837. New 21 Game

Medium Math Dynamic Programming Sliding Window

Problem Description

In this problem, Alice plays a game similar to "21," where her objective is to accumulate points through random draws without exceeding a certain threshold. Alice starts with 0 points and continues to draw integers from 1 up to maxPts until the total points reach or exceed k. The goal of the problem is to calculate the probability that Alice finishes the game with n or fewer points. The points gained from each draw are independent and have equal probability. One key condition is that if Alice's score is already k or more, she stops drawing cards. The result should be accurate within a tolerance level of 10^-5.

Probability and Statistics

The solution involves using dynamic programming, specifically in a recursive manner with memoization to cache previously

Intuition

explore each possible sum Alice could attain starting from 0 and moving forward. The algorithm starts with a function dfs that takes the current score i as an argument and returns the probability of Alice having n or fewer points when starting with i points. If i is greater than or equal to k, Alice can no longer draw cards, and we return 1

computed results, as seen by the <a>@cache decorator. The process can be thought of as a depth-first search (DFS), where we

if \mathbf{i} is less than or equal to \mathbf{n} (successful outcome), and 0 otherwise (failure). The base cases cover when i is k or more (stop drawing), and a special optimization when i is exactly k - 1. In this case, Alice has only one draw left, so we calculate the probability based on how many points from 1 to maxPts would be successful.

The recursive step of the algorithm tries to calculate the probabilities for the next draw based on previous calculations, thus reducing the number of calculations needed and using the principle of mathematical expectation. It combines the probabilities of

all possible outcomes from the next draw and subtracts the probabilities beyond the maximal points Alice needs to win, as she

stops drawing after reaching k. By calling dfs(0), we're initiating our search from a score of 0 and climbing up to find out what the probability would be for Alice to end up with n or fewer points, which provides us with the final answer.

Solution Approach The solution provided here is a recursive implementation that makes use of dynamic programming principles and incorporates memoization to store the results of previously computed probabilities. This is to prevent recomputing the probability for each

score, which would otherwise lead to a very inefficient algorithm with a lot of repeated work.

The data structure used for memoization is the internal cache provided by Python's functools library, which is applied using the @cache decorator on top of the recursive function dfs.

Now, breaking down the solution:

by maxPts which is the range of outcomes for a single draw.

• We first look at dfs(20) (which is already established as 1/10).

• dfs(i: int) -> float: This is the main function that computes the probability of Alice ending the game with i points by drawing numbers. • If i is greater than or equal to k, Alice must stop drawing cards. Therefore, the probability of Alice having n or fewer points is 1 if i is less than or equal to n, and zero otherwise because she can't have more than n points under the given conditions.

• The case of i == k - 1 is a special base case, which accounts for the last draw before reaching the stopping point at k. In this situation, it's

possible to precisely calculate the probability because there's only one possible draw. It is the ratio of the count of numbers leading to a total

score of n or fewer points to the maxPts. • For other cases, the solution uses the recursive formula dfs(i + 1) + (dfs(i + 1) - dfs(i + maxPts + 1)) / maxPts. Here's what it does: 1. dfs(i + 1) gives the probability starting at the next point.

2. Then we include the probabilities for each point between i + 1 and i + maxPts, ensuring that we account for the varying outcomes of the

next draw. This is accomplished by adding (dfs(i + 1) - dfs(i + maxPts + 1)) / maxPts, which normalizes the sum of probabilities over the range of possible next draws, by subtracting the sum from the point that is maxPts + 1 steps away from the current point and dividing

efficiently solve the problem of calculating the chance of Alice's success in this card-game-based simulation.

Let's assume $\max Pts = 10$, k = 21, and n = 20 for our simplified example to illustrate the solution approach.

The function dfs(0) is called to start the process from a score of 0 and computes the desired probability using this algorithm. It leverages the cache to store intermediate calculations, making the solution efficient enough to handle larger inputs. In summary, this solution approach utilizes a combination of recursion, memoization, and mathematical probability calculations to

Step 1: Initial Call We start by calling dfs(0) which signifies that Alice is starting the game with 0 points, and we wish to find out the probability that she ends up with 20 points or fewer by the time she decides to stop drawing cards.

Step 2: Recursive Calls and Memoization When dfs(i) is called, we first check if i is greater than or equal to k (which is 21 in

our example). If i is 21 or more, we return 1 if i is less than or equal to n, which is not the case here so recursion won't start at

Step 3: Handle Special Case When i is exactly 20 (which is k - 1), the function checks the special case. At i = 20, Alice can

only make one draw. She succeeds if she draws 1 point, which is the only possibility within the given maxPts without exceeding n. Thus, the probability in this case would be 1 / maxPts, which is 1 / 10.

