

Projects 5: Three Partition Problem

Advanced Data Structure and Algorithm

2025.12.01

林润铎，窦宇浩，张钊

Contents



1	Problem Definition	4
1.a	The 3-Partition Problem	5
1.b	NP-Completeness Proof	6
2	3-Dimension Dynamic Programming	7
2.a	Core Idea	8
2.b	Key Implementation	9
2.c	Performance	11
3	Bucket-Centric Backtracking	12
3.a	Core Idea: Recursive Subset Construction	13
3.b	Key Implementation (Recursive)	14
3.c	Analysis and Generalization	16
4	Bitmask DP	17
4.a	Core Idea: State Compression	18
4.b	Key Optimizations: Implicit State	19
4.c	Complexity Analysis: Bitmask DP	20
5	Test	21

Contents (ii)



5.a Random sample	22
5.b N value	23
5.c Target sum	24
5.d K value	25
6 Thank You!	27

1 Problem Definition

The 3-Partition Problem



Input: A multiset of N positive integers $S = \{a_1, a_2, \dots, a_n\}$.

Output: Decide whether S can be partitioned into three disjoint subsets S_1, S_2, S_3 such that:

$$\sum_{x \in S_1} x = \sum_{x \in S_2} x = \sum_{x \in S_3} x = \frac{\sum_{x \in S} x}{3}$$

Example: Given $S = \{1, 2, 3, 4, 5, 6, 9\}$, Sum = 30, Target = 10. Valid Partition: $\{1, 9\}, \{4, 6\}, \{2, 3, 5\}$.

NP-Completeness Proof



Theorem: The 3-Partition Problem is NP-Complete.

Proof Logic (Reduction): To prove it is NP-Hard, we show that if we can solve 3-Partition, we can solve the known NP-Complete **2-Partition Problem**(can be reduced to **subset problem**).

1. Reduction Strategy:

- **Given:** An instance of 2-Partition (Set A , total sum $2M$).
- **Goal:** Determine if A can be split into two subsets of sum M .
- **Construction:** Create a new set $A' = A \cup \{M\}$.
 - Total sum of A' is $3M$. Target for 3-Partition is M .

2. Equivalence:

- (\rightarrow) If A has a 2-partition (A_1, A_2) , then $\{A_1, A_2, \{M\}\}$ is a valid 3-partition of A' .
- (\leftarrow) If A' has a 3-partition, one subset **must** be $\{M\}$ (since it contains element M and target is M , no other positive integers can be added). The remaining two subsets form a 2-partition of A .

Conclusion: 3-Partition is at least as hard as 2-Partition.

2 3-Dimension Dynamic Programming

Core Idea



Judge First, then Find Solution: Use 3D DP to determine if solution exists, then reconstruct actual partition.

DP:

State Definition: `bool dp[i][j][k]` represents first `k` numbers, which first subset with `sum=i`, and another subset with `sum=j`.

```
dp[i][j][k] =  
    dp[i][j][k-1] OR                // Put in third subset  
    (i >= x && dp[i-x][j][k-1]) OR    // Put in first subset  
    (j >= x && dp[i][j-x][k-1])        // Put in second subset
```

Reconstruct: Choose the last number, check which subset keeps the state is true

Key Implementation



```
Function DP(dp[][][], num[],n,target):
    dp[0][0][0]=1;
    for(k=1;k<=n;k++) {
        dp[0][0][k]=1,x=num[k-1];
        for(i=0;i<=target;i++) {
            for(j=0;j<=target;j++) {
                if(dp[i][j][k-1]) dp[i][j][k]=1; //subset3
                else if(i>=x && dp[i-x][j][k-1]) dp[i][j][k]=1;//subset1
                else if(j>=x && dp[i][j-x][k-1]) dp[i][j][k]=1;//subset2
            }
        }
    }
    if(dp[target][target][n]) return true;
```

Key Implementation (ii)



```
while(n>0){  
  //subset3  
  if(dp[i][j][n]) part3.push(num[n-1]);  
  //subset1  
  else if(i>=x && dp[i-x][j][n]) part1.push(num[n-1]);  
  //subset2  
  else if(j>=x && dp[i][j-x][n]) part2.push(num[n-1]);  
  n--;  
}  
return part1,part2,part3
```

Performance



Time Complexity: $O(n \times target^2)$

- We iterate N times, each time we fill $dp[i][j]$ from $(0,0)$ to $(target,target)$.
- it is pseudo-polynomial time complexity.

