2188. Minimum Time to Finish the Race

Hard | Array | Dynamic Programming | Leetcode Link

Problem Description

In this problem, you are provided with a `tires` array representing different types of tires a racing car can use. Each type of tire has a fatigue factor, meaning each successive lap takes longer to complete than the previous one. A tire's performance is defined by two parameters: `f_i` and `r_i`, where `f_i` is the time it takes to finish the first lap, and `r_i` is the factor by which the time increases for each subsequent lap.

Your goal is to finish a race consisting of `numLaps` as quickly as possible. You can start the race with any tire, and you can change tires between laps, which takes `changeTime` seconds. Changing to a new tire resets the fatigue for that tire, meaning it will perform its first lap at its `f_i` time again. Each tire type can be used an unlimited number of times.

The objective is to determine the minimum time required to complete the race.

Intuition

To solve this problem efficiently, one must combine dynamic programming with a greedy strategy to decide when to change tires.

Tire Performance Cost Calculation

First, we need to understand when it is beneficial to change a tire rather than keep using it. For each tire type, there is a point at which the increasing time due to fatigue makes it slower than just changing to a new tire. We calculate this break-even point under which it's better to keep using the tire and create a cost array representing the minimum time to finish a certain number of laps with a single tire before switching becomes faster.

Dynamic Programming

Next, we use dynamic programming to decide the minimum time to finish `i` laps. The state `f[i]` represents the minimum time to finish `i` laps. For each state, we consider using one of the optimal tire strategies for the last `j` laps (where `j` is derived from the previously calculated tire performance cost), and then find the minimum time taking `f[i - j] + cost[j] + changeTime` over all feasible `j`.

This dynamic programming algorithm will incrementally build up the answer until it gets to `f[numLaps]`, which will be the final answer — the minimum time to finish the race.

Solution Approach

The solution involves both optimization through precalculating the best time for a certain number of laps with each tire and dynamic programming to decide on when to change the tires to result in the minimum total time.

Precalculating Tire Costs

We start by initializing a `cost` list to store the optimal tire usage time for each possible number of successive laps from 1 to a limit where changing tires is more efficient (in the given solution, the limit is 17 laps). We use infinity (`inf`) initially to signify that we have not calculated any times yet.

For each tire type given in the `tires` array, with `f` as the initial lap time and `r` as the fatigue factor, we iterate through successive laps, calculating the time it would take for that tire to complete up to that lap (applying `r_i^(x-1)` to the initial time `f_i` for each lap `x`). We update the `cost` list only if the time calculated (`s`) is less than what's already stored there, thereby keeping the minimum time for each possible number of successive laps.

This calculation is cut off at the point where the time for the next lap would exceed the time it would take to change tires and use a new one (`changeTime + f`).

Dynamic Programming Algorithm

Next, we define a dynamic programming array `f` to store the best time to finish `i` laps. We initialize the first element `f[0]` to `-changeTime` (since we do not need to account for a tire change at the start). We then iterate through the `numLaps` laps to be completed.

For each lap `i`, we consider every possible `j` number of laps that might have been completed with the current tire before changing (limited by the smaller of 17, representing our precalculated limit, and the current lap number `i`). We use these precalculated costs to determine if changing before the `i-th` lap is beneficial. The update is done using `f[i] = min(f[i], f[i - j] + cost[j])`, which represents the minimum time of completing `i-j` laps, then doing `j` laps on a new tire.

Finally, we add `changeTime` to `f[i]` as it represents the added time to change the tire after having completed `i-j` laps.

Conclusion

By iterating through all laps and all feasible `j`s, the `f` array incrementally builds up the solutions until we reach `f[numLaps]`, which gives us the minimum time to finish all `numLaps`. The implementation efficiently combines both the precalculation of an empirical threshold (based on tire performance) and a bottom-up dynamic programming approach to solve the problem.

