2403. Minimum Time to Kill All Monsters

Problem Description

In this problem, you are tasked with defeating all the monsters in an array `power`, where `power[i]` represents the power level of the $i-th$ monster. You start with `0` mana points, and each day, you gain more mana points. Initially, you gain `1` mana point per day, and each day this amount increases by `1` after you defeat a monster.

However, you can only defeat a monster on a given day if your mana points exceed or are equal to that monster's power. If you defeat a monster, your mana points reset to `0`, and the daily mana gain (`gain`) increases by `1`. The goal is to find the minimum number of days required to defeat all monsters in the array.

To put it simply, the problem asks you to optimize the order of defeating monsters while considering the growth of your mana points so that all monsters can be defeated in the least number of days possible.

Intuition

The solution approach for this problem can be understood in terms of a depth-first search (DFS) through all possible scenarios. Since each day you can choose to defeat any monster whose power level is less than or equal to your current mana, there could be multiple choices, and thus multiple possible futures, to consider.

Given that each time you defeat a monster, your mana resets, the number of days needed depends on the order in which you choose to defeat the monsters. Using a bit-mask to represent which monsters have been defeated, we can systematically explore each choice. A bit-mask is a binary representation where each bit corresponds to a monster; if the bit is `1`, the monster is defeated, and if it's `0`, the monster remains.

The function `dfs(mask)` explores all possible choices from the current state defined by `mask`. If all monsters are defeated (the base case), it returns `0` since no additional days are needed. Otherwise, for each undefeated monster, it calculates the days needed to defeat it plus the days from defeating the remaining monsters and takes the minimum over all these possibilities.

The `@cache` decorator is utilized to memoize results of subproblems making the algorithm more efficient by avoiding redundant calculations. `mask.bit_count()` is used to count how many bits are set to `1` (how many monsters have been defeated) which determines the day's `gain`. `(v + cnt) // (cnt + 1)` calculates the number of days needed to accumulate enough mana to defeat the monster `i` with power `v`.

Essentially, this approach applies recursion to simulate the passage of days, defeat monsters, and track mana gain, recording the minimum days taken to reach the condition where all monsters are defeated.

Solution Approach

The implementation employs a recursive function `dfs(mask)` to walk through all possible ways one could defeat the monsters. Here is the step-by-step explanation of the algorithm:

1. **Recursion and Bitmask**: The primary technique used is recursion combined with bitmasking. Each bit in the `mask` corresponds to a monster. If the bit is set (1), the monster is already defeated; else (0), the monster is yet to be defeated.
2. **Base Case**: The base case for the recursion is when all monsters have been defeated, which is when the `bit_count` of the `mask` is equal to the length of the `power` array. In this case, `dfs` returns `0` because no additional days are required.
3. **Exploring Paths**: For each undefeated monster (those bits that are `0`), the algorithm explores what would happen if you were to defeat this monster next. It does so by calculating the number of days needed to defeat the monster `(v + cnt) // (cnt + 1)` and then recursively calling `dfs` with the monster defeated (`mask | (1 << i)`). The calculation considers the current count of defeated monsters (`cnt`) to determine the daily `gain` in mana points.
4. **Minimum Days**: The `dfs` function looks for the minimum number of days across all such recursive calls for different monsters. The variable `ans` stores the minimum number of days found so far and is updated whenever a smaller value is found.
5. **Memoization**: The algorithm uses memoization with the `@cache` decorator to store results of subproblems. This means that once the number of days required to defeat a certain set of monsters (i.e., a particular `mask`) has been calculated, it won't be recalculated if encountered in the future.
6. **Inf**: The value `inf` from the Python `math` module (which represents "infinity") is used to initialize the `ans` variable. This ensures that the first comparison will always favor the first actual computed number of days over `inf`.
7. **The Initial Recursive Call**: The function `dfs(0)` kicks off the recursion with all monsters undefeated.

In summary, this is a classic brute-force search with optimization. The bitmask acts as a state representation for the set of monsters defeated, and the minimum number of days is computed recursively with memoization to speed up the search process by not recalculating already known states.

Example Walkthrough

To illustrate the solution approach, let's consider an example. Suppose we have the following array of monster powers: `power = [2, 4, 2]`.

This means we have three monsters with power levels 2, 4, and 2, respectively.

Day 1: We start with `0` mana points and a gain of `1` mana point per day. We cannot defeat any monsters on day 1 because our mana is less than the power of any monster.

Day 2: We now have `1` (from Day 1) + `2` mana (since our gain increases daily) = `3` mana points total. We can defeat the first or the third monster (both with power 2), but not the second (with power 4). Let's defeat the first monster. Our mana resets to `0`, but now our mana gain increases to `2` mana points per day.

Mask 1: 001, meaning the first monster has been defeated, and two are left.