Space Complexity: $O(n \times target^2)$

- The space of dp Array.

3 Bucket-Centric Backtracking

Core Idea: Recursive Subset Construction



Three Key Optimization Strategies:

- 1. Implicit Final Bucket:** If $K - 1$ buckets are successfully filled with the target sum, the remaining unpicked numbers must inevitably sum to the target. We stop recursion at $k = 1$, reducing search depth significantly.
- 2. Heuristic Sort (Descending):** Sorting the input array in descending order allows the algorithm to attempt filling buckets with larger numbers first. This reduces the branching factor early in the recursion tree.
- 3. State-Based Pruning (Fail-Fast):** We utilize a `used` array to track element availability. Crucially, if the algorithm fails to fill a new (empty) bucket with the largest available number, it immediately returns `false` (pruning the entire branch), as that number cannot be placed elsewhere.

Key Implementation (Recursive)



The logic focuses on filling one bucket at a time. Once a bucket is full (`current_sum == target`), it recursively attempts to fill the next bucket.

```
Function Backtrack(k, current_sum, start_index):  
    // Optimization 1: Last bucket is automatic  
    IF k == 1: Mark remaining as Bucket 1; RETURN TRUE  
  
    // Bucket filled, reset to fill next bucket  
    IF current_sum == target:  
        RETURN Backtrack(k - 1, 0, 0)  
  
    FOR i FROM start_index TO N:  
        IF used[i]: CONTINUE  
        IF current_sum + numbers[i] > target: CONTINUE
```

Key Implementation (Recursive) (ii)



```
// Pruning: Skip duplicates to avoid symmetry
IF i > start AND numbers[i] == numbers[i-1] AND !used[i-1]: CONTINUE

used[i] = TRUE
IF Backtrack(k, current_sum + numbers[i], i + 1): RETURN TRUE
used[i] = FALSE // Backtrack step

// Pruning: Largest available item failed in empty bucket
IF current_sum == 0: RETURN FALSE

RETURN FALSE
```

Analysis and Generalization



Complexity Analysis

- **Time Complexity:** Roughly $O(K_c \cdot 2^{\{N\}})$. By isolating one bucket at a time, the problem size (N) effectively decreases for subsequent buckets, though worst-case remains exponential.
- **Space Complexity:** $O(N)$. Requires linear space for the recursion stack and the auxiliary `used` and `bucket_id` arrays.

Generalization to K-Partition

- **Algorithm Validity:** The bucket-centric logic is the standard solution for the “Partition to K Equal Sum Subsets” problem.
- **Implementation Changes:**
 - Initial call changes from `backtrack(3, ...)` to `backtrack(K, ...)`.
 - Total sum validation becomes `total_sum % K == 0`.

4 Bitmask DP

Core Idea: State Compression



Problem Modeling: We use Top-Down Dynamic Programming with Memoization. Since the order within a bucket doesn't matter, the state is uniquely defined by the subset of used numbers.

Bitmask Representation: A `mask` (integer) represents the set of used numbers:

- If the i -th bit is `1`, the number is used.
- If the i -th bit is `0`, the number is available.

State Transition: We define `DP(mask)`: Is it possible to partition the **remaining** numbers into valid buckets? We cache results in a `memo` table to avoid re-calculating the same subproblems.

Key Optimizations: Implicit State



We reduced the state space from standard $Nc \cdot 2^N$ to just 2^N using mathematical properties:

- 1. State Reduction:** We do **not** store `current_sum` in the DP state.
 - **Reasoning:** For any given `mask`, the total sum of used numbers is fixed. The current bucket's fill level is deterministically calculated.
- 2. Implicit Bucket Switching:** We use modulo arithmetic to handle bucket transitions automatically:

$$\text{next_sum} = (\text{current_sum} + \text{value}) \bmod \text{target}$$

- If the sum reaches `target`, the modulo operation resets it to `0`, signaling the start of a new bucket.