Example Walkthrough

Let's consider a scenario where we have two tire types defined in the array `tires` as follows: `[[2,3],[3,4]]`. This means the first tire type has an initial lap time `f1=2` seconds and a fatigue factor `r1=3`. The second tire has `f2=3` seconds and `r2=4`. Let's also assume `changeTime=10` seconds, and we want to complete `numLaps=5` laps.

Precalculating Tire Costs

Before we start, we calculate the break-even points for each tire type using the provided intuition.

- For tire type 1: Starting at 2 seconds, the lap times would be 2, 6 (2 * 3), 18 (6 * 3), ... We stop calculating when the next lap time is greater than `changeTime + f1` = 12 seconds. So, we only consider the first two laps for this tire before changing.
- For tire type 2: Starting at 3 seconds, the times are 3, 12 (3 * 4), ... Again, we only consider the first lap for this tire because the second lap is already too slow.

From this precalculation, we determine that for our set of tires, using a tire for only the first lap before changing is the most efficient; this is what our `cost` list reflects.

Dynamic Programming Algorithm

Now we use dynamic programming to find the best strategy to complete all 5 laps. We initialize the array `f` with size `numLaps+1` as `f = [0, inf, inf, inf, inf, inf]`. The first element represents the starting point with no laps completed.

- To complete `1 lap (i=1)`, we check both tires for the first lap, which as per our precalculated costs, would take minimum of `2` and `3` seconds respectively. We do not have to change tires, so we pick the minimum time which is `2` seconds for tire type 1. Thus `f[1] = 2`.
- For `2 laps (i=2)`, since changing tires for each lap is most efficient, we look at the cost for `1 lap` and add a tire change time to that cost for each tire type. The minimum cost would be for tire type 1, which is `2 seconds (previous lap) + 10 seconds (changeTime) + 2 seconds (new lap with tire type 1)` = `14 seconds`. Thus `f[2] = 14`.
- Applying the same logic for `i=3, 4, and 5`, we will find that changing tires every lap ensures the lowest possible time for each i-th lap. Therefore, for `f[3]` it would be `f[2] + changeTime + cost[1]`, and since `cost[1]` is `2` (using tire type 1), `f[3]` becomes `26` seconds; continuing this till `f[5]`, we get `f[5] = 50` seconds.

Conclusion

By combining precalculations of tire performance with dynamic programming, we have found that the minimum time to finish 5 laps using either of the two tire types, with the liberty to change tires after each lap, would be 50 seconds. The dynamic programming approach ensures that we consider every possible scenario at each stage (for each lap) and build upon previous computations to efficiently arrive at the final answer.

Python Solution

```python
from typing import List
from math import inf

class Solution:
    def minimum_finish_time(self, tires: List[List[int]], change_time: int, num_laps: int) -> int:
        # Initialize the minimum cost for each number of laps up to 17
        # These values represent the minimum time to complete a given number of laps without changing tires
        min_cost_for_laps = [inf] * 18

        # Calculate the minimum time to complete successive laps with each tire configuration
        for base_time, decay_factor in tires:
            lap_count, current_cost, time_for_next_lap = 1, 0, base_time
            # Continue if the time to complete the next lap is less than or equal to the time it takes to change the tire plus the ba
            while time_for_next_lap <= change_time + base_time:
                current_cost += time_for_next_lap
                min_cost_for_laps[lap_count] = min(min_cost_for_laps[lap_count], current_cost)
                # Increase the time for the next lap by the decay factor, and increment lap count
                time_for_next_lap *= decay_factor
                lap_count += 1

        # Initialize the dp array to store the minimum time to finish a certain number of laps
        min_time_to_finish_laps = [inf] * (num_laps + 1)
        # Set the base case where 0 laps take 0 time, minus the time for tire change as it's not needed yet
        min_time_to_finish_laps[0] = -change_time

        # Calculate the minimum time to complete i laps
        for lap_i in range(1, num_laps + 1):
            # Consider all possible numbers of laps one could run before changing tires,
            # but it is not worth considering more laps than the tire can handle before decaying, hence the min(18, lap_i + 1) limit
            for consecutive_laps_with_one_tire in range(1, min(18, lap_i + 1)):
                min_time_to_finish_laps[lap_i] = min(
                    min_time_to_finish_laps[lap_i],
                    min_time_to_finish_laps[lap_i - consecutive_laps_with_one_tire] + min_cost_for_laps[consecutive_laps_with_one_tir
                )
            # Add the change time for each tire switch
            min_time_to_finish_laps[lap_i] += change_time

        # Return the minimum time to finish all the laps
        return min_time_to_finish_laps[num_laps]

# The rewritten code now has a clearer naming convention, follows Python 3 syntax,
# and includes comments that explain what each part of the code does.
```

