1553. Minimum Number of Days to Eat N Oranges

Hard | Mathematics | Dynamic Programming | Leetcode Link

Problem Description

In the given LeetCode problem, you have an initial amount of `n` oranges. Each day, you can choose from one of the three possible actions to eat oranges:

1. Eat one orange.
2. If the remaining number of oranges `n` is divisible by `2`, you can eat `n / 2` oranges.
3. If the remaining number of oranges `n` is divisible by `3`, you can eat `2 * (n / 3)` oranges.

You can only take one action per day. The objective is to find the minimum number of days to eat all the `n` oranges.

This problem falls into the category of dynamic programming and requires a careful choice each day to minimize the total number of days needed.

Intuition

The intuition behind this solution is that in order to minimize the days needed to eat all the oranges, we want to eat as many oranges as possible each day while being adaptable in our strategy as the number of remaining oranges changes. However, since we have multiple options on any given day, we should make the decision that sets us up for future days.

This problem is solved using a Dynamic Programming approach, to store the results of subproblems to avoid calculating them multiple times. Specifically, we use depth-first search (DFS) with memoization, where each state is defined by the remaining number of oranges `n`, and we aim to find the minimum days to finish eating `n` oranges.

The DFS function, `dfs`, takes the current number of oranges and returns the minimum days to eat all of them. When the number of oranges is less than 2, the base case is hit, where we know it takes `n` days to eat `n` oranges (either 0 or 1). Otherwise, we calculate the minimum days by deciding between eating `n % 2 + dfs(n // 2)` on days when `n` is even or `n % 3 + dfs(n // 3)` on days when `n` is divisible by 3. `n % 2` and `n % 3` account for the remaining oranges that can't be eaten by dividing by 2 or 3, which will be eaten one by one.

The `@cache` decorator is used to memoize the results of the DFS function, which reduces the time complexity by storing the outcomes of subproblems and preventing the repetition of the same calculations.

By utilizing this approach, we ensure we're not just greedily taking the option that consumes the most oranges at once but also keeping options open that may lead to fewer days in total, which is achieved by exploring and comparing both divisible by 2 and 3 pathways recursively.

Solution Approach

The solution to this problem makes effective use of a technique known as Depth-First Search (DFS) in combination with memoization in the form of caching. This approach is widely used in dynamic programming problems to efficiently solve complex problems that have overlapping subproblems and optimal substructure, both characteristics found in our orange eating problem. Here's a walk-through of the implementation details:

1. **DFS with Memoization**: We define a helper function, `dfs`, within our `Solution` class which will be used to perform a depth-first search to find the minimum days to eat all `n` oranges. The `@cache` decorator from Python's `functools` module is employed to automatically memoize the results of `dfs` for various values of `n`. Memoization is a critical feature that enables us to save the results of expensive function calls and return the cached result when the same inputs occur again, avoiding the need to compute the same thing multiple times. This significantly reduces the number of calculations and, thus, the run time.
2. **Base Case**: The base case for the `dfs` function is when `n` is less than 2. If `n` is 0 or 1, we know instantaneously how many days it will take to eat all the oranges, which would be exactly `n`, as you simply eat one orange per day until there are none left.
3. **Recursive Cases**: When `n` is greater than or equal to 2, we have to decide each day whether to eat one orange, half of the oranges if `n` is even, or two-thirds of the oranges if `n` is divisible by 3. To make this decision, we compute the number of days it will take for both scenarios:
   - When `n` is even, you can eat `n // 2` oranges plus the days needed to deal with the remaining `n % 2` oranges. This results in the recursive call `dfs(n // 2)` with the added `n % 2` days for leftovers.
   - When `n` is divisible by 3, you can eat `2 * (n // 3)` oranges plus the days needed for the remaining `n % 3` oranges. This leads to the recursive call `dfs(n // 3)` with an additional `n % 3` days for leftovers.
4. **Minimization Strategy**: After calculating both options, we choose the one that gives the minimum number of days. We add 1 to our result to account for the current day's action.
5. **DFS Function Execution**: Finally, we call our `dfs(n)` function from the `minDays` method, which has been provided with the initial number of oranges.
6. **Complexity Analysis**: The time complexity of the DFS function without memoization would be exponential, as it explores two different recursive paths for many values of `n`. However, with memoization, the time complexity drops significantly since we avoid re-computation. The exact time complexity is difficult to determine due to the irregularity of the recursive calls, but it is substantially better than the exponential complexity.

The dynamic programming approach combined with the DFS technique and caching makes an efficient solution that can now solve the given problem in shorter times, even for larger values of `n`.