Complexity Analysis: Bitmask DP



Time Complexity

- **Analysis:** $O(Nc \cdot 2^N)$.
- **Independence from K:** Unlike backtracking, the complexity depends purely on N . Increasing the number of partitions (K) does **not** increase the search depth significantly.

Space Complexity (The Bottleneck)

- **Memory Usage:** $O(2^N)$.
- **Constraint:**
 - The `memo` array grows exponentially.
 - $N = 20$: 1MB (Fast).
 - $N = 30$: Requires Gigabytes (MLE).
- **Conclusion:** DP is superior for small N with complex constraints, but fails for large N due to memory limits.

5 Test

Random sample



Almost Failed Cases

1. Pure Random

- Generates N random integers uniformly distributed in $[1, \text{max_val}]$. These cases almost **never** have a valid solution.

2. Sum-Divisible Random

- We tried to generate random numbers and adjust one number to ensure the total sum is divisible by K to avoid initial pruning.

Nearly Succeeded Cases

3. Guaranteed “Yes” Case

- Constructs a valid solution by generating K buckets individually.
- Choose a target sum and then randomly fill each bucket with numbers summing to that target.
- Shuffled to hide the structure.

4. Mode 3: Near-Miss Case

- Starts with the case below
- Increase one number and decrease the other.
- Tried to cause a worse case

N value



1. Bitmask DP:
 - $O(N \cdot 2^N)$ derived in Chapter 3.
2. Three-dimensional DP:
 - $O(N \cdot \text{target}^2) = O(N^3)$ in Chapter 4.
 - fixed `max_val=100`, but as N increases, the `target` grows linearly with N .
3. Backtracking:
 - **worst-case** $O(K^N)$ (exponential)
 - Pruning strategies and ordered filling drastically reduce the effective search space in practice.

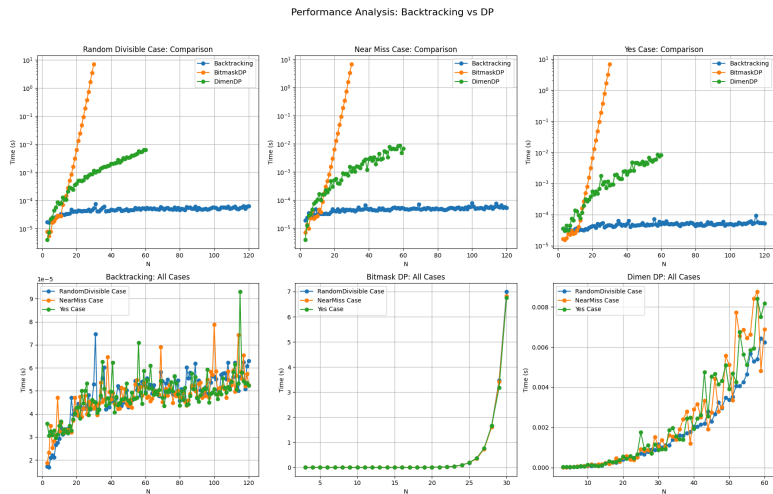


Figure 1: Time vs N , $k = 3$, `max_val` = 20

Target sum



1. Three-dimensional DP:

- $O(N \cdot \text{target}^2)$ complexity. Since `target` is proportional to `max_val`, the time complexity is effectively $O(\text{max_val}^2)$.