Day 3: No mana from previous days since it reset, so you only gain `2` mana points this day. With only `2` mana points, we cannot defeat any remaining monsters, since the next weakest has a power of `2` and we would need to start the day with at least `3` mana points to defeat it (2 + 2 = 4 since we have already defeated one monster, which is our additional mana gain per day).

Day 4: We carry over `2` mana points from Day 3 and gain `3` more (since our gain increases daily), giving us `5` mana points total. Now we can defeat either the second or third monster. Let's choose the third monster. Our mana resets again.

Mask 2: 011, indicating that the first and third monsters are defeated, and only one monster, with power 4, is left.

Day 5: We start again with `0` mana.

Day 6: We gain `3` mana points.

Day 7: We gain `4` mana points (since we defeated 2 monsters already, our gain is `2` and increases daily by 1), totaling `7` mana points, and now we can defeat the second monster with power `4`.

At the end of day 7, we have defeated all monsters.

Mask 3: 111. This means all monsters are defeated and the recursion base case is met.

To calculate the minimum number of days needed, we must consider all the permutation ways to defeat the monsters, which means not only the path we walked through above. The algorithm does just that, by exploring all possibilities and caching the results for efficiency. We end up with:

```
1 dfs(0) -> minimum of (dfs(001), dfs(010), dfs(100)) -> these represent defeating each monster first.
```

Each of these calls will traverse further with their respective masks, accounting for the increasing mana gain and resetting after each defeat. In this simple example, it took us 7 days, but in a different order of defeating monsters, or a different power array, it might take fewer or more days to figure out the least number of days needed. The `dfs` function handles these comparisons and finds the optimal solution.

Python Solution

```python
from functools import lru_cache
from math import inf

class Solution:
    def minimumTime(self, powers: List[int]) -> int:
        # This function uses depth-first search to calculate the minimum
        # to accumulate powers from list `powers`. It uses memoization to
        # store and reuse previously computed results.
        @lru_cache(maxsize=None)  # Using LRU cache for memoization
        def dfs(mask):
            # Count the number of bits set to 1 in `mask` - this represents
            # the number of powers acquired so far.
            count = mask.bit_count()

            # If all powers are acquired, no additional time is needed.
            if count == len(powers):
                return 0

            # Initialize the answer for this subproblem to infinity.
            answer = inf

            # Iterate through each power.
            for i, power_value in enumerate(powers):
                # Check if the power represented by the current index 'i'
                # has already been acquired using the mask.
                if mask & (1 << i):
                    continue

                # Recursive call to explore further by including the current index's power.
                # The time to acquire this power is (power_value + count) divided by (count + 1),
                # representing the efficient use of acquired powers to reduce the time.
                answer = min(answer, dfs(mask | (1 << i)) + (power_value + count) // (count + 1))

            return answer

        # Start DFS with an initial mask of 0 (no powers acquired)
        return dfs(0)
```

Java Solution

```java
class Solution {
    private int taskCount;          // Number of tasks
    private long[] memoization;     // Array to store results of subproblems
    private int[] taskPower;        // Array to hold the power of each task

    /**
     * Calculates the minimum time required to complete all tasks.
     *
     * @param power Array representing the power of each task
     * @return Minimum time to complete all tasks
     */
    public long minimumTime(int[] power) {
        taskCount = power.length; // Initialize the number of tasks
        memoization = new long[1 << taskCount]; // Initialize memoization array to store intermediate results
        Arrays.fill(memoization, -1L); // Fill the array with -1 to denote uncalculated states
        taskPower = power; // Reference the power of tasks
        return dfs(0); // Begin the dfs with an empty mask (no tasks completed)
    }

    /**
     * Performs depth-first search to find the minimum time needed to complete the remaining tasks.
     *
     * @param mask A bitmask representing the state of tasks completion
     * @return Minimum time to complete the tasks represented by the current mask
     */
    private long dfs(int mask) {
        // If we have already computed the answer for this state, return it
        if (memoization[mask] != -1) {
            return memoization[mask];
        }

        int completedTasks = Integer.bitCount(mask); // Count how many tasks have been completed so far
        if (completedTasks == taskCount) { // If all tasks are completed, return 0 as no more time is needed
            return 0;
        }

        long minTime = Long.MAX_VALUE; // Initialize the minimum time to a very large number
        for (int i = 0; i < taskCount; ++i) {
            // If the task 'i' is not completed yet
            if ((mask >> i & 1) == 0) {
                // Calculate time required if we choose to complete task 'i' next
                long time = dfs(mask | (1 << i)) + (long) Math.ceil((double)taskPower[i] / (completedTasks + 1));
                // Update the minimum time
                minTime = Math.min(minTime, time);
            }
        }

        memoization[mask] = minTime; // Store the computed minimum time for this state
        return minTime; // Return the computed minimum time
    }
}
```