Example Walkthrough

Let's consider a small example where `n = 10` oranges to illustrate the solution approach.

First, we create a function `dfs` to encapsulate our depth-first search logic. We employ a decorator `@cache` to memoize the results of our function calls to `dfs`.

1. We start with `n = 10`. This is neither divisible by 2 nor 3 without a remainder, so our options are as follows:
   - Eat one orange, leaving us with 9 oranges. (We'll have to decide from there in the next step.)
   - Eat `n // 2 = 5` oranges, leaving 5 with a remainder of 0. (A good option since no remainder is left.)
   - Eat `2 * (n // 3) = 6` oranges, leaving 4 with a remainder of 1. (Not a valid move since `n` is not divisible by 3.)
2. Let's explore the second option from the previous step, where `n` becomes 5 after eating 5 oranges:
   - Now, we have only one option according to the constraints because 5 is not divisible by 2 or 3. So, we eat 1 orange, leaving 4 oranges.
3. Next, `n = 4`. Now, since 4 is divisible by 2, we can eat `n / 2 = 2` oranges, leaving us with 2 oranges. There is no remainder.
4. With `n = 2`, we again can eat `n / 2 = 1` orange. Now we have 1 orange remaining.
5. Finally, we have `n = 1` and eat the last orange.

Throughout this example, our `dfs` function will first be called with `dfs(10)` and then recursively with the updated `n` values. Each time, it compares the different options and chooses the path with the fewer total days.

Counting the days we took each action gives us the minimum number of days to eat all the oranges:

- Eat half (5 oranges)
- Eat one (4 oranges remaining)
- Eat half (2 oranges remaining)
- Eat half (1 orange remaining)
- Eat one (0 oranges remaining)

Thus, in this example, it takes us 5 days to eat all the 10 oranges if we follow the optimal strategy.

The memoization aspect is crucial here, as without it, we might revisit some state multiple times. For example, if we encountered `n = 4` through a different sequence of actions, `@cache` ensures that we use the previously computed result rather than recalculating the minimum days from that point.

Following this process, the `minDays` method will return 5 when provided with the initial number of oranges `n = 10`.

Python Solution

```python
from functools import lru_cache  # Import lru_cache decorator for memoization.

class Solution:
    def minDays(self, n: int) -> int:
        # Define a helper function with memoization to compute the minimum days.
        @lru_cache(maxsize=None)  # Use lru_cache to store function calls with unique parameters.
        def dfs(count):
            # Base case: If count is less than 2, return count itself (0 or 1 day).
            if count < 2:
                return count

            # Recursive step: Calculate using min of two strategies,
            # eating half the oranges if n is even (n % 2 + dfs(n // 2))
            # or eating 2/3 of the oranges if n modulo 3 equals 0 (n % 3 + dfs(n // 3))
            # Either step takes at least 1 day (+1).
            #
            # To eat half the oranges, you either do nothing (if n is even)
            # or eat one orange before you can eat half, so n % 2 is added.
            days_eat_half = 1 + (n % 2 + dfs(n // 2))

            # To eat two-thirds of the oranges, you eat n % 3 oranges before
            # you can eat two-thirds.
            days_eat_two_thirds = 1 + (n % 3 + dfs(n // 3))

            # Return the minimum of the days calculated by the two strategies.
            return min(days_eat_half, days_eat_two_thirds)

        # Call the recursive helper function with the initial count of oranges.
        return dfs(n)

# Example usage
# sol = Solution()
# print(sol.minDays(10))  # Output should be the minimum number of days needed to eat n oranges.
```

Java Solution

```java
class Solution {
    // A memoization map to store already computed results for minimum days.
    private Map<Integer, Integer> memoMap = new HashMap<>();

    // Public method to calculate minimum days to eat n oranges.
    public int minDays(int n) {
        return findMinDays(n);
    }

    // Helper method using DFS and memoization to find the minimum days.
    private int findMinDays(int n) {
        // Base cases: If n is 0 or 1, it takes n days (0 for no oranges, 1 for one orange).
        if (n < 2) {
            return n;
        }

        // If the result for current n is already computed, return it to save time.
        if (memoMap.containsKey(n)) {
            return memoMap.get(n);
        }

        // Recurse to find the minimum days by taking the minimum of the two possible actions:
        // 1. Eat n % 2 oranges and remove half of the remaining oranges
        // 2. Eat n % 3 oranges and remove two-thirds of the remaining oranges
        // Add 1 to the result since it represents action taken for the current day.
        int days = 1 + Math.min(
                // Taking one action: n % 2 (eating) + remaining (recursively calling findMinDays).
                n % 2 + findMinDays(n / 2),
                // Taking another action: n % 3 (eating) + remaining (recursively calling findMinDays).
                n % 3 + findMinDays(n / 3)
        );

        // Store the computed result in the memoMap and return it.
        memoMap.put(n, days);
        return days;
    }
}
```

