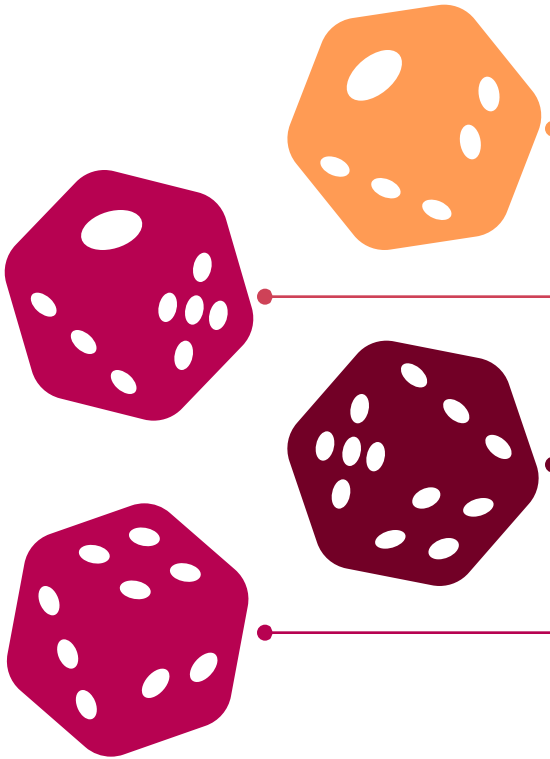
# Number of Dice Rolls With Target Sum

LeetCode 1155.

Thunchanok Iacharoen 6214599
Nahatai Sirisak 6214600

**Difficulty: Medium**

# Table of Contents



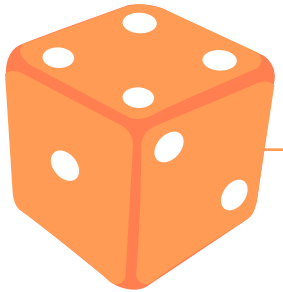
01

# Problem

# Question

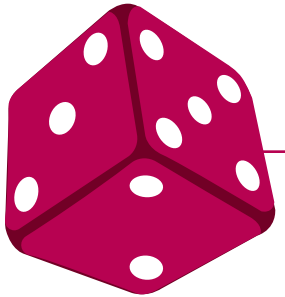
You have d dice and each die has f faces numbered 1, 2, ..., f. You are given three integers d, f, and target.

Return the number of possible ways (out of $f^d$ total ways) modulo $10^9 + 7$ to roll the dice so the sum of the face-up numbers equals target.



## Example 1

**Input:** d = 1, f = 6, target = 3
**Output:** 1
**Explanation:** You throw one die with 6 faces. There is only one way to get a sum of 3.



## Example 2

**Input:** d = 2, f = 6, target = 7
**Output:** 6
**Explanation:** You throw two dice, each with 6 faces. There are 6 ways to get a sum of 7: 1+6, 2+5, 3+4, 4+3, 5+2, 6+1.

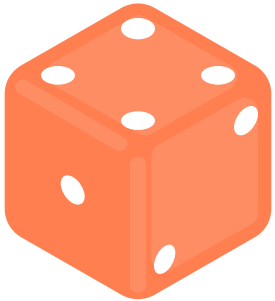
## Constraints

- 1 <= d, f <= 30
- 1 <= target <= 1000

02

# Analysis

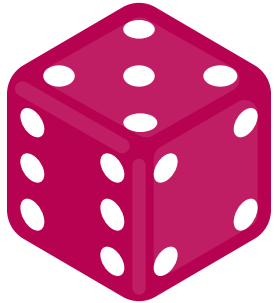
# Problem Analysis



A general algorithm for computing the number of ways to roll a particular total

Explore all combinations of all faces for all dices

This problem can be divided and solved in small sub problems



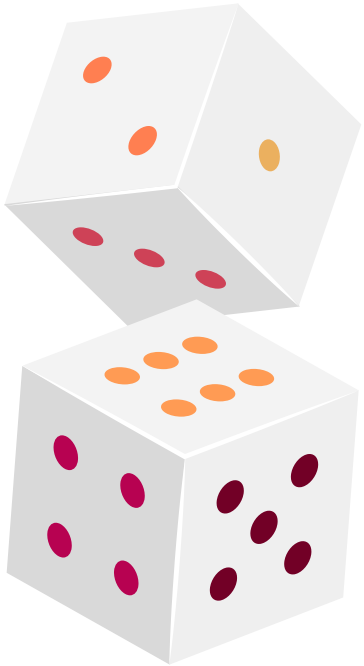
We’re looking for the most optimal solution

