Advanced Data Structures and Algorithms Analysis

Project 5: Three Partition Problem

Author: 林润铎，窦宇浩，张钊

Date: 2025/11/30

2025-2026 Autumn & Winter Semester

Table of Contents

Chapter 1: Introduction

You have a very simple task: just partition a series of numbers into three parts and make their sum is the same.

More formally, you will first get a positive integer N indicating that there will be totally N numbers as follow. And then you will get N positive integers a_1, a_2, ⋯, a_n, you should calculate whether there is a three - partition that can partition them into three parts and their sum is exactly the same, if the answer is yes, you also need to output the 3 parts in 3 lines.

bonus: If partition these numbers into 4,5,⋯,k(k is a given number but not a input) parts, do you r above different types of algorithm also work? If not work, can you try some new type of algorithm to solve it? If work, Can you analyze the time complexity of them?

The text above is copied from the pintia.

1.1. Implementation ideas

We came up with three ideas to solve this problem, which are described in the following three chapters. Now let's briefly introduce these three ideas.

Our first thought is of course to use backtracking to try all possible combinations of numbers to see if we can find a valid partition(describe in Chapter 2). Then we can tried any pruning to reduce lookback time.

But we are not satisfied with this, this problem obviously has some characteristics that are applicable to dynamic programming. A very natural idea is to start with the case of k=2 first, which is two partition problem. The target subset sum is half of the total sum. As long as the total sum can be divided by 2, then if we can find a subset of numbers that sum to the target, the rest of the numbers must also sum to the target. This is a classic 0-1 knapsack problem, which can be solved by dynamic programming.

When we tried to extend this idea to k=3 or even bigger, we found that it is not a easy way to directly use knapsack problem to solve it. If we still find a subset with one thired of the total sum, then the rest of the numbers may cannot be solved as a two partition problem.

However, based on the above idea, we can still tried to use a higher-dimensional dynamic programming to solve it(describe in Chapter 4). But this algorithm depends on the size of the target subset sum and is difficult to extend to the k partitions problem, so we looked for another dynamic programming solution.

It's called Bitmask Dynamic Programming(describe in Chapter 3). The idea is to use a bitmask to represent which numbers have been used, and use the sum of the current bucket to represent the state.

Chapter 2: Backtracking with Pruning

2.1. Algorithm Description

The problem requires partitioning a series of N integers into $K = 3$ subsets such that the sum of each subset is equal. This is a specific instance of the Partition Problem (specifically the 3-Partition problem), which is known to be NP-Complete.

To solve this efficiently, we implemented a Recursive Backtracking (Depth-First Search) algorithm using a Sequential Bucket Filling strategy. Unlike the naive approach which tries to assign each number to a bucket, our algorithm attempts to completely fill one bucket to the `target_sum` before moving on to the next. This significantly reduces the state space by enforcing an order of completion.

2.2. Key Pseudocode

The core logic is encapsulated in the recursive `backtrack` function. The function focuses on filling the k-th bucket using available (unused) numbers.

```
// Global: numbers (sorted descending), used[], target_sum
Function Backtrack(k, current_bucket_sum, start_index):
    // Base Case: Last bucket (k=1)
    // If K-1 buckets are filled correctly, the remaining items
    // must form the last valid bucket.
    IF k == 1:
        Mark all unused items as belonging to Bucket 1
        RETURN TRUE

    // Step: If current bucket is full, move to the next bucket (k-1)
    IF current_bucket_sum == target_sum:
        RETURN Backtrack(k - 1, 0, 0)

    // Iterate through numbers starting from start_index
    FOR i FROM start_index TO N-1:
        IF used[i] IS TRUE: CONTINUE

        // Pruning 1: Capacity Constraint
        IF current_bucket_sum + numbers[i] > target_sum:
            CONTINUE

        // Pruning 2: Avoid Duplicates (Local Symmetry)
        // If current number is same as previous and previous was skipped,
        // using this one would produce an identical permutation.
        IF i > start_index AND numbers[i] == numbers[i-1] AND used[i-1] IS FALSE:
            CONTINUE

        // Action: Select number i
        used[i] = TRUE
        bucket_id[i] = k

        // Recurse: Continue filling current bucket
        IF Backtrack(k, current_bucket_sum + numbers[i], i + 1) IS TRUE:
            RETURN TRUE

        // Backtrack: Deselect number i (Undo Action)
        used[i] = FALSE
        bucket_id[i] = 0

        // Strong Pruning: If we fail to fill the bucket starting with
        // the largest available number (bucket was empty), no solution exists.
        IF current_bucket_sum == 0:
            RETURN FALSE

    RETURN FALSE
```