21 or more.

Example Walkthrough

Step 4: Recursion for General Case For any i less than 20, we calculate the probability recursively. Take dfs(19) for example:

compound probability considering all the paths of play Alice could take is the final result returned to the caller.

dfs(30) here, but since 30 >= k, its value is 0. • Thus, for dfs(19), the probability becomes dfs(20) + (dfs(20) - dfs(30)) / maxPts. Substituting the known values, we get 1/10 + (1/10 -0) / 10 which is 1/10 + 1/100 equal to 0.11 (or 11/100).

Step 5: Continue until Base Case or Cache Hit The algorithm continues like this for dfs(18), dfs(17), ... until it either

reaches a base case or hits a cached value. At each step, it uses the probabilities from the higher points to calculate the current

• Then, we consider the recursion step, (dfs(19 + 1) - dfs(19 + maxPts + 1)) / maxPts. We need to calculate dfs(19 + maxPts + 1) which is

point's probability while ensuring it doesn't calculate the same value multiple times thanks to memoization. Step 6: Final Result This continues until we reach back to the initial call, dfs(0), at which point all required probabilities have been cached and are used to find the probability that Alice ends the game with 20 or fewer points starting from 0. This

from functools import lru_cache class Solution: def new21Game(self, N: int, K: int, maxPoints: int) -> float: # Use an LRU cache to cache results and avoid re-computation. @lru cache(None)

return dfs(current_points + 1) + (dfs(current_points + 1) - dfs(current_points + maxPoints + 1)) / maxPoints

Recursively calculate the probability of winning by either # drawing a card of points 1 up to maxPoints from the current score. The probability # is a difference of probabilities of getting the next score, and the score that is # 'maxPoints' away from current + 1, because that's no longer reachable in one draw.

return dfs(0)

Solution Implementation

def dfs(current points: int) -> float:

return float(current_points <= N)</pre>

if current points >= K:

if current points == K - 1:

Base condition: If current points are at or beyond K,

the game stops; return 1.0 if not beyond N, otherwise 0.0.

If we are at the last point before K, the probability of

points would lead to a score within the [K, N] range,

return min(N - K + 1, maxPoints) / maxPoints

getting a final score not more than N depends on how many

divided by the maximum number of points we can get at this draw.

The total is divided by maxPoints to get the average probability.

private double[] probabilityLookup; // Cached probabilities for intermediate results.

Start DFS from 0 points to calculate the probability of winning.

private int maxFinalPoints, pointsToStop, maxPointsPerDraw;

// Calculates the probability of winning the 21 game.

for (int i = k; $i < k + maxPts && i <= n; ++i) {$

// Iterate backwards to fill in the dp vector.

// Sliding window for the sum of probabilities.

points would lead to a score within the [K, N] range,

Recursively calculate the probability of winning by either

Start DFS from 0 points to calculate the probability of winning.

return min(N - K + 1, maxPoints) / maxPoints

if current points == K - 1:

divided by the maximum number of points we can get at this draw.

The total is divided by maxPoints to get the average probability.

drawing a card of points 1 up to maxPoints from the current score. The probability

is a difference of probabilities of getting the next score, and the score that is

'maxPoints' away from current + 1, because that's no longer reachable in one draw.

wSum += 1.0; // Only scores <= n can lead to a win.

double new21Game(int n, int k, int maxPts) {

// Initialize the base cases

dp[k - 1] = wSum / maxPts;

for (int i = k - 2; $i \ge 0$; --i) {

dp[i] = wSum / maxPts;

double result;

if (k > 0) {

result = dp[0];

return result;

dp[i] = 1.0;