C++ Solution

```cpp
#include <unordered_map>
using namespace std;

class Solution {
public:
    // Create a memoization map to store results of previously computed subproblems
    unordered_map<int, int> memo;

    // Function to find the minimum number of days to eat n oranges
    int minDays(int n) {
        // Start the recursive depth-first search (DFS) to compute the solution
        return dfs(n);
    }

    // The DFS function that calculates the minimum days to eat the given number of oranges
    int dfs(int n) {
        // Base case: if the number of oranges is less than 2, return 'n' (0 or 1 day)
        if (n < 2) return n;

        // Check if the result for 'n' oranges has already been computed
        if (memo.count(n)) return memo[n];

        // Recursively calculate the days needed by reducing 'n' either by eating 'n/2' oranges
        // (using a minimum of one day plus days needed for remaining after eating 'n/2' oranges)
        // or by eating 'n/3' oranges (minimum of one day plus days needed for remaining after eating 'n/3' oranges)
        // Use the '%' operator to calculate the cost (in additional days) of getting to a number that is
        // divisible by 2 or 3 (e.g., if n % 2 == 1, we need one extra day to reach an even number)
        int res = 1 + min(n % 2 + dfs(n / 2), n % 3 + dfs(n / 3));

        // Memoize the result for 'n' before returning
        memo[n] = res;

        // Return the computed minimum number of days
        return res;
    }
};
```

Typescript Solution

```typescript
// Map to store results of previously computed subproblems for memoization
const memo: Map<number, number> = new Map();

// Function to find the minimum number of days to eat "n" oranges
function minDays(n: number): number {
    // Start the recursive depth-first search (DFS) to compute the solution
    return dfs(n);
};

// The DFS function that computes the minimum days to eat the given number of oranges
function dfs(n: number): number {
    // Base cases: if the number of oranges is less than 2, return "n" (0 or 1 day)
    if (n < 2) return n;
    // Check if the result for "n" oranges has already been computed and is in the memo
    if (memo.has(n)) return memo.get(n);
    // Recursively calculate the days needed by reducing "n" through eating "n / 2" or "n / 3" oranges
    // Add 1 to account for the day needed to eat the additional days needed to make "n" divisible by 2 or 3
    const byTwo: number = n % 2 + dfs(Math.floor(n / 2));
    const byThree: number = n % 3 + dfs(Math.floor(n / 3));
    const result: number = 1 + Math.min(byTwo, byThree);
    // Memoize the result for "n" before returning
    memo.set(n, result);
    // Return the computed minimum number of days
    return result;
};
```

Time and Space Complexity

The given Python function `minDays` employs a depth-first search strategy with memoization (cache) to find the minimum number of days required to reduce the input number `n` to 0 with the operations: subtract 1, divide by 2, or divide by 3 when possible.

Time Complexity

The time complexity of this function can be observed by the number of sub-problems it evaluates. Due to memoization, each number from `n` to 0 needs to be computed at most once. At each step, there are up to two recursive calls - one for the division by 2 case and one for the division by 3 case. However, due to memoization and the nature of the reductions (multiplicative), the actual number of unique sub-problems encountered forms something akin to a Fibonacci sequence, which grows exponentially.

For a tight upper bound, consider the worst-case scenario where `n` is just below a power of 3, say 3^k, then the complexity would have a similar worst-case upper bound to that of O(log3(n)). Similarly, if `n` is just above a power of 2, the bound would be akin to O(log2(n)). Therefore, given that the memoization ensures each subproblem is calculated once, the time complexity is bounded by O(log(n)).

Space Complexity

The space complexity is determined by the size of the call stack and the space used by the cache. The stack's depth correlates to the recursive calls, and because of the memoization, it will only go as deep as the number of unique sub-problems, which, as analyzed, is at most O(log(n)).

Additionally, the cache requires space for each unique sub-problem. Given the discussion on time complexity, we know that in the worst-case scenario we would cache all numbers descending from `n` to 1 using divisions by 2 and 3, leading to potentially O(log(n)) unique entries.

Combining the call stack and cache space, the overall space complexity can also be considered O(log(n)).