Java Solution

```java
import java.util.Arrays;

class Solution {
    public int minimumFinishTime(int[][] tires, int changeTime, int numLaps) {
        // Initialize the infinity value to be used for the comparison.
        final int infinity = 1 << 30;
        // This array will store the minimum cost for laps up to 17, since for laps 18 and over
        // it's better to change tires than to keep using the same tire.
        int[] minCost = new int[18];
        // Fill the cost array with infinity to later find the minimum.
        Arrays.fill(minCost, infinity);

        // Calculate the minimum cost for each tire for up to 17 laps.
        for (int[] tire : tires) {
            int firstLapTime = tire[0];
            int rotFactor = tire[1];
            int cumulativeTime = 0;
            int lapTime = firstLapTime;

            // Loop to calculate the time of using the same tire consecutively without changing.
            for (int i = 1; lapTime <= changeTime + firstLapTime; ++i) {
                cumulativeTime += lapTime;
                minCost[i] = Math.min(minCost[i], cumulativeTime);
                lapTime *= rotFactor; // Increase lap time by rotation factor for the next lap
            }
        }

        // Initialize dp array to store the best time to complete i laps.
        int[] f = new int[numLaps + 1];
        // Fill the dp array with infinity.
        Arrays.fill(f, infinity);
        // Time to finish 0 lap is 0 minus the change time to account for the initial starting point (no tire change before the rac
        f[0] = -changeTime;

        // Compute the minimum time to finish each number of laps.
        for (int i = 1; i <= numLaps; ++i) {
            // Try every possible last stint that spans j laps (where j < 18 and j <= current lap number).
            for (int j = 1; j < Math.min(18, i + 1); ++j) {
                f[i] = Math.min(f[i], f[i - j] + minCost[j]);
            }
            // Add the tire change time since every stint ends with changing tires unless it's the final one.
            f[i] += changeTime;
        }

        // Return the minimum time to finish all laps.
        return f[numLaps];
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minimumFinishTime(vector<vector<int>>& tires, int changeTime, int numLaps) {
        int minCost[18];
        memset(minCost, 0x3f, sizeof(minCost)); // Initialize minCost to a large number

        // Populate minCost array with the minimum cost of completing each number of laps (up to 17)
        for (auto& tire : tires) {
            int baseTime = tire[0]; // base lap time for this tire
            int fatigueRate = tire[1]; // rate at which the tire gets slower each lap
            int totalTime = 0;  // total time to complete a certain number of laps with this tire
            long long currentTime = baseTime; // time for the current lap
            for (int laps = 1; currentTime <= changeTime + baseTime; ++laps) {
                totalTime += currentTime;
                minCost[laps] = min(minCost[laps], totalTime);
                currentTime *= fatigueRate; // for the next lap, time increases by fatigueRate
            }
        }

        int dp[numLaps + 1]; // dp[i] will store the minimum time to complete i laps
        memset(dp, 0x3f, sizeof(dp)); // Initialize dp array to a large number
        dp[0] = -changeTime; // base case: no time needed before starting

        // Compute minimum time for each number of laps from 1 to numLaps
        for (int i = 1; i <= numLaps; ++i) {
            // Consider the time to do the last j laps
            for (int j = 1; j < min(18, i + 1); ++j) {
                dp[i] = min(dp[i], dp[i - j] + minCost[j] + changeTime);
            }
        }
        // Return the minimum time to complete numLaps
        return dp[numLaps];
    }
};
```