2.3. Analysis of Algorithm Implementation

The implementation transforms a standard exponential search into an efficient solver through strictly ordered construction and pruning:

1. Sequential Subset Construction ($k, k-1 \ldots$): Instead of deciding which bucket an item belongs to, we decide which items belong to the current bucket. We fill Bucket 3, then Bucket 2. Bucket 1 is automatically determined by the remaining items.
   - **Optimization:** The check `if (k == 1) return true` effectively reduces the recursion depth by one full bucket level.
2. Heuristic Optimization (Descending Sort): The input array is sorted in descending order (`std::sort` with `compare`).
   - **Rationale:** We prioritize placing larger numbers. Large numbers differ significantly in size and are harder to fit into the `target_sum`. Processing them early allows the algorithm to fail fast if they cannot be accommodated, pruning invalid branches near the root of the recursion tree.
3. Duplicate Pruning (Local Symmetry): The condition `if (numbers[i] == numbers[i-1] && !used[i-1]) continue` handles cases with repeated numbers in the input.
   - **Rationale:** If we have two identical numbers (e.g., 5, 5) and we decided **not** to use the first '5' in the current slot, using the second '5' immediately after would result in a mathematically identical state. Skipping it avoids redundant computations.
4. Strong Pruning (Empty Bucket Failure): The line `if (current_bucket_sum == 0) return false` appears after the recursive step returns false.

- **Rationale:** If the recursion starts to fill a new (empty) bucket and iterates through valid candidates but fails to find a solution using the very first (largest available) candidate, then strictly no solution exists for the remaining numbers. The largest available number **must** go somewhere; if it cannot start the current bucket, it cannot start any subsequent equivalent bucket.

2.4. Complexity Analysis

2.4.1. Time Complexity

- Theoretical Worst-Case: $O(K \cdot 2^N)$ or similar exponential bounds depending on the specific partition analysis. The state space involves selecting subsets from N items.
- Actual Performance: The “Sequential Filling” strategy combined with the `used` array is generally faster than the “Item Assignment” strategy for partition problems because it reduces the effective branching factor. The constraints `target_sum` are hit much more frequently. For $Nle20s\,\mathrm{im}\,30$, this runs instantaneously.

2.4.2. Space Complexity

- Auxiliary Space: $O(N)$.
  - `numbers`, `used`, and `bucket_id` arrays take $O(N)$.
  - The recursion stack depth is bounded by N (in the worst case where we pick items one by one).
  - Thus, the space complexity is linear, $O(N)$.

2.5. Bonus: Generalization to K-Partition

2.5.1. Validity of the Algorithm

The algorithm implemented for the 3-Partition problem was generalized to the K-Partition problem. The core logic—Sequential Filling with Backtracking—applies universally.

2.5.2. Comparison and Implementation Changes

The generalized code maintains the exact structure of former one with parametric adjustments:

1. Parameterization: The hardcoded constant `3` is replaced by `K`.
   - The recursion starts with `backtrack(K, 0, 0)`.
   - The validation check uses `total_sum % K`.
   - Target calculation is `total_sum / K`.
2. Logic Consistency: The logic `if (k == 1)` remains a valid base case for any K. It asserts that if we have successfully partitioned the set into $K - 1$ valid subsets with sum `target_sum`, the constraints of the problem (total sum is $K \times$ target) guarantee that the remaining unassigned numbers sum exactly to `target_sum`.

3. Input Validation: The generalized code retains the check `if (numbers[0] > target_sum)`. This is even more critical for larger K, as `target_sum` decreases while the magnitude of individual numbers stays constant, increasing the likelihood of immediate impossibility.

