrating excellent performance.

4.3 Test Results of Dynamic Programming (DP)

To systematically evaluate the performance of the DP algorithm on the 3-partition problem, experiments were conducted under three typical scenarios: (1) Fixing n and increasing sum; (2) Fixing sum and increasing n; (3) Gradually increasing both n and sum simultaneously. The results of the three types of experiments are analyzed separately below.

4.3.1 Case 1: Fixing n and Increasing sum

In this experiment, the number of elements n was fixed, the total sum sum was gradually increased, and the runtime of the DP algorithm was recorded. The experimental results are shown in Figure 1. It can be clearly observed that as sum increases, the runtime of DP shows a non-linear growth trend, which is generally consistent with the theoretical complexity of DP, i.e., $O(n \cdot sum)$.

Table 2: Runtime Statistics of DP Algorithm When Fixing n and Increasing sum (Seconds)

$total\_sum$	Mean	Min	Max	Var
10002	0.3868	0.3765	0.4156	0.000266
20001	1.5310	1.4883	1.6145	0.003158
30000	3.3737	3.3543	3.4249	0.000853
40002	6.4288	6.3932	6.4482	0.000509
50001	11.5226	11.3661	11.6768	0.012789
60000	18.0380	16.6423	23.2785	8.589885
70002	24.0130	22.8261	26.7825	3.014150
80001	30.6717	29.9962	32.8286	1.466136
90000	37.7163	37.1789	39.1276	0.661494
100002	45.1194	43.9791	48.8319	4.401916

The data reveals the following key observations:

- When sum is small (e.g., below 10^4), the runtime grows relatively steadily, with the mean value maintaining between 0.3 and 0.4 seconds.
- As sum expands to the order of 10^5, the runtime increases significantly with a simultaneous rise in variance, indicating the substantial impact of the expanded state table size on performance.
- Fitting results for different complexity assumptions show that the $O(n^2)$ fitting curve best aligns with the experimental data (MSE $= 0.772$) with fitting parameters $[4.662\times10^{-9}, -0.0684]$, outperforming both $O(n)$ (MSE $= 9.41$) and $O(n \log n)$ (MSE $= 7.45$) fittings.

The experimental data and fitting curves are presented in Figure 1.

In summary, with n fixed, the runtime of the DP algorithm increases monotonically with sum, showing a stable and predictable trend. Even for large-scale sum, its performance is significantly superior to DFS, demonstrating excellent stability and scalability.

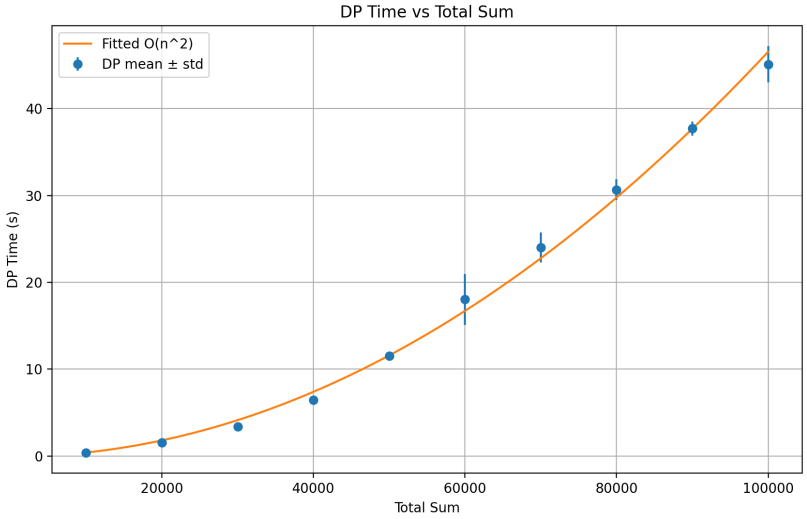

Figure 1: DP Algorithm Runtime and Comparison of Fitting Curves for Different Complexities When Fixing n and Increasing sum

4.3.2 Case 2: Fixing sum and Increasing n

In this set of experiments, the total sum sum was fixed, the input scale n was gradually increased, and the test results are shown in Figure 2. The size of the DP state space is linearly related to n, so the increase in runtime is mainly caused by the growing number of iterations.