Typescript Solution

```typescript
function minimumFinishTime(tires: number[][], changeTime: number, numLaps: number): number {
    // Define 'minCostPerLap' to store the minimum cost to complete each lap from 1 to 17 (inclusive).
    // We use 18 because after certain laps, changing tires is cheaper than using worn out tires.
    const minCostPerLap: number[] = Array(18).fill(Infinity);

    // Calculate the minimum cost for each lap for all given tires configurations.
    for (const [firstLapTime, lossFactor] of tires) {
        let cumulativeTime = 0;
        let currentTime = firstLapTime;

        // Loop to calculate and update the minimum cost to complete 'i' laps without changing tires.
        // This loop ends when the cost to run another lap is greater than the cost of changing tires plus the time of the first lap.
        for (let i = 1; currentTime <= changeTime + firstLapTime; ++i) {
            cumulativeTime += currentTime; // Add the current lap time to the cumulative time.
            minCostPerLap[i] = Math.min(minCostPerLap[i], cumulativeTime); // Store the minimum cumulative time.
            currentTime *= lossFactor; // Increment next lap time by the loss factor.
        }
    }

    // Define 'totalCostToCompleteLaps' to store the total cost to complete from 0 to 'numLaps' laps.
    const totalCostToCompleteLaps: number[] = Array(numLaps + 1).fill(Infinity);
    totalCostToCompleteLaps[0] = -changeTime; // Initialize the 0th lap since there is no tire change needed initially.

    // Calculate the total cost to complete 'i' laps.
    for (let i = 1; i <= numLaps; ++i) {
        // Evaluate the minimum cost for completing 'i' laps by either continuing with current tires or changing tires.
        for (let j = 1; j < Math.min(18, i + 1); ++j) {
            totalCostToCompleteLaps[i] = Math.min(
                totalCostToCompleteLaps[i], // Current cost
                totalCostToCompleteLaps[i - j] + minCostPerLap[j] // Cost of (i-j) laps + cost to complete 'j' laps without changing
            );
        }
        // Add the time to change tires for going beyond the 0th lap.
        totalCostToCompleteLaps[i] += changeTime;
    }

    // Return the total cost to complete 'numLaps' laps.
    return totalCostToCompleteLaps[numLaps];
}
```

Time and Space Complexity

Time Complexity

The given code consists of two main parts: calculating the minimum cost to complete a lap using one set of tires for up to 17 consecutive laps and computing the minimum time to finish all `numLaps`.

1. Calculating minimum cost for each lap (up to 17):
   - For each tire configuration, we iterate through laps, recalculating the time taken until it exceeds the change time plus the initial cost `f`.
   - This while loop runs at most until `t <= changeTime + f`, which depends on the rate of growth determined by `r`. The maximum number of iterations is limited to 17, as we stop adding laps costs once we reach this number (`cost` array size).
   - Since there are `T` tire configurations, the time complexity for this part becomes O(17T).
2. Computing minimum time to complete `numLaps`:
   - We iterate over each lap from 1 to `numLaps` inclusive.
   - For each lap, we iterate again for a maximum of 17 times (which is the maximum consecutive laps before a tire change is needed).
   - At each inner loop iteration, we perform a constant number of operations seeking the minimum cost.
   - The time complexity for this nested loop is O(numLaps * 17).

Combining the two, we have the final time complexity: `O(17T) + O(numLaps * 17)`, simplifying this down to the major terms gives us `O(T + numLaps)`.

Space Complexity

The space complexity of the given solution includes:

1. The `cost` array, which is of size 18 (constant size), giving O(1).
2. The `f` array, which is of size `numLaps + 1` to store the minimum time to finish every possible number of laps.

Therefore, the overall space complexity is `O(numLaps)` because the size of `f` array scales with the input `numLaps`, which is the dominant term in space usage.