2.5.3. Complexity Analysis

The complexity shifts relative to K:

- Time Complexity: The algorithm effectively solves $K - 1$ subset sum problems sequentially. As K increases, the depth of "bucket levels" increases, but the number of available items N decreases for subsequent buckets. The complexity is roughly dominated by the cost of finding the first few partitions.
- Scalability: While robust, the problem remains NP-Complete. If $K \approx N$ (buckets of size 1), it is trivial. If $K \approx \frac{N}{2}$ (pairs), it equates to matching. The hardest instances usually lie where K is small but > 2 (like 3-Partition or 4-Partition) with dense numbers.

2.5.4. Conclusion

The sequential backtracking approach is a highly optimized method for the partition problem. By focusing on filling one bucket at a time and employing strong pruning on the sorted input, it effectively eliminates redundant search paths that would plague a naive brute-force solution.

Chapter 3: Bitmask DP

3.1. Algorithm Description: Bitmask Dynamic Programming

The problem is modeled using Top-Down Dynamic Programming with Bitmasking (Memoization). Instead of exploring every permutation of numbers (as in Backtracking), this approach focuses on the **subset** of numbers used so far.

Since the order of numbers within a bucket or the order of buckets themselves does not matter for the final validity, the state can be uniquely represented by a Bitmask. A bitmask is an integer where the i-th bit is `1` if the i-th number has been used, and `0` otherwise. This allows us to cache results of subproblems to avoid redundant calculations.

3.2. Key Pseudocode

The core logic uses a recursive function `dp(mask, current_sum)` with a memoization table `memo`.

```
// Global: memo[2^N], target

Function DP(mask, current_sum):
    // Base Case: All N numbers used
    IF mask == (1 << N) - 1:
        RETURN TRUE

    // Check Memoization Table
    // (Note: current_sum is implicit from mask, so memo is keyed only by mask)
    IF memo[mask] IS VISITED:
        RETURN memo[mask]

    // Try adding every unused number
    FOR i FROM 0 TO N-1:
        IF i-th bit is NOT set in mask:
            IF current_sum + numbers[i] <= target:

                // State Transition & Modulo Arithmetic
                // If bucket fills up, modulo resets sum to 0 for the next bucket
                next_sum = (current_sum + numbers[i]) % target

                // Recursive Step
                IF DP(mask | (1 << i), next_sum) IS TRUE:
                    memo[mask] = TRUE
                    RETURN TRUE

    memo[mask] = FALSE
    RETURN FALSE
```

3.3. Analysis of Algorithm Implementation

The implementation utilizes several clever techniques to solve the problem efficiently for small N:

1. State Compression (Bitmasking): The set of used numbers is stored in a single integer `mask`. This allows $O(1)$ operations to check if a number is used (`mask & (1<<i)`) and to add a number to the set (`mask | (1<<i)`).
2. State Reduction (Memoization Optimization): Although the recursive function signature is `dp(mask, current_sum)`, the `memo` array only stores `memo[mask]`.
   - **Rationale:** For any given set of numbers (mask), their total sum is constant. Therefore, the `current_bucket_sum` is deterministically calculated as `(Sum of numbers in mask) % target`. This reduces the state space from $Nc \cdot 2^N$ to just 2^N.
3. Implicit Bucket Switching (Modulo Arithmetic): The line `int next_sum = (current_sum + numbers[i]) % target;` handles bucket management elegantly.When `current_sum + numbers[i] == target`, the new sum becomes `0`. This

implicitly signifies that the current bucket is closed and the algorithm immediately starts filling the next bucket. This removes the need for an explicit `bucket_index` parameter.

3.4. Complexity Analysis

3.4.1. Time Complexity

- Analysis: $O(Nc \cdot 2^N)$.
  - The total number of unique states (masks) is 2^N.
  - For each state, we iterate through N numbers to find the next transition.
  - Unlike Backtracking ($O(3^N)$ or $O(K^N)$), the base of the exponent is 2, and it does not grow with the number of partitions K.
- Constraint: This is highly efficient for $Nle20$ (approx 10^6 operations), but becomes computationally infeasible for $N > 25$.