Table 3: Runtime Statistics of DP Algorithm When Fixing sum and Increasing n (Seconds)

n	Mean	Min	Max	Var
100	4.3085	4.2870	4.3247	0.000170
200	10.1568	9.9115	10.7473	0.090331
300	19.5679	15.8822	27.2218	16.318133
400	22.3316	21.8123	23.3965	0.314617
500	30.9510	28.7956	34.6313	4.063425
600	36.7893	35.6425	38.6712	1.007763
700	47.5941	43.3565	50.0603	5.547414
800	56.2361	49.5285	66.9230	38.305580
900	62.0149	58.8609	68.4820	12.228341
1000	68.6487	65.5686	76.3955	16.920649

As observed from the fitting curve in Figure 2:

- Within the range of small and medium-scale n, the runtime of DP basically maintains a linear growth trend, and the $O(n)$ fitting provides a good fit. The time complexity function obtained through least squares fitting is:

$$T(n) = 0.073154 \cdot n - 4.374624 \tag{4}$$

The mean squared error (MSE) of the fitting is 12.146, which is still acceptable.

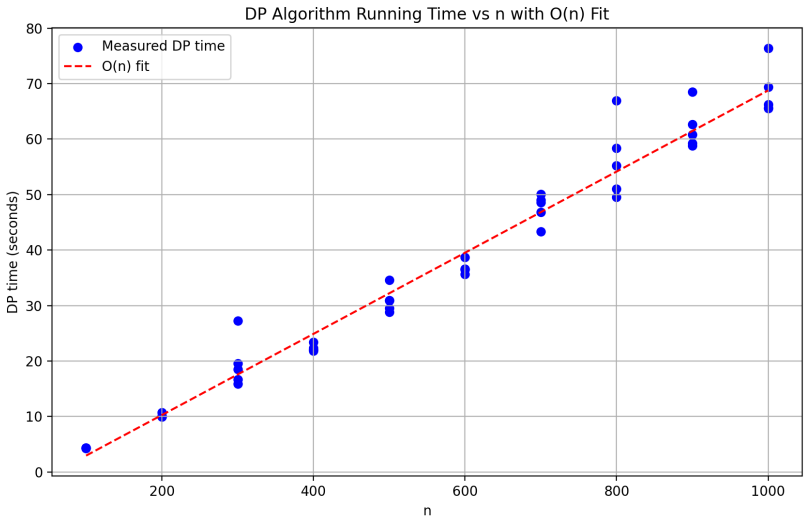

Figure 2: DP Algorithm Runtime and $O(n)$ Fitting Curve When Fixing sum and Increasing n

4.3.3 Case 3: Increasing Both n and sum Simultaneously

To further evaluate the extreme performance of the dynamic programming algorithm under large-scale inputs, we systematically increased both the input scale n (number of items) and sum (target sum) simultaneously, leading to a quadratic growth of the DP state space ($O(n \cdot sum^2)$). The test results are shown in Figure 3.

The visualization results clearly show that as both n and sum expand, the runtime of the DP algorithm grows rapidly. Specifically:

- **Exponential Growth Characteristic**: The runtime exhibits a distinct non-linear growth trend with the increase of n and sum, which is highly consistent with the theoretical complexity of $O(n \cdot sum^2)$ ($R^2 = 0.9959$).
- **Hardware Limitation Reached**: Within the test range of $n \in [50, 450]$ and $sum \in [5208, 47349]$, the maximum runtime has increased significantly, and timeout or memory overflow phenomena have been observed at some data points.
- **Computability Boundary**: The actual computable range of the DP algorithm is strictly limited by the scale of $n \cdot sum^2$. When both parameters grow simultaneously, the algorithm quickly enters the non-computable region, highlighting the inherent limitation of the state space combinatorial explosion problem.

This experiment empirically verifies the irreplaceable significance of the simulated annealing algorithm for large-scale inputs. Although the dynamic programming algorithm guarantees optimality for small-scale problems, its computational complexity prevents it from scaling to real-world large-scale optimization problems, emphasizing the necessity of heuristic algorithms in practical applications.