C++ Solution

```cpp
using ll = long long;

class Solution {
public:
    // Stores the minimum time for each subset of power-ups used
    vector<ll> minTimeMemo;
    // Store the input power values
    vector<int> heroPowers;
    // The number of heroes
    int n;

    // The function to calculate the minimum time to beat all the monsters
    ll minimumTime(vector<int>& power) {
        n = power.size();

        // Initialize memoization vector for each possible combination of heroes (2^n)
        minTimeMemo.assign(1 << n, -1);

        // Assign the hero powers to a member variable for easy access
        this->heroPowers = power;

        // Start solving the problem using Depth-First Search Approach from an empty mask (no hero used)
        return dfs(0);
    }

    // Recursive function to find the minimum time using Depth-First Search
    ll dfs(int mask) {
        // If we have already calculated the minimum time for this mask, return it to avoid recomputation
        if (minTimeMemo[mask] != -1) return minTimeMemo[mask];

        // Count the number of bits set in the mask (number of heroes that have already fought)
        int cnt = __builtin_popcount(mask);

        // Base case: if cnt == n, all heroes have been used, no time needed beyond this point
        if (cnt == n) return 0;

        // Initialize the answer to maximum possible value to effectively find the minimum
        ll ans = LONG_MAX;

        // Loop through each hero
        for (int i = 0; i < n; ++i) {
            // Check to make sure the hero hasn't already been used (bit not set in mask)
            if ((mask >> i) & 1) continue;

            // Recur to get the minimum time if we use this hero as the next one
            ll timeIfHeroUsed = dfs(mask | (1 << i)) + (heroPowers[i] + cnt) / (cnt + 1);

            // Compare with the best(minimum) answer so far
            ans = min(ans, timeIfHeroUsed);
        }

        // Store the calculated minimum time for this mask for future reference
        minTimeMemo[mask] = ans;
        return ans;
    }
};
```

Typescript Solution

```typescript
function minimumTime(power: number[]): number {
    // The length of the power array.
    const numTasks = power.length;
    // An array to memoize the minimum time for each state.
    const memo = new Array(1 << numTasks).fill(-1);

    // Depth-first search to find the minimum time for a given mask of tasks.
    function dfs(currentMask: number): number {
        // If this state has already been computed, return the memoized result.
        if (memo[currentMask] !== -1) {
            return memo[currentMask];
        }

        // Base case: if all tasks are turned on (i.e., mask has all bits set), return 0.
        const numBitsSet = bitCount(currentMask);
        if (numBitsSet === numTasks) {
            return 0;
        }

        // Set initial minimum time as infinite to ensure it will be replaced by any valid time.
        let minTime = Infinity;

        // Iterate through all tasks.
        for (let i = 0; i < numTasks; ++i) {
            // Skip if the task is already included in the currentMask.
            if ((currentMask >> i) & 1) {
                continue;
            }

            // Make recursive call by including the new task i and calculate the total time.
            // Here we use Math.ceil to divide the power by the number of active tasks.
            const totalTime = dfs(currentMask | (1 << i)) + Math.ceil(power[i] / (numBitsSet + 1));
            // Update minTime if the new calculated time is less than the current minimum.
            minTime = Math.min(minTime, totalTimeIfNewTask);
        }

        // Memoize and return the minimum time for the current mask.
        memo[currentMask] = minTime;
        return minTime;
    }

    // Begin recursion starting with an empty task mask.
    return dfs(0);
}

// Helper function to count the number of bits set to 1 in an integer.
function bitCount(n: number): number {
    let count = 0;
    // Loop until all bits are zero.
    for (; n !== 0; n &= n - 1) {
        ++count;
    }
    return count;
}
```

Time and Space Complexity

The given Python code defines a function `minimumTime` which finds the minimum time required to reduce a given list of powers to zero by deactivating machines in sequence, with the speed of deactivation increasing with each machine that is turned off.

Time Complexity

The time complexity of the code can be determined by analyzing the `dfs` function, which is a depth-first search with memoization. The recursion explores each possible state represented by `mask`, which has a length of n where n is the number of elements in `power`. There are 2^n possible states given that each element can be either active (1) or inactive (0). For every state, the function iterates over all n elements to check which machines can be turned off next. Therefore, the total number of operations is $O(n * 2^n)$.

Memoization is used to ensure that each state is computed only once. This is important because without memoization, the algorithm would have to recompute the same states exponentially more times. With memoization, each state's result is stored and reused, so the total number of distinct function calls matches the number of distinct states, which is 2^n.

Space Complexity

The space complexity is determined by the space required for memoization and the recursion call stack. The memoization stores results for each of the 2^n possible states, so the space complexity for memoization is $O(2^n)$.

The recursion stack's maximum depth would be n, since it represents turning off each machine exactly once. Thus, in terms of space complexity, the recursion stack contributes $O(n)$.

However, as the space used for memoization dominates the space required for the call stack as n grows, the overall space complexity is $O(2^n)$ for storing the results of each of the unique states.