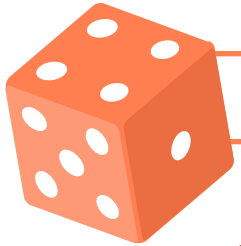
# 03

# Solutions

# Brute Force

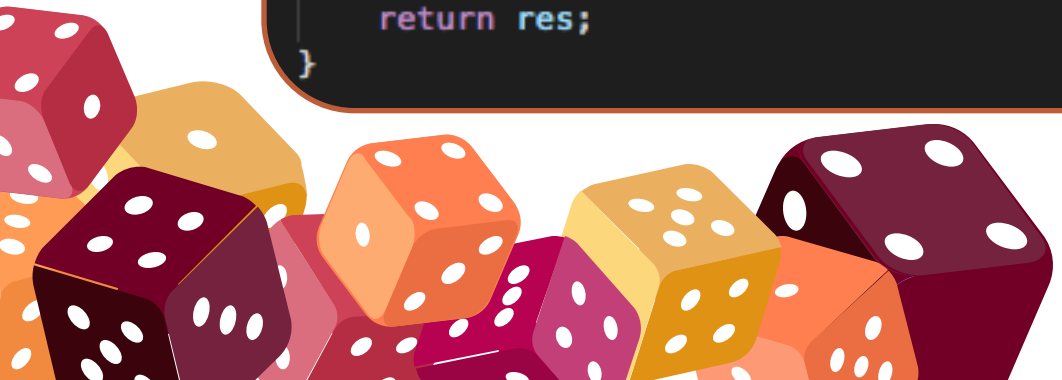
We started solving the problem using the most obvious solution. In this case, we try to explore all combinations of all faces for all dices, and count the ones that give a total sum of target.

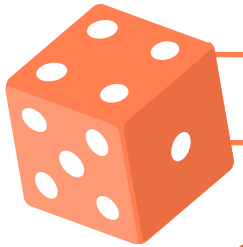




# The Code

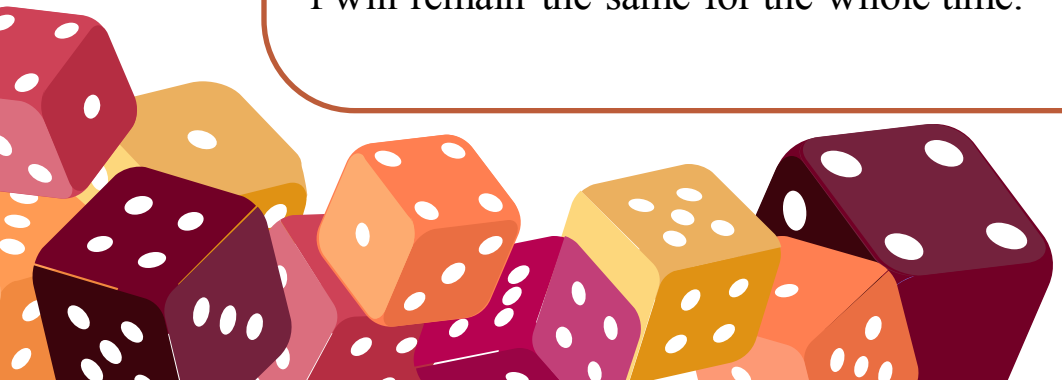
```
int numRollsToTarget(int d, int f, int target) {
    long long mod = 1e9 + 7;
    int res = 0;
    if (d == 0 || target <= 0) {
        return d == target;
    }
    for (int i = 1; i <= f; i++) {
        res = (res + numRollsToTarget(d - 1, f, target - i)) % mod;
    }
    return res;
}
```

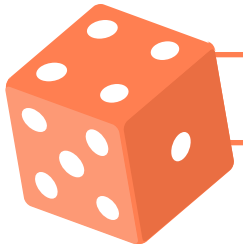




# The Code

We check if the number of dice is 0 or number of target left is 0, then return d == target which is the shorthand of number of dice and target is 0. Then, we target the value between 1 until f. We reduced the number of dice by 1 and our target by i because we already use one of our dice but on first part of the solving. So that's the number of way we can solve if our first dice is targeting certain value and the rest we do the process again. The number of f will remain the same for the whole time.