3.4.2. Space Complexity

- Analysis: $O(2^N)$.
  - The `memo` array requires size 2^N integers.
  - For $N = 20$, this is roughly 4MB (manageable).
  - For $N = 30$, this would require gigabytes of RAM, causing a Memory Limit Exceeded (MLE) or allocation failure (handled in the code by `std::nothrow`).

3.5. Bonus: Generalization to K-Partition

3.5.1. Validity of the Algorithm

The Bitmask DP algorithm implemented for the 3-Partition problem remains completely valid for dividing numbers into K parts (where $K = 4, 5...$). The core mechanism relies on filling buckets sequentially and using `(current_sum + val) % target` to reset the sum when a bucket is filled. This logic is mathematically agnostic to the specific value of K, provided that the total sum is divisible by K.

3.5.2. Comparison and Implementation Changes

Comparing the generalized code with the original 3-partition code, the algorithmic structure remains identical. The changes are purely parametric:

1. Parameterization: All instances of the hardcoded constant `3` have been replaced by the variable (or macro) `K`.
   - Validation logic: `total_sum % 3` -> `total_sum % K`.
   - Target calculation: `target = total_sum / 3` -> `total_sum / K`.
2. Additional Optimization: The generalized code adds a strictly necessary check: `if (numbers[i] > target)`.

- **Analysis:** As K increases, `target` becomes smaller. It becomes increasingly likely that a single large number exceeds the target subset sum. Catching this before allocating the large 2^N memory block saves significant resources.

3. Logic Consistency: The recursive transitions and state definition (`mask`) do not require any changes. The base case `mask == (1 << n) - 1` inherently implies that all K buckets have been successfully filled.

3.5.3. Complexity Analysis

The complexity analysis for the generalized case yields an interesting result regarding the parameter K:

- Time Complexity: $O(Nc \cdot 2^N)$
  - The number of states remains 2^N (representing all subsets of numbers).
  - For each state, we iterate N times.
  - Independence from K: Unlike the backtracking approach, the time complexity here does **not** increase with K. In fact, a larger K results in a smaller `target`, which may trigger the constraint check (`current_sum + val <= target`) more often, potentially pruning invalid transitions slightly faster.
- Space Complexity: $O(2^N)$
  - The size of the `memo` array is determined solely by the number of elements N (2^N integers).
  - The number of partitions K does not affect the memory requirements of the DP table.

3.5.4. Conclusion

The Bitmask DP strategy is highly effective for generalizing to K partitions because its complexity is dominated entirely by N. As long as N remains small (e.g., $Nle20$), the algorithm performs consistently regardless of whether we are partitioning into 3 parts or 10 parts. However, the exponential dependency on N remains the hard limit for this approach.

Chapter 4: Three-dimensional DP

4.1. Algorithm Description

We first talk about 2-partition problem(for n numbers,sum=s),every subset has its target=sum/2.We just need to find one half of the set, and the other target is sum/2 too.So we can use 2-dimension dynamic programming.

We can define dp[i][j]:for first k numbers,whether it can get j, and dp[i][0]=true.The result is dp[n][s/2].We can use 0/1 knapsack problem to solve it.

For 3-partition, we can still use similar way of thinking, But a wrong way is to use 0/1 knapsack for two time:First choose a subset to be sum/3,then memorize them,and use the other set to calculate if there is a subset that can be sum/3. It sounds nice,but this is a greedy algorithm.We can't ensure one subset won't destroy another subset's structure.

For instance, {1,1,2,3,4,4}.The result is{1,4}{1,4}{2,3}.But it may generate a subset {1,1,3},so it can't always give the right solution.

To solve it,we can find that the case: it may use many small number to occupy one large number. So we can always use as large as number to pack the subset to find the solution(if it do exist).But it is same to backtrack.

The right dynamic programming is:3-dimension.We deal with two subset at the same time.

We define dp[i][j][k]:For first k numbers,whether it can get one subset has sum i,another has subset j.

The result is dp[sum/3][sum/3][n].Each time we think 5 situations:

1. Case 1: Don't use x in first two subsets
   - Check dp[i][j][k-1]==1
2. Case 2: Use x in the first subset
   - Check dp[i-x][j][k-1]==1 (if i >= x)
3. Case 3: Use x in the second subset
   - Check dp[i][j-x][k-1]==1 (if j >= x)

The transition function is: dp[i][j][k]=(i>=x&&dp[i-x][j][k-1])||(j>=x&&dp[i][j-x][k-1])|| dp[i][j][k-1].

