Projects 3: Recover the Design

Advanced Data Structure and Algorithm

2025.11.03

Contents

1 Chapter 1: Algorithm Design - Backtracking

Core Idea: Sequential Bucket Filling

Naive Approach vs. Our Strategy: Instead of trying to decide which bucket an item belongs to (which leads to K^N states), we invert the decision process.

Sequential Filling: We attempt to completely fill one bucket to the `target_sum` before moving on to the next.

> The algorithm focuses on filling the k-th bucket using available numbers. Only when `current_bucket_sum == target_sum` do we proceed to bucket $k - 1$.

Heuristic Optimization:

- The input array is sorted in descending order.
- **Rationale**: We prioritize placing larger numbers. They are harder to fit, allowing the algorithm to "fail fast" and prune invalid branches early in the recursion tree.

Pruning Strategies (Constraint & Symmetry)

To transform the exponential search into an efficient solver, we apply three strict pruning rules:

1. **Capacity Constraint**: If `current_sum + numbers[i] > target_sum`, skip the number immediately.
2. **Local Symmetry (Duplicate Pruning)**: If `numbers[i] == numbers[i-1]` and the previous number was skipped (`!used[i-1]`), using the current one would produce an identical state. Skip to avoid redundancy.
3. **Strong Pruning (Empty Bucket Failure)**: If we fail to fill a **newly opened empty bucket** starting with the largest available number, no solution exists.
   - **Logic**: The largest remaining number **must** go somewhere. If it cannot start the current bucket, it cannot start any subsequent equivalent bucket.

Complexity Analysis: Backtracking

Time Complexity

- **Worst-case**: $O(Kc \cdot 2^N)$.
- **Practical Performance**: Due to "Sequential Filling" and aggressive pruning (especially the **Empty Bucket** check), the effective branching factor is drastically reduced.
- **Scalability**: Runs instantaneously for $Nle30$.

Space Complexity

- **Auxiliary Space**: $O(N)$.
- **Components**:
  - `used` array and `bucket_id` array take linear space.
  - Recursive stack depth is bounded by N.
- **Advantage**: Very memory efficient compared to DP.

2 Chapter 2: Algorithm Design - 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.

3 Chapter 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 generates random numbers and adjusts one number to ensure the total sum is divisible by K to avoid initial pruning.

Nearly Successed Cases

3. Guaranteed "Yes" Case
   - Constructs a valid solution by generating K buckets individually.
   - Choose a target sum and then randomly fills 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 (target $\approx N \cdot \frac{\text{avg}}{3}$).
3. Backtracking:
   - **worst-case** $O(K^N)$ (exponential)
   - Pruning strategies and ordered filling drastically reduce the effective search space in practice.

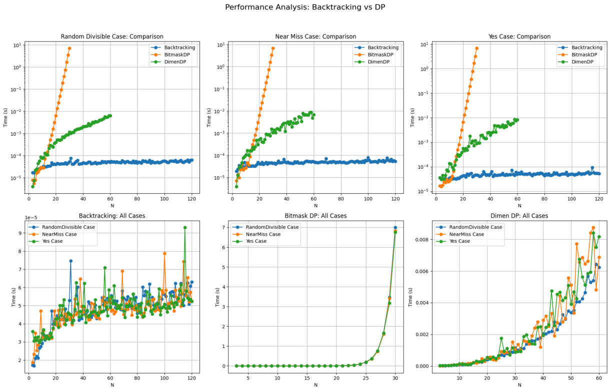

Figure 1: Time vs N

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.

