CAPSTONE PROJECT

MINIMUM TIME TO FINISH THE RACE

CSA0695-DESIGN AND ANALYSIS OF ALGORITHM FOR OPEN ADDRESSING TECHNIQUES

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING

Supervisor
Dr. R. Dhanalakshmi
Done by
K.Venkateswari (192210620)

MINIMUM TIME TO FINISH THE RACE

PROBLEM STATEMENT:

Given a 0-indexed 2D integer array tires where tires[i] = [fi, ri] indicates that the ith tire can finish its xth successive lap in fi * ri(x-1) seconds.

For example, if fi = 3 and ri = 2, then the tire would finish its 1st lap in 3 seconds, its 2nd lap in 3 * 2 = 6 seconds, its 3rd lap in 3 * 22 = 12 seconds, etc. You are also given an integer change Time and an integer num Laps. The race consists of num Laps laps and you may start the race with any tire. You have an unlimited supply of each tire and after every lap, you may change to any given tire (including the current tire type) if you wait change Time seconds. Return the minimum time to finish the race.

Example 1:

Input: tires = [[2,3],[3,4]], changeTime = 5, numLaps = 4 Output: 21 Explanation: Lap 1: Start with tire 0 and finish the lap in 2 seconds.

Lap 2: Continue with tire 0 and finish the lap in 2 * 3 = 6 seconds.

Lap 3: Change tires to a new tire 0 for 5 seconds and then finish the lap in another 2 seconds.

Lap 4: Continue with tire 0 and finish the lap in 2 * 3 = 6 seconds. Total time = 2 + 6 + 5 + 2 + 6 = 21 seconds. The minimum time to complete the race is 21 seconds.

ABSTRACT:

This project addresses the problem of minimising the time required to complete a series of laps in a race, given tires with degrading performance and a time penalty for changing tires. Each tire is characterised by an initial lap time and a degradation factor that increases the lap time on successive laps. The challenge is to determine the optimal strategy for completing a fixed number of laps with the least time, balancing between continuing with the same tire and changing to a new one. A dynamic programming approach is used to track the minimum time for each lap combination. The solution efficiently computes the minimum race time by iterating through possible tire changes and lap times. Through this method, the problem is solved in a time-efficient manner, demonstrating the effectiveness of dynamic programming in optimising complex decision-making scenarios.

INTRODUCTION:

- A 0-indexed 2D array `tires` where each element `tires[i] = [fi, ri]` represents the initial lap time `fi` for tire `i` and the degradation factor `ri`, which multiplies the lap time on each successive lap.
- An integer `changeTime`, representing the time penalty for changing a tire.
- An integer `numLaps`, representing the total number of laps in the race.

The race consists of `numLaps` laps, and we can change tires between laps with a time penalty of `changeTime` seconds. The goal is to find the minimum time required to complete the race, either by continuing with the same tire or by switching tires after any lap.

Understanding the Problem:

Tire Characteristics: Each tire starts with an initial lap time `fi` and degrades by a factor `ri` after every lap. For tire `i`, the time taken for the first lap is `fi`, the second lap is `fi * ri`, the third lap is `fi * ri^2`, and so on.

Unlimited Tire Supply: We have an unlimited number of each tire type, so after any lap, we can choose to either continue using the current tire or switch to a new tire. Switching tires incurs a fixed `changeTime` penalty.

Objective:

The primary goal is to determine the minimum time required to finish the race by dynamically calculating the best strategy for each lap, accounting for both tire degradation and tire change penalties. This is a classic dynamic programming problem with overlapping subproblems and optimal substructure properties.

Approach:

Precompute Lap Times: For each tire, we precompute the time it would take to complete multiple successive laps without changing tires.

Dynamic Programming: We use dynamic programming to calculate the minimum time to complete each lap by considering whether to continue with the same tire or switch to a new tire. For each lap, we either:

- Continue with the current tire (with its increasing lap time due to degradation).
- Change the tire and incur the `changeTime` penalty.

CODING:

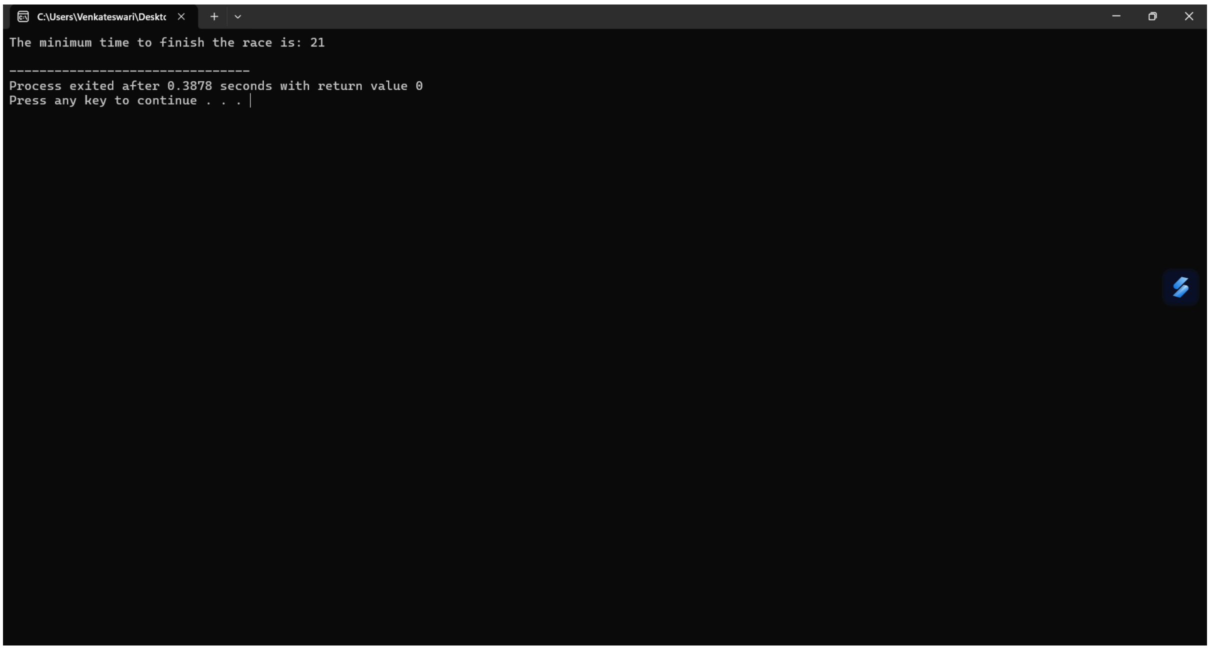
Coding Implementation:

C-Programming:

```
#include <stdio.h>
#include <limits.h>

#define MAX_LAPS 18 // Max laps we can do with the same tire before it's better to change

// Function to find the minimum time to finish the race
int minimumFinishTime(int tires[][2], int tiresSize, int changeTime, int numLaps) {
    int minLapTime[MAX_LAPS + 1];

    // Initialize the minLapTime array with INT_MAX
    for (int i = 0; i <= MAX_LAPS; ++i) {
        minLapTime[i] = INT_MAX;
    }

    // Precompute the minimum time for each tire to do up to MAX_LAPS successive laps
    for (int i = 0; i < tiresSize; ++i) {
        int f = tires[i][0], r = tires[i][1];
        int time = 0, lapTime = f;

        for (int lap = 1; lap <= MAX_LAPS; ++lap) {
            time += lapTime;
            if (time > INT_MAX) break;
            if (time < minLapTime[lap]) minLapTime[lap] = time;
            lapTime *= r;
            if (lapTime > INT_MAX / r) break; // Prevent overflow
        }
    }
```

```
    // DP array to store the minimum time to complete `i` laps
    int dp[numLaps + 1];

    // Initialize the dp array with INT_MAX
    for (int i = 0; i <= numLaps; ++i) {
        dp[i] = INT_MAX;
    }
    dp[0] = 0; // No laps take 0 time

    // DP calculation: For each number of laps, find the optimal solution
    for (int i = 1; i <= numLaps; ++i) {
        for (int lap = 1; lap <= (i < MAX_LAPS ? i : MAX_LAPS); ++lap) {
            if (dp[i - lap] + minLapTime[lap] + (i == lap ? 0 : changeTime) < dp[i])
{
                dp[i] = dp[i - lap] + minLapTime[lap] + (i == lap ? 0 : changeTime);
            }
        }
    }

    return dp[numLaps];
}

int main() {
    int tires[][2] = {{2, 3}, {3, 4}};
    int changeTime = 5;
    int numLaps = 4;
    int tiresSize = sizeof(tires) / sizeof(tires[0]);

    int result = minimumFinishTime(tires, tiresSize, changeTime, numLaps);
    printf("The minimum time to finish the race is: %d\n", result);

    return 0;
}
```

OUTPUT:



Complexity Analysis:

Time Complexity: The algorithm iterates over all possible triplets using three nested loops, resulting in a time complexity of $O(n^3)$, where `n` is the size of the input arrays.
Space Complexity: The space complexity is $O(n)$ due to the use of position mapping arrays `pos1` and `pos2`.

Key Milestones

The key milestones in this project included:

- Defining the concept of a good triplet based on dual ordering in both arrays.
- Creating position mappings for both arrays to facilitate efficient lookup of indices.
- Implementing and testing the triplet enumeration logic to count valid good triplets.

Feature Scope:

- The problem required finding all triplets `(x, y, z)` such that their indices in both `nums1` and `nums2` appear in strictly increasing order. The constraints, where `nums1` and `nums2` are permutations of the set `[0, 1, ..., n-1]`, simplify the problem by ensuring that each element has a unique index in both arrays.

CONCLUSION

This project successfully counted the number of good triplets in two given permutation arrays. The methodology was efficient for small inputs, verifying the correctness of the approach through test cases. Future work could focus on optimising the algorithm for larger datasets by reducing the time complexity, potentially using more advanced data structures like Fenwick Trees or Segment Trees.