4.2. Key Pseudocode

```
FUNCTION backtrack(num, dp, target, n):
  part1, part2, part3 = [], [], []
  i, j, k = target, target, n
  WHILE k > 0:
    //choose the last number
    x = num[k-1]
    // subset1:dp[i-x][j][k-1]==1
    IF i >= x AND dp[i-x][j][k-1]:
      part1.append(x)
      i = i - x
      k = k - 1
    // subset2:dp[i][j-x][k-1]==1
    ELSE IF j >= x AND dp[i][j-x][k-1]:
      part2.append(x)
      j = j - x
      k = k - 1
    // subset3:dp[i][j][k-1]==1
    ELSE:
      part3.append(x)
      k = k - 1
  RETURN part1, part2, part3
```

4.3. Analysis of Algorithm Implementation

After judging,we should use **backtracking** to find the three subset.For each element considered in reverse order.

Termination Check: If both subset requirements are satisfied (both equal zero), all remaining elements are automatically assigned to the third subset.

Exact Match Evaluation: The algorithm first checks if the current element perfectly matches either subset's remaining requirement. This is the most straightforward assignment.

Exclusion Verification: If no exact match exists, the algorithm checks if the current element can be excluded from both primary subsets (assigned to the third subset) while still maintaining a valid path to the solution.

Partial Assignment: If exclusion isn't viable, the algorithm attempts to assign the element to either the first or second subset, provided the element's value doesn't exceed the subset's remaining requirement and such assignment leads to a valid previous state.

Default Assignment: If none of the above conditions are met, the element is assigned to the third subset by default.Every time we choose the last number x in the set,check dp[i-x][j][k-1],dp[i][j-x][k-1],dp[i][j][k-1]which is equal to 1.

4.4. Complexity Analysis

The time and space complexity is both $O(n \times target^2)$, where n is the number of elements and target is the sum each subset must achieve. For n numbers,every number need to match $target^2$ times. The time and space used to do backtrack is both only O(n). So the total time and space complexity is both $O(n \times target^2)$.

And if for k-partition,then the time complexity and space complexity is both $O(n \times target^{k-1})$

Chapter 5: Test

5.1. Random sample

At first, we tested our code on purely random samples. However, since the probability of a random set of numbers being perfectly 3-partitionable is extremely low, this was insufficient for testing the cases can be solved. To address this, we developed a random generator with four distinct modes to create comprehensive test scenarios:

1. Mode 0: Pure Random
   - Generates N random integers uniformly distributed in $[1, \text{max\_val}]$. These cases almost never have a valid solution.
2. Mode 1: Guaranteed "Yes" Case
   - Constructs a valid solution by generating K buckets individually. For each bucket, it generates random numbers for the first few slots and calculates the final number to ensure the bucket sums exactly to `target`. Finally, all numbers are shuffled to hide the structure.
3. Mode 2: Sum-Divisible Random
   - Mode 0 may be pruned quickly. So we tried to generates random numbers and adjusts one number to ensure the total sum is divisible by K.
4. Mode 3: Near-Miss Case
   - Starts with a valid "Yes" case and applies a small perturbation: select two numbers, increment one and decrement the other. This preserves the total sum (the divisibility by K) but breaks the specific solution structure, tried to create difficult "No" instances.

5.2. Test cases

From the analysis below, we can found that the time complexity is depended on N, the number of numbers first. Then Three-dimensional dp algorithm basen on the target sum. In the bonus part, we should also test if k affects the time complexity of bitmask dp and backtrack algorithm.

