Jumanji

The main non-standard insight of this problem is this:

Claim

We can separately compute the probability that Alice lives, that Bob lives, and that Cindy lives. We can solve each player's problem independently, without caring about the others. The answer is

$$\text{prob Alice lives} \times \text{prob Bob lives} \times \text{prob Cindy lives}.$$

Proof

If anyone dies, then the game is over anyway.

If anyone wins early, then they can just sit on square n indefinitely and wait for the others to catch up.

Otherwise, notice that their moves do not affect each other.

So, the problem asks: What is the probability that someone with A health points makes it to the end, starting from square 0 (and resp. for B and C later). Let's consider the probability tree of all possibilities:

- $1/36$ to roll a 2, in which case we now want the prob. that Alice lives with $A - d_2$ HP from square 2.
- $2/36$ to roll a 3, in which case we now want the prob. that Alice lives with $A - d_3$ HP from square 3.
- $\vdots$
- $2/36$ to roll a 11, in which case we now want the prob. that Alice lives with $A - d_{11}$ HP from square 11.
- $1/36$ to roll a 12, in which case we now want the prob. that Alice lives with $A - d_{12}$ HP from square 12.

The complete probability tree has exponentially many nodes, but you will still be able to get points from the first two subtasks at least.

Implementation

Let `prob(HP, k)` compute the probability that someone survives Jumanji, if they start with that much HP from square k.
As illustrated earlier, you can solve this problem by breaking it down into similarly-shaped subproblems, so we can solve this using recursion.

```
def prob(HP, k):
    if k >= n:
        return 1.0
    HP -= damage[k]
    if HP <= 0:
        return 0.0

    ans = 0
    for roll_1 in [1, 2, 3, 4, 5, 6]:
        for roll_2 in [1, 2, 3, 4, 5, 6]:
            ans += prob(HP, k + roll_1 + roll_2) / 36.0
    return ans
```

To get full points, we note that `prob(HP, k)` is a *pure* mathematical function. Its return value is entirely determined by HP and k, and is *always the same* when the same HP and k are passed into it as input.

So, we can *cache* those values to make sure each (HP, k) pair is evaluated only at most once. If `prob(HP, k)` is ever called on input values we've computed already, just lookup the stored value instead of computing it all over again.

The running time is $O(36 \cdot \max(HP) \cdot n)$—the time needed to evaluate each state muliplied by the number of unique states. This gets 100 pts!

Standard Toolbox: DP

Congratulations! You just did DP!