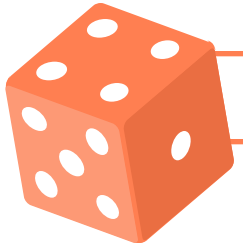
# The Result

After we conduct some test runs with sample data, we've come to the conclusion that even though the brute force technique worked quite well and can delivered us the correct answer, it's not a particularly efficient solution.

```
[Running] cd "/Users/pitpiboon/cplusplus/dic
d = 1, f = 6, and target = 3
1

[Done] exited with code=0 in 0.79 seconds

[Running] cd "/Users/pitpiboon/cplusplus/dic
d = 2, f = 6, and target = 7
6

[Done] exited with code=0 in 0.485 seconds
```

Unfortunately, it exceeded the time limit due to the amount of time we'll need to repeat the computating process.

| | | | | |
|---|---|---|---|---|
| 09/21/2021 17:55 | Time Limit Exceeded | N/A | N/A | cpp |

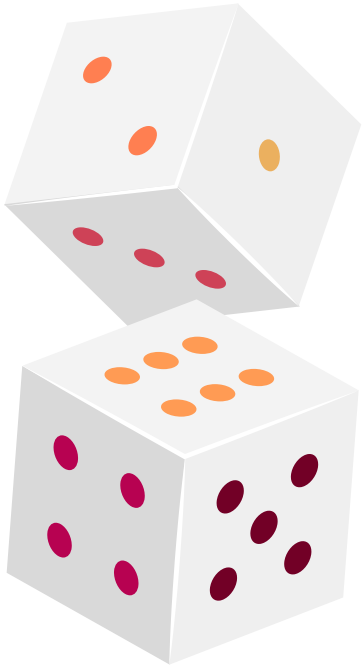


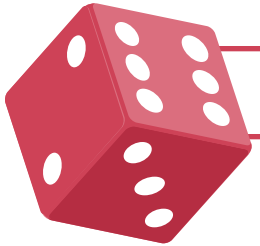
# Complexity Analysis

- The time complexity is $O(f^d)$ since brute force recursively call the function where each d calls every f.
- The space complexity is O(d) for stack.

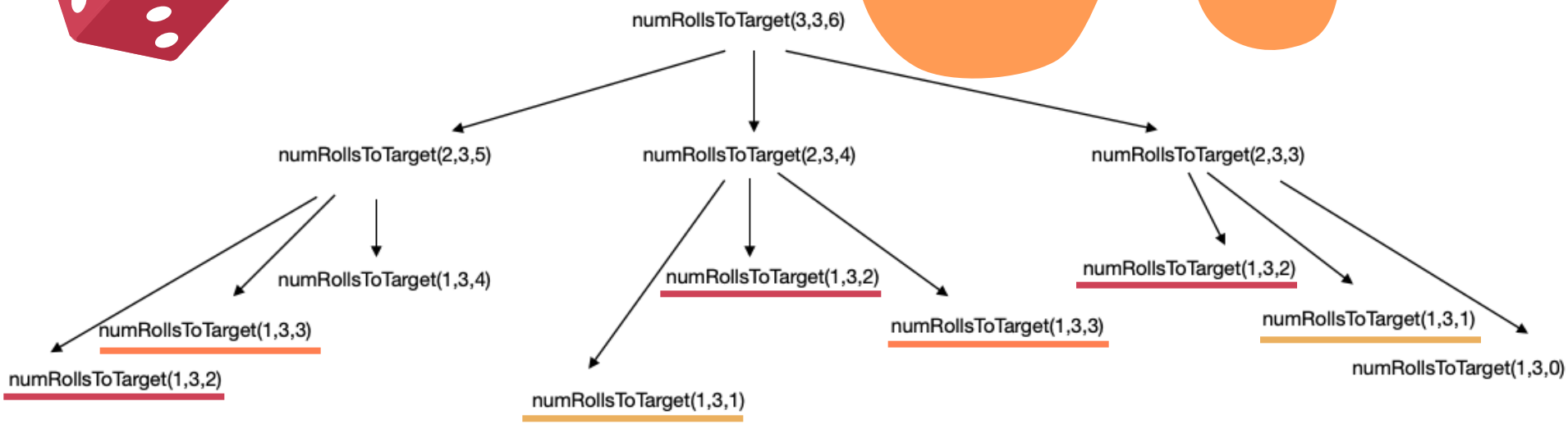
# Memoization (DP)

We attempt to optimize the runtime and memory by keeping the pre-computed results for dice i and target, then return it when the recursive function is called.