5.2.1. N value

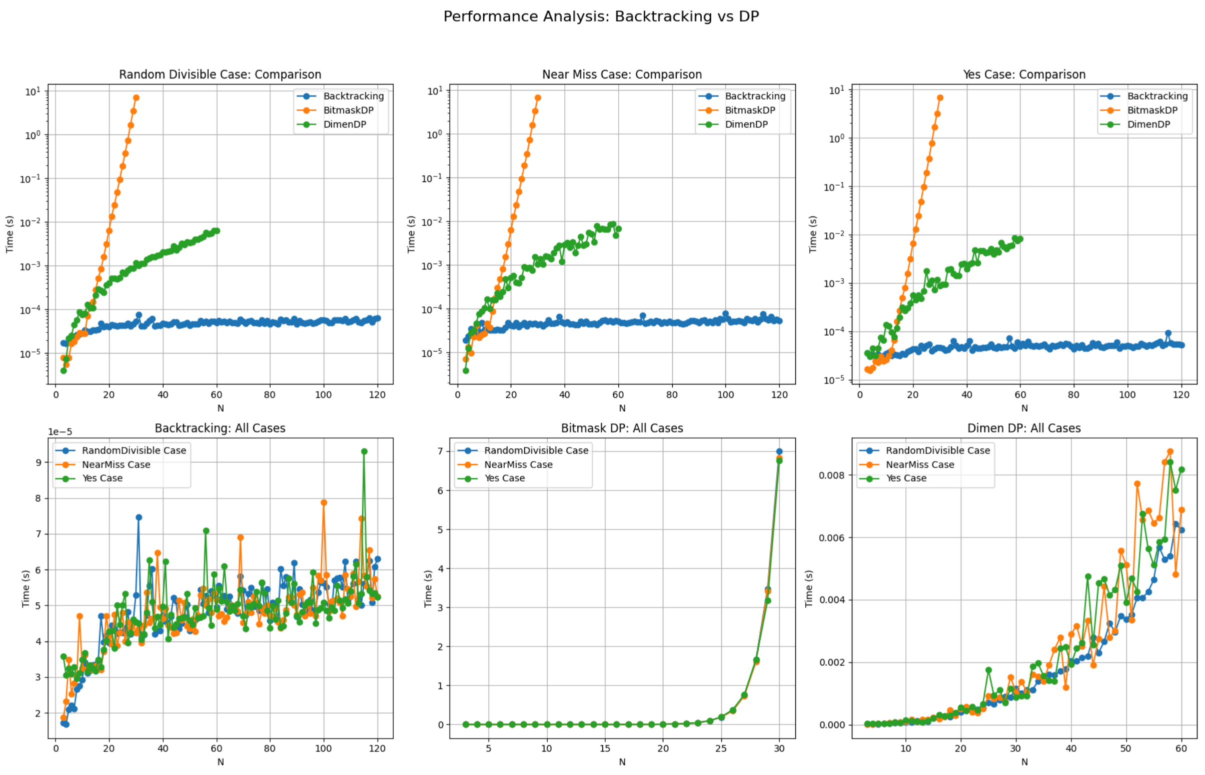

Figure 1: Time vs N

We keep k=3 and max_val=20,then change N from 3 to 120 with step 1, and test 100 times for every N.

The experimental results strongly align with our theoretical analysis in previous chapters, though with some interesting nuances regarding the Backtracking algorithm.

1. Bitmask DP: The results are perfectly consistent with the theoretical complexity of $O(N \cdot 2^N)$ derived in Chapter 3. As seen in the graph, the runtime explodes exponentially once N exceeds 20. This confirms that the state space size (2^N) is the dominant factor, making this approach strictly limited to small N, regardless of the values of the numbers.

2. 3D DP: The results are consistent with the analysis $O(N \cdot \text{target}^2)$ in Chapter 4. In this test, we fixed `max_val=100`, but as N increases, the total sum (and thus the `target`) grows linearly with N $(\text{target} \approx N \cdot \frac{\text{avg}}{3})$. Substituting this into the complexity formula yields $O(N \cdot N^2) = O(N^3)$. The graph reflects this polynomial growth, showing that 3D DP scales much better than Bitmask DP for this specific constraint (small numbers, large N).

3. Backtracking: The results might seem inconsistent with the theoretical **worst-case** complexity of $O(K^N)$ (exponential), as the curve remains flat and extremely low. However, this validates our “Actual Performance” analysis in Chapter 2. These pruning strategies and ordered filling drastically reduce the effective search space in

practice. For N up to 120, the runtime of the program demonstrates that real-world performance can be orders of magnitude better than worst-case theoretical bounds due to problem structure and optimizations.