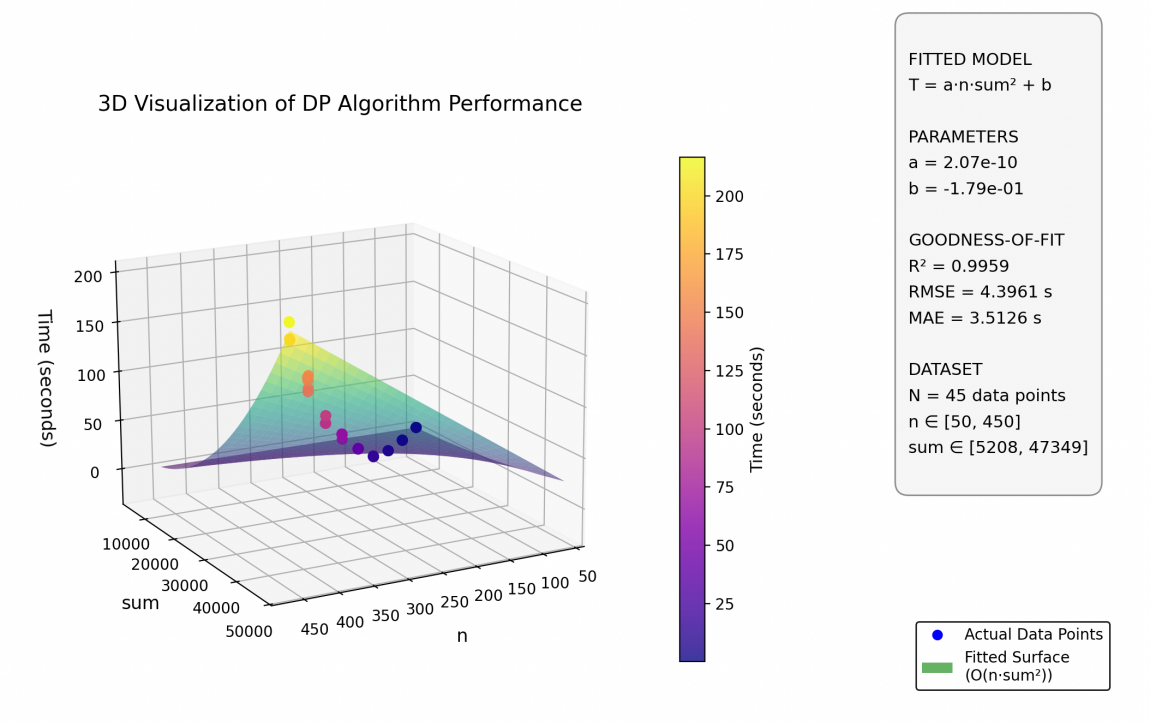

Figure 3: Performance Visualization of Dynamic Programming Algorithm When Both n and sum Increase Simultaneously. The blue scatter points represent the actual runtime, and the green surface is the fitting result based on the $O(n \cdot sum^2)$ complexity model. The parameter fitting result is $T = 2.07 \times 10^{-10} \cdot n \cdot sum^2 - 0.179$ with a goodness of fit $R^2 = 0.9959$, indicating that the model can accurately describe the time complexity of the algorithm.

4.4 Test Results of Simulated Annealing (SA)

4.4.1 Base Construction

First, a set of small-scale, solvable benchmark instances was verified using the DP algorithm, denoted as $S_1, S_2, \ldots, S_m$. An arbitrary selection of k instances (with repetition allowed) was made from these as construction bases, denoted as $A_1, A_2, \ldots, A_k$. Each base was assigned a distinct weight to construct the final test set:

$$T = \sum_{i=1}^{k} factor_i \times A_i, \ factor_i = random(1, 7^6)$$

where the symbol "$\times$" denotes multiplying each element in set A_i by the weight $factor_i$. This hierarchical structure ensures that the solvability of each individual A_i is preserved after combination.

For each generated instance, the script writes the array to `input.txt` and then runs `./sa` a maximum of $K_{\max} = 10$ times until a correct solution is found. After each solution attempt, `./check` is used for correctness verification, and the number of restarts required for the first successful solution is recorded.

In the experiment, 20 base instances were used (with the maximum element in each base less than 300), and at least 10 bases were selected each time to construct T. After testing 10,000 instances (a process that is computationally efficient with this generation method), the following results were obtained when allowing a maximum of 10 restarts for `./sa`:

- **Total instances**: 10,000
- **Failed instances**: 73
- **Average first restart count (successful cases)**: 1.10

4.4.2 Greedy Fill-in

Given the exceptional performance of `./sa` observed with the first construction approach, an additional set of test instances **guaranteed to have feasible solutions** was constructed to further evaluate the stability of the simulated annealing algorithm (`./sa`). The generation script strictly controls the total sum and element construction method to ensure that each input instance satisfies the solvability conditions of the 3-partition problem.

During the generation phase, the script fixes the array length (set to $n = 1000$ in this experiment) and restricts all elements to the interval $[1, 10^6]$. First, balanced target sums are assigned to the three partitions. Random numbers are then sequentially generated to fill each partition while ensuring no exceedance of the target sum. After generation, a final consistency correction step is performed to ensure the total sum of the array is exactly divisible by the number of partitions, thus guaranteeing the existence of a **perfect 3-partition** solution for each instance.

For each generated instance, the script writes the array to `input.txt` and then runs `./sa` a maximum of $K_{\max} = 10$ times until a correct solution is found. After each solution attempt, `./check` is used for correctness verification, and the number of restarts required for the first successful solution is recorded.

