Dice Rolls With Target Sum — LeetCode 1155

Goal: Given `n` dice, each with `k` faces (1..k), count the number of ways to get sum = `target`. Return result **mod 1e9+7**.

Problem statement (short)

- Input: integers `n`, `k`, `target`.
- Output: number of possible ways to roll `n` dice so their face-sum equals `target`, **modulo** **`1e9+7`**.
- Constraints: `1 <= n, k <= 30`, `1 <= target <= 1000`.

Important note: The exact number of ways can be astronomical (exponential). Problem asks for **answer % (1e9+7)**, so we compute and store remainders during DP to avoid overflow.

Intuition (simple)

- Think recursively: to get sum `t` with `n` dice, pick the first die face `i` (1..k), then count ways to get sum `t-i` with `n-1` dice.
- Recurrence:

```
ways(n, t) = sum_{i=1..k} ways(n-1, t-i)
```

- Base cases:
  - `ways(0, 0) = 1` (0 dice, sum 0 ⇒ one valid empty outcome)
  - `ways(0, t) = 0` for `t != 0`
  - `ways(n, t) = 0` for `t < 0`
- Because of overlapping subproblems we memoize or use bottom-up DP.

Why `% 1e9+7` and why we store remainders

1. Counts grow exponentially → cross `long long` limits. Storing full exact number is infeasible.
2. Modular arithmetic property makes it safe to store remainders at every step:

```
(a + b) % M = ((a % M) + (b % M)) % M
```

So storing `ways % M` at each DP cell yields a correct final `(exact ways) % M`.

Tradeoff: you cannot recover the exact count from the remainder — but problem explicitly asks for remainder, so that's fine.

Top-down (recursion + memo) — explained

- Use `dp[n+1][target+1]` initialized to `1`.
- `f(n, k, t)` returns number of ways to make `t` with `n` dice.
- Memoize `dp[n][t]` after computing.
- Apply `% MOD` after each addition to avoid temporary overflow.

Code (top-down)

```
class Solution {
public:
    const long long MOD = 1000000007LL;

    long long f(int n, int k, int t, vector<vector<long long>> &dp) {
        if (n == 0) return (t == 0) ? 1 : 0;   // base-case
        if (t < 0) return 0;                   // safe-guard
        if (dp[n][t] != -1) return dp[n][t];

        long long sum = 0;
        for (int i = 1; i <= k; ++i) {
            if (t - i >= 0) {
                sum = (sum + f(n - 1, k, t - i, dp)) % MOD;  // important: mod here
```

```
            }
        }
        dp[n][t] = sum;
        return dp[n][t];
    }

    int numRollsToTarget(int n, int k, int target) {
        vector<vector<long long>> dp(n + 1, vector<long long>(target + 1, -1));
        return (int)f(n, k, target, dp);
    }
};
```

Complexity (top-down):

- States: `(n+1) * (target+1)` → `O(n * target)` distinct states.
- For each state we loop up to `k` faces → time `O(n * target * k)`.
- Space: `O(n * target)` for memo + recursion stack `O(n)`.

Bottom-up (tabulation) — iterative (recommended)

- `dp[d][t]` = number of ways to get sum `t` using exactly `d` dice.
- Base: `dp[0][0] = 1`.
- Transition:

```
for d in 1..n:
  for t in 0..target:
    dp[d][t] = sum_{i=1..k, t-i >= 0} dp[d-1][t-i]  (apply % MOD)
```

Code (bottom-up)

```
class Solution {
public:
    int numRollsToTarget(int n, int k, int target) {
        const int MOD = 1e9 + 7;
```

```
        vector<vector<int>> dp(n + 1, vector<int>(target + 1, 0));
        dp[0][0] = 1;

        for (int dice = 1; dice <= n; ++dice) {
            for (int t = 0; t <= target; ++t) {
                long long ways = 0;
                for (int face = 1; face <= k; ++face) {
                    if (t - face >= 0) {
                        ways += dp[dice-1][t-face];
                    }
                    if (ways >= MOD) ways -= MOD; // optional micro-optimization
                }
                dp[dice][t] = ways % MOD;
            }
        }
        return dp[n][target];
    }
};
```

Complexity (bottom-up):

- Time: `O(n * target * k)`
- Space: `O(n * target)` — can be optimized to `O(target)` using rolling array because `dp[d]` only depends on `dp[d-1]`.

Space-optimized version (1D rolling array)

- Keep `prev[t]` for `d-1` and `cur[t]` for `d`.
- After finishing a dice-layer, swap `prev` and `cur`.

```
class Solution {
public:
    int numRollsToTarget(int n, int k, int target) {
        const int MOD = 1e9 + 7;
        vector<int> prev(target + 1, 0), cur(target + 1, 0);
```

```
        prev[0] = 1;

        for (int dice = 1; dice <= n; ++dice) {
            fill(cur.begin(), cur.end(), 0);
            for (int t = 0; t <= target; ++t) {
                long long ways = 0;
                for (int face = 1; face <= k; ++face) {
                    if (t - face >= 0) ways += prev[t - face];
                    if (ways >= MOD) ways -= MOD;
                }
                cur[t] = ways % MOD;
            }
            prev.swap(cur);
        }
        return prev[target];
    }
};
```

Complexity after optimization:

- Time: `O(n * target * k)`
- Space: `O(target)`

Worked example (small)

- `n = 2`, `k = 6`, `target = 7`:
  - Ways: (1,6),(2,5),(3,4),(4,3),(5,2),(6,1) = 6
  - `dp[1][1..6] = 1 each` ; `dp[2][7] = dp[1][6] + dp[1][5] + ... + dp[1][1] = 6`

Edge cases / gotchas (common mistakes)

1. **Missing base-case**: If you only check `if (n==0 && t==0) return 1;` and forget `if(n==0) return 0;` for `t>0`, recursion can go to negative `n` and crash.
2. **Forgot modulo**: Leads to `long long` overflow and UB (sanitizer complaint). Apply `% MOD` on every addition.

3. **Wrong dp indices**: Store result in `dp[n][t]`, not `dp[n][k]` or other typos.
4. **Type mismatch**: Keep DP array type consistent (`long long` or `int` with modulo assumptions). Usually using `int` with modulo is fine because stored values < MOD.

Final notes (tl;dr)

- Use bottom-up with rolling array for best space/time tradeoff in practice.
- Always apply `% MOD` during additions to avoid overflow and match problem requirement.
- You will only get `(exact_ways % MOD)`, not the exact astronomical count — that's intended.

Quick reference: final recommended code (space-optimized)

```
class Solution {
public:
    int numRollsToTarget(int n, int k, int target) {
        const int MOD = 1e9 + 7;
        vector<int> prev(target + 1, 0), cur(target + 1, 0);
        prev[0] = 1;

        for (int dice = 1; dice <= n; ++dice) {
            fill(cur.begin(), cur.end(), 0);
            for (int t = 0; t <= target; ++t) {
                long long ways = 0;
                for (int face = 1; face <= k; ++face) {
                    if (t - face >= 0) ways += prev[t - face];
                    if (ways >= MOD) ways -= MOD;
                }
                cur[t] = ways % MOD;
            }
```

```
            prev.swap(cur);
        }
        return prev[target];
    }
};
```