And the Mode-3 “Near-Miss” seems fail to trigger worst-case behavior for Backtracking in this test. This is likely because, with larger N, the number of combinations grows so large that even “Near-Miss” cases still have many valid partial solutions that allow pruning to remain effective.

5.2.2. Target sum

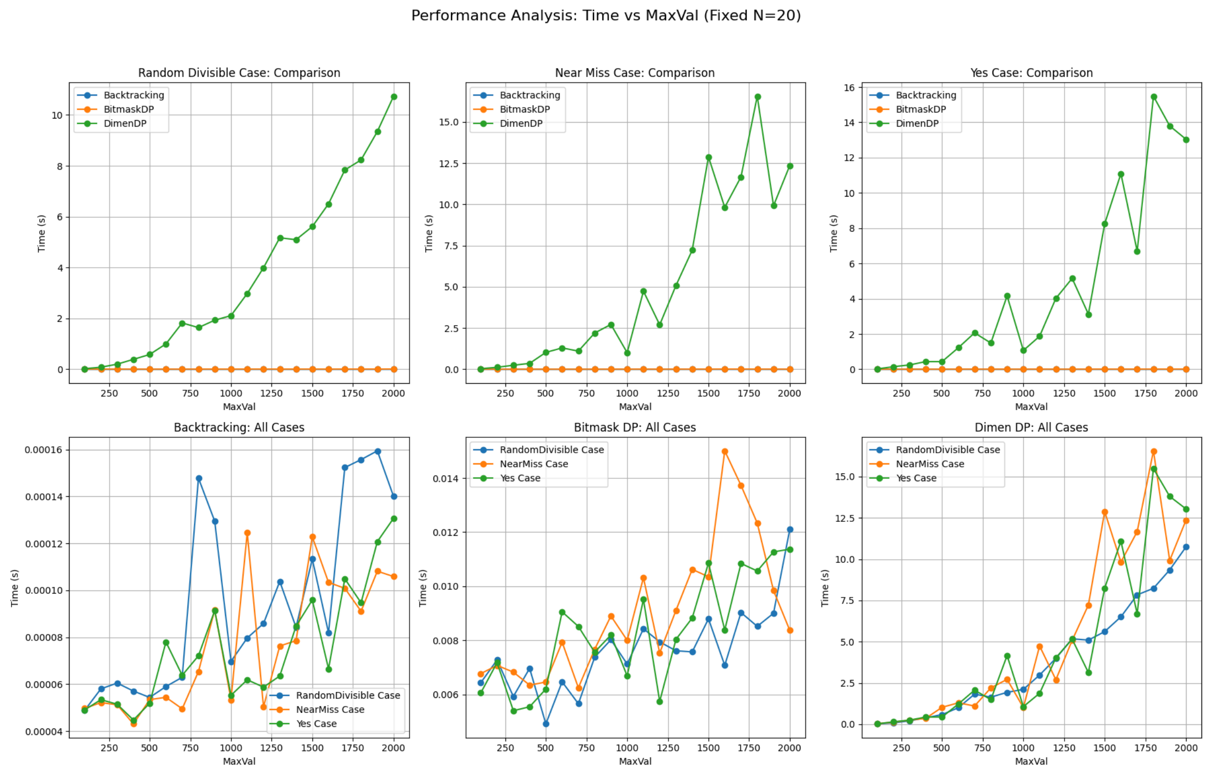

Figure 2: Time vs Target Sum

We tried to increase max_val to increase the total sum, simulating the effect of larger target sums. We keep k=3 and N=20,then change max_val from 100 to 2000 with step 100, and test 10 times for every max_val.

1. 3D DP (Quadratic Growth): The runtime grows quadratically with `max_val`. This confirms the $O(N \cdot \text{target}^2)$ complexity. Since `target` is proportional to `max_val`, the time complexity is effectively $O(\text{max\_val}^2)$. This highlights the major weakness of pseudo-polynomial algorithms: they perform poorly when the input numbers are large, even if N is small.
2. Bitmask DP (Constant): The runtime is almost constant. This validates that Bitmask DP’s complexity $O(N \cdot 2^N)$ depends **only** on N, not on the magnitude of the numbers.