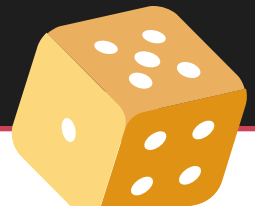
# Example




There's a lot of repeated work during the computation. That's why memoization is needed. As shown in the example, the number of ways of current die is decided by its previous die's ways to compose some targets. The repeated pairs are underlined. If we can catching the result for them, then we will be able to reduce the time for recalculation.

# The Code

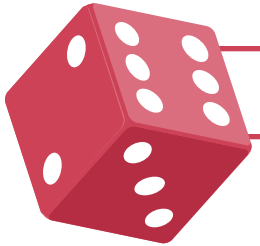
```
int dp[31][1001] = {};
int numRollsToTarget(int d, int f, int target, int res = 0) {
    int mod = 1e9 + 7;
    if (d == 0 || target <= 0) {
        return d == target;
    }
    if (dp[d][target]) {
        return dp[d][target] - 1;
    }
    for (int i = 1; i <= f; ++i) {
        res = (res + numRollsToTarget(d - 1, f, target - i)) % mod;
    }
    dp[d][target] = res + 1;
    return res;
}
```



# The Code

The code is similar to the brute force version but adding the array and checking if it exist.

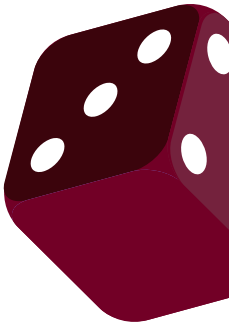
(1) Set the array as [31][1001] which is the maximum of the dice and the target.
(2) Set the base case similar to the brute force version and check if there exists the already computed value, if it exists, then it will return the value which is already computed instead of call the function again.
(3) Store the previous value in the array and use it when the call is repeated. We set the res + 1 to check whether this was pre-computed as all dp initialized to be 0. If pre-computed, return dp[d][target] - 1. When we get the value that already exist , then we subtract 1 as we add 1 to this state.



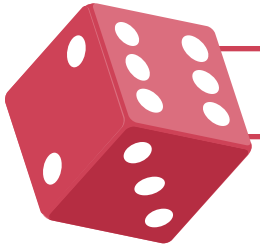
# The Result

With the additional of memoization technique to our code, we successfully improve the complexity.

It passed the time limit.



| | | | | |
|---|---|---|---|---|
| 09/22/2021 00:17 | Accepted | 8 ms | 5.9 MB | cpp |

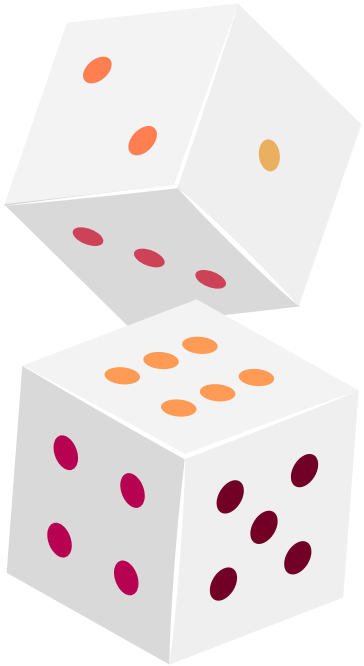


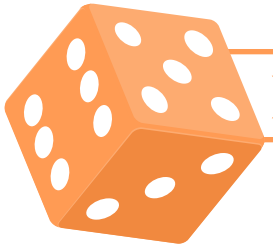
# Complexity Analysis

- The time complexity is O(d * f * target) since the function calls vary from 0 to d and 0 to target.
- The space complexity is O(d * target) for memoization.

# Tabulation (DP)

Another approach of dynamic programming that can also improve the complexity is tabulation. It used the bottom up technique which is different from the memoization that use the top down technique. With this technique there's no recursion but instead we starting at the base case and building up.