Target sum

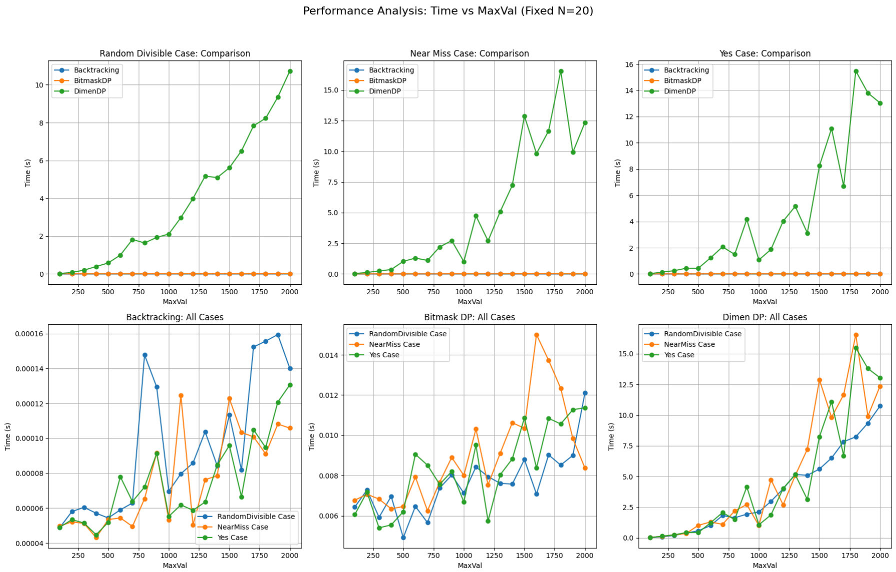

Figure 2: Time vs Target Sum

Target sum (ii)

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.

Target sum (iii)

3.c.a 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.

Target sum (iv)

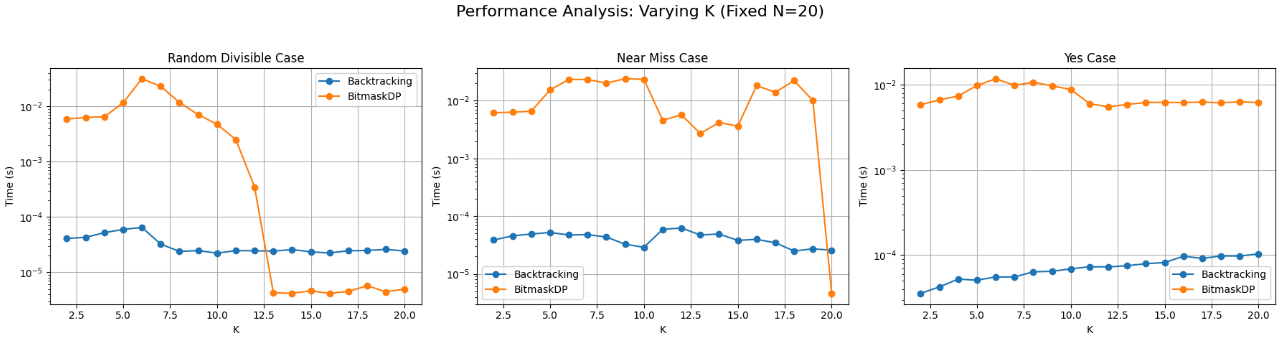

Figure 3: Time vs K, N=15

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.

Target sum (v)

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.

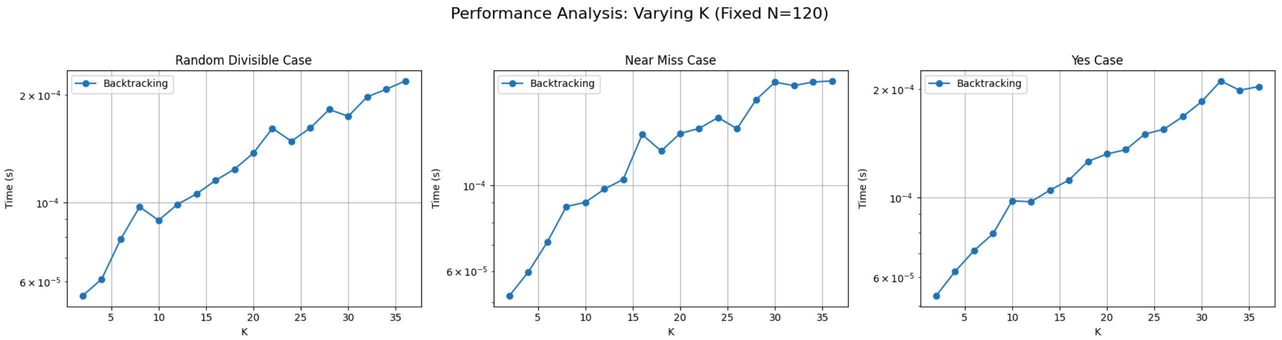

Figure 4: Time vs K, N=120

For Backtracking show in this figure, the complexity is theoretically $O(K \times 2^N)$ (assigning each of N items to one of K buckets), which means for k, this is an algorithm of polynomial time. However, as K

Target sum (vi)

increases, the runtime does not explode as strictly as K^N would suggest. This aligns with our analysis in Chapter 2.

4 Thank You!