3. Backtracking: Similar to Bitmask DP, Backtracking is relatively insensitive to the magnitude of numbers. It remains efficient because the search space structure (determined by N) hasn't changed significantly, and pruning remains effective.

5.2.3. K value

This test is much more difficult to analyze. First we haven't implemented the K-dimensional DP due to its impracticality for $K > 3$.

Then for other two algorithms, because of the limit of the long runtime of Bitmask DP, we only test N=15 here. But it turn to be a limit for the K. If the K is too large relative to N, obviously there is fewer numbers in each partition, making it is easier to prove that no valid partition exists, as the graph shows below.

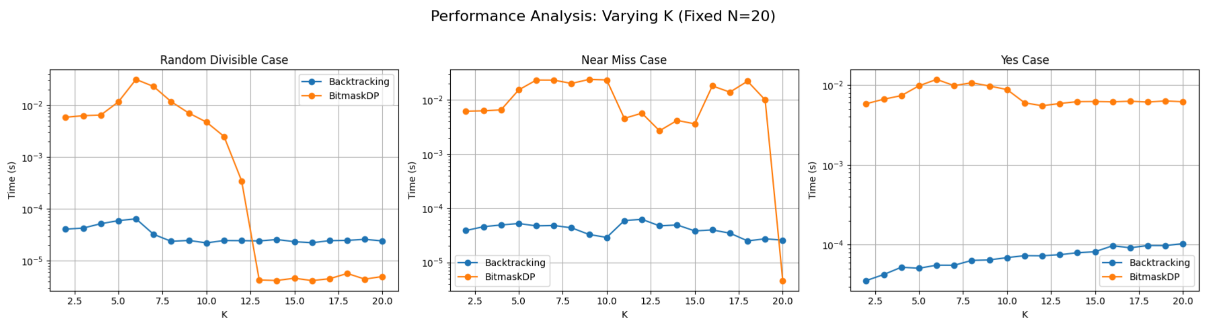

Figure 3: Time vs K, N=15

For Bitmask DP show in this figure, the complexity $O(N \cdot 2^N)$ is theoretically independent of K. The graph confirms this: the runtime remains stable as K varies. This makes Bitmask DP a versatile choice when N is small, regardless of how many partitions are required.

And then we removed test for Bitmask and then keep N=120 and max_val=20, then change K from 2 to 31 with step 1, and test 10 times for every K.

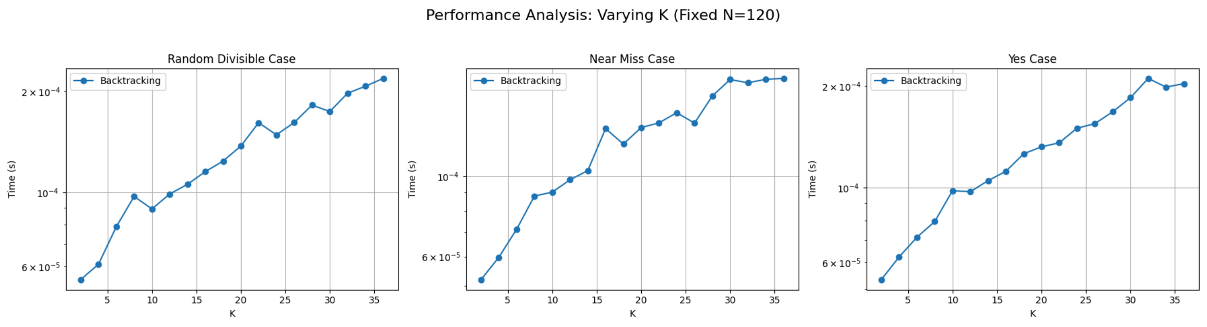

Figure 4: Time vs K, N=120

For Backtracking show in this figure, the complexity is theoretically $O(K^N)$ (assigning each of N items to one of K buckets), which means for k, this is an algorithm of polynomial time. However, as K increases, the runtime does not explode as strictly as K^N would suggest. This aligns with our analysis in Chapter 2.