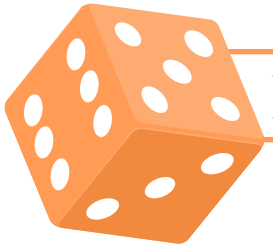




# Example

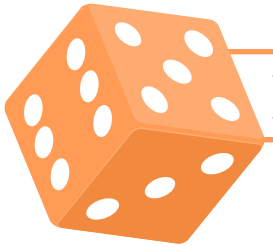
| D = 3 / F = 5 / T = 6 | TARGET | J | J | J | J | J | J | J |
|---|---|---|---|---|---|---|---|---|
| DICE | DICE/TARGET | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| I | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| I | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| I | 2 | 0 | 0 | 1 | 2 | 3 | 4 | 5 |
| I | 3 | 0 | 0 | 0 | 1 | 3 | 6 | 10 |

The table represents the array which will keep the result of the function which the value inside show the possibility of ways can dice i reached the target. The table use the range value of the dice and the target. The value that we getting are calculated from the previous dice sum up with face dp[I][j] = (dp[I][j] + dp[I-1][j-k]) % mod. With 3 dice 6 face and target 6 we can get 10 possible ways to reach the target.



# The Code

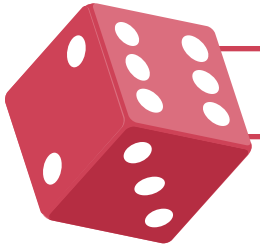
```
int numRollsToTarget(int d, int f, int target, int res = 0) {
    long long mod = 1e9 + 7;
    long long dp[d + 1][target + 1];
    for(int i = 0; i <= d; i++) {
        for(int j = 0; j <= target; j++) {
            if(i == 0 && j == 0) dp[i][j] = 1;
            else if(i == 0 || j == 0) dp[i][j] = 0;

            else {
                dp[i][j] = 0;
                for(int k = 1; k <= min(j, f); k++) {
                    dp[i][j] = (dp[i][j] + dp[i - 1][j - k]) % mod;
                }
            }
        }
    }
    return dp[d][target];
}
```



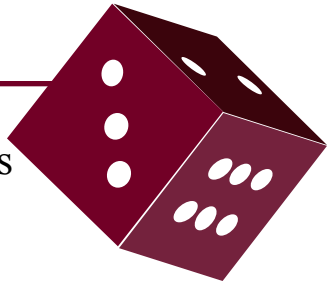
# The Code

We use nested loop in the function to loop through dice then target then loop through the range min(face, target) to sum up the possible way of the previous dice. after finish computation then store in the array to use it to execute the future value.

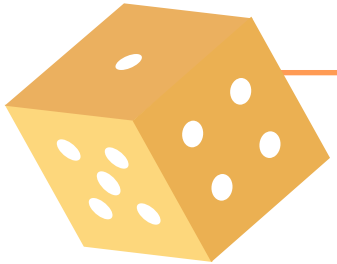
(1) Set the array to the size of the dice * target > [d+1][target+1]
(2) Set the base case: if the number of dice and target is 0 then return 1 because there is only 1 way to solve. However, if either dice or target is 0, then it's impossible to solve.
(3) For the Recurrence, with i dice, the possible ways to reach j target is the sum of the previous dice with each possible f face value of the current die. The compute value will be stored in the array. After compute all the possible ways return the final value.



# The Result

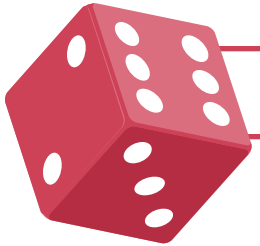


Tabulation technique helps us speed up the algorithm since there's no need to repeat the same process over and over again anymore.



It passed the time limit.

| | | | | |
|---|---|---|---|---|
| 09/20/2021 19:28 | Accepted | 37 ms | 6.2 MB | cpp |

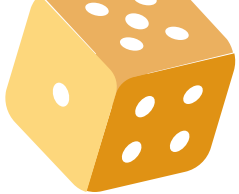


# Complexity Analysis

- The time complexity is O(d * f * target).
- The space complexity is O(d * target) > O(target) since we only need to store counts for the previous dice.

04

# Conclusion



# Final Comment

After we tested all three approaches, it can be clearly seen that DP (memoization and tabulation) is more efficient in term of runtime and memory usage.

During the test runs of each techniques, we are able to grasp the concept behind it quickly due to the similarity between them with a slight change.




# THANK YOU

**CREDITS:** The template was created by Slidesgo and Freepik.

**REFERENCES:**


- https://leetcode.com/problems/number-of-dice-rolls-with-target-sum/discuss/770166/evolve-from-brute-force-to-dp
- https://medium.com/tech-life-fun/leet-code-1155-number-of-dice-rolls-with-target-sum-graphical-explained-python3-solution-224f8c0af23
- https://leetcode.com/problems/number-of-dice-rolls-with-target-sum/discuss/473863/C%2B%2B-solution%3A-recursive-greater-memoization-greater-tabulation