Java

class Solution {

Python

```
// The new21Game method where the game's calculation starts.
    public double new21Game(int maxFinalPoints, int pointsToStop, int maxPointsPerDraw) {
        this.maxFinalPoints = maxFinalPoints;
        this.pointsToStop = pointsToStop;
        this.maxPointsPerDraw = maxPointsPerDraw;
        this probabilityLookup = new double[pointsToStop]; // Cache array initialized for stopping points.
        return calculateProbability(0); // Start calculating from point 0.
    // Recursive method to calculate the probability of winning.
    private double calculateProbability(int currentPoints) {
       // If we've reached the stopping point, return 1 if it's a win, 0 if it's a loss.
        if (currentPoints >= pointsToStop) {
            return currentPoints <= maxFinalPoints ? 1 : 0;</pre>
        // Base case for the stopping point.
        if (currentPoints == pointsToStop - 1) {
            return Math.min(maxFinalPoints - pointsToStop + 1, maxPointsPerDraw) / (double) maxPointsPerDraw;
       // If we've already calculated the probability for these points, return it.
        if (probabilityLookup[currentPoints] != 0) {
            return probabilityLookup[currentPoints];
        // Recursive call and formula to calculate the probability.
        // We advance one point and subtract the probability of going out of bounds, normalized by the max points per draw.
        probabilityLookup[currentPoints] = calculateProbability(currentPoints + 1)
            + (calculateProbability(currentPoints + 1) - calculateProbability(currentPoints + maxPointsPerDraw + 1))
            / maxPointsPerDraw;
        return probabilityLookup[currentPoints];
C++
#include <vector>
#include <functional>
class Solution {
public:
```

// The last score leading to a win is from k-1, if there are more window states than needed, only count as much as n-k+1.

std::vector<double> dp(k + maxPts, 0.0); // Dynamic programming vector initialized with zeros.

double wSum = 0.0; // Window sum to hold the sum of probabilities of the last 'maxPts' states.

wSum += dp[i]; // When sliding the window, add the current dp value to the window sum.

wSum -= dp[i + maxPts]; // Remove the dp value going out of the window.

// The result is the probability of being in state 0 with the game continuing.

};

```
TypeScript
// Holds memoized results for optimization
const memo: number[] = new Array(k).fill(0);
// Define a recursive function to calculate the probability
// i: current sum of points
function dfs(i: number): number {
    // Base case: When i is at least k but not greater than n, it means the game ends successfully
    if (i >= k) {
        return i <= n ? 1 : 0;
    // Edge case: When we're at the last draw that might push the score over k
    if (i === k - 1) {
        // The probability will be the fraction of possible points that won't exceed n
        return Math.min(n - k + 1, maxPts) / maxPts;
    // If the value is already computed, return it to avoid re-calculation
    if (memo[i] !== 0) {
        return memo[i];
    // Recursive relation:
    // The probability of reaching current 'i' is the probability of 'i+1'
    // plus the correction term which accounts for the sliding window effect of the next 'maxPts' states
    // This term is divided by 'maxPts' which represents the uniform probability of drawing each point from 1 to maxPts.
    memo[i] = dfs(i + 1) + (dfs(i + 1) - dfs(i + maxPts + 1)) / maxPts;
    return memo[i];
// Begin the game from zero points
return dfs(0);
from functools import lru_cache
class Solution:
    def new21Game(self, N: int, K: int, maxPoints: int) -> float:
        # Use an LRU cache to cache results and avoid re-computation.
        @lru cache(None)
        def dfs(current points: int) -> float:
            # Base condition: If current points are at or beyond K.
            # the game stops; return 1.0 if not beyond N, otherwise 0.0.
            if current points >= K:
                return float(current_points <= N)</pre>
            # If we are at the last point before K, the probability of
            # getting a final score not more than N depends on how many
```

The given code snippet is an implementation of a dynamic programming solution to solve the 21 Game problem. The time and space complexity analysis is as follows:

Time and Space Complexity

1) to calculate the current state's probability.

return dfs(0)

The time complexity of the dfs function is based on two factors—the range of possible scores [0, k) and the number of recursive calls made for each score. Let's analyze it step by step:

return dfs(current_points + 1) + (dfs(current_points + 1) - dfs(current_points + maxPoints + 1)) / maxPoints

1. We can have at most k different states since the recursion stops once i >= k. 2. For each state i, the dfs function is memoized—meaning results of previous calls are cached to avoid re-computation. 3. For each state i, the dfs function makes a recursive call to dfs(i + 1) and accesses the cached results of dfs(i + 1) and dfs(i + maxPts +

Space Complexity:

Time Complexity:

4. Since we're caching the results, each state is computed only once, and the recursion has a depth of at most maxPts. Combining these observations, the time complexity can be calculated as 0(k * maxPts), where 'k' is the number of states up to

when the game stops and maxPts is the number of recursive branches we explore before hitting memoized states.

1. We're using memoization, and therefore need cache space for each potential state from [0, k) which gives us 0(k).

- The space complexity is determined mostly by the space needed for caching the results of the dfs function and the call stack during the recursion:
- 2. The call stack's maximum depth would be maxPts in the worst-case scenario, due to the nature of the dfs function making recursive calls only maxPts times before reaching a memoized value or a base case.

So, the space complexity is 0(k + maxPts), which simplifies to 0(k) if k is larger than maxPts, as the cache storage is the dominant term.