2. Bitmask DP and Backtracking:

Theoretically, they have nothing to do with the size of the target, but as the number grows, so does the calculation time

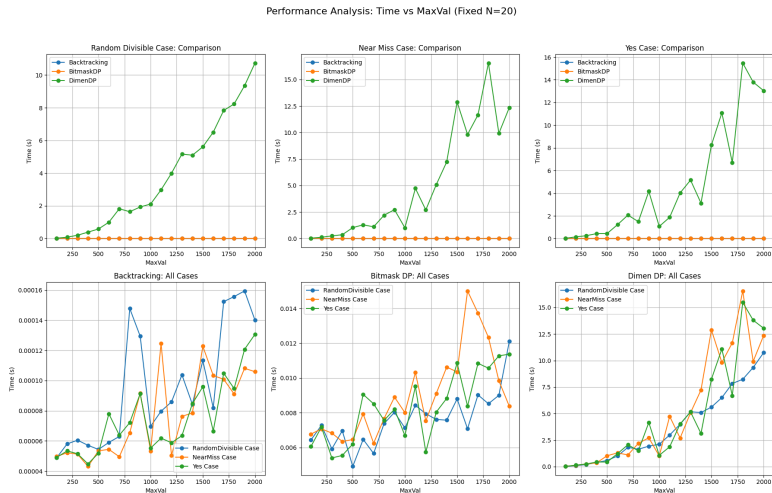


Figure 2: Time vs Target Sum, $k = 3$, $N = 20$

K value



Performance Analysis: Varying K (Fixed N=20)

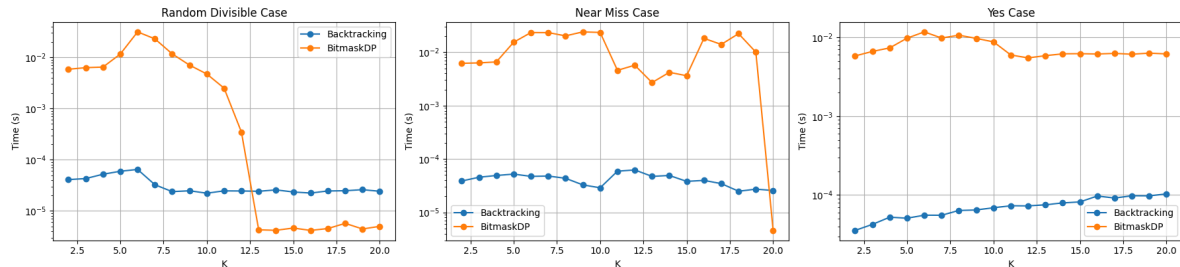


Figure 3: Time vs K, N=15, max_val=100

We only test Bitmask DP and Backtracking, but because of the limit of the long runtime of Bitmask DP, N must be small. It turn to be a limit for the K.

For Bitmask DP, the time complexity is $O(N \cdot 2^N)$, which is independent of K. But due to the limit of N, 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 above.

K value (ii)



Performance Analysis: Varying K (Fixed N=120)

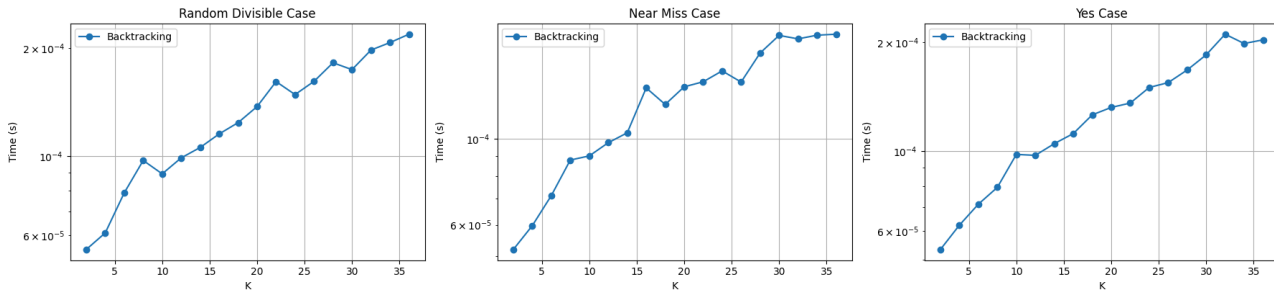


Figure 4: Time vs K, N = 120, max_val = 100

For Backtracking, the time complexity is theoretically $O(K^N)$, 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.

6 Thank You!