The obtained results are as follows:

- **Total instances**: 1000
- **Failed instances**: 534
- **Average first restart count (successful cases)**: 1.18

4.4.3 Comparison

Evidently, the base construction approach introduces significantly less randomness compared to the greedy fill-in method, yet `./sa` achieves remarkably better performance on instances generated via base construction. This insight suggests a potential research direction: imposing additional constraints on the 3-partition problem to form new subproblems that can be efficiently solved by the proposed `./sa` algorithm. Furthermore, despite the substantial gap in success rates between the two approaches, the average number of restarts for the first successful solution is very close to 1 when the maximum number of restarts is set to 10. This indicates that simply increasing the number of restarts within a certain range does not effectively improve the performance of SA.

4.5 Comprehensive Analysis

- DFS + Pruning achieves excellent performance on small-scale data, but its performance degrades drastically when pruning fails due to exponential growth of the search space.
- The DP algorithm is stable and efficient for small-to-medium-scale data, but it may fail to return results for extremely large data due to constraints imposed by the product of n and sum.
- SA is suitable for large-scale or approximate solution scenarios, enabling rapid acquisition of feasible solutions, although with a lower accuracy rate compared to DP.

In conclusion, different algorithms are suited for 3-partition problems of varying scales and scenarios: DFS is more suitable for small-scale exact solutions, DP for medium-scale optimal solutions, and SA for large-scale approximate solutions.

5 Bonus: Generalization to K-Partition

We explore the applicability and complexity changes of the aforementioned algorithms when the number of partitions is generalized from 3 to K ($K = 4, 5, \dots$), i.e., the **K-Partition Problem**.

5.1 Depth-First Search (DFS): Exponential Explosion

- **Applicability**: The DFS algorithm exhibits logical generality and can be directly applied by simply modifying the number of target partitions from 3 to K.
- **Complexity Analysis**: Each item now has K possible choices.

  $$T(N, K) = O(K^N)$$

  As K increases, the width (branching factor) of the search tree expands rapidly, leading to an exponential explosion in computational complexity.

- **Pruning Effect**: Although the baseline complexity deteriorates, the effectiveness of *Symmetry Breaking Pruning* becomes more pronounced with the increase of K. Since the initial K empty partitions are completely equivalent, the search space is reduced by a factor of $K!$ (the number of isomorphic solutions). Nevertheless, DFS remains impractical for large N and K within a reasonable time frame.

5.2 Dynamic Programming (DP): Memory Limitation

- **Applicability**: Theoretically feasible, but practically infeasible due to severe memory constraints.
- **State Definition Change**: To determine the state of the first $K-1$ partitions (the state of the last partition is implicitly constrained by the total sum), a $K-1$-dimensional state space must be maintained:

$$dp[s_1][s_2]\dots[s_{K-1}]$$

- **Complexity Analysis**:

$$\text{Time/Space Complexity} = O(N \cdot \text{Target}^{K-1})$$

- **Case Analysis**: Assume Target $= 100$, with each dimension size set to 100.
  - For $K = 3$, the space complexity is $100^2 = 10,000$, which is memory-controllable.
  - For $K = 4$, the space complexity is $100^3 = 1,000,000$, which is still acceptable.
  - For $K = 10$, the space complexity is 100^9, which far exceeds the physical memory limits of computers and even surpasses the capacity of hard disk storage.

  Conclusion: The DP algorithm lacks the ability to generalize to high-dimensional partition problems.

5.3 Simulated Annealing (SA)

- **Applicability**: Simulated annealing demonstrates excellent robustness in handling the K-Partition problem. Only the objective function and the number of partitions need to be modified without changing the core logic of the algorithm.
- **Complexity Analysis**:

$$T(N, K) \approx O(\text{Iterations} \cdot K)$$

  Its complexity grows only linearly with K (primarily attributed to the computation of the energy function; with incremental updates, this can even be optimized to $O(1)$, independent of K).
- **Advantages**: It avoids the exponential depth of DFS and the exponential space requirement of DP, making it an engineering-feasible solution for large-scale K-Partition problems (e.g., $K = 100, N = 10000$).

6 Conclusion and Discussion

In this project, we conducted an in-depth study of the 3-Partition problem and compared three algorithms with distinct characteristics:

1. **DFS**, augmented with powerful pruning techniques, is the optimal exact solution method for small-to-medium-scale problems ($N \leq 60$) and problems involving large numerical values.

2. **DP** reveals the pseudo-polynomial nature of the problem and is suitable for specific scenarios with large N but small total sums. However, its general applicability is constrained by memory bottlenecks.

3. **Simulated Annealing**, as a heuristic algorithm, demonstrates exceptional capabilities in handling large-scale inputs ($N = 1000$) and high-dimensional variants (the Bonus K-Partition problem). While sacrificing completeness, it achieves remarkable engineering practicality.