<https://leetcode.com/problems/maximum-earnings-from-taxi/description/>

There are n points on a road you are driving your taxi on. The n points on the road are labeled from 1 to n in the direction you are going, and you want to drive from point 1 to point n to make money by picking up passengers. You cannot change the direction of the taxi.

The passengers are represented by a **0-indexed** 2D integer array rides, where rides[i] = [starti, endi, tipi] denotes the ith passenger requesting a ride from point starti to point endi who is willing to give a tipi dollar tip.

For**each**passenger i you pick up, you **earn** endi - starti + tipi dollars. You may only drive **at most one**passenger at a time.

Given n and rides, return **the maximum number of dollars you can earn by picking up the passengers optimally.**

**Note:** You may drop off a passenger and pick up a different passenger at the same point.

**Example 1:**

**Input:** n = 5, rides = [[2,5,4],[1,5,1]]

**Output:** 7

**Explanation:** We can pick up passenger 0 to earn 5 - 2 + 4 = 7 dollars.

**Example 2:**

**Input:** n = 20, rides = [[1,6,1],[3,10,2],[10,12,3],[11,12,2],[12,15,2],[13,18,1]]

**Output:** 20

**Explanation:** We will pick up the following passengers:

- Drive passenger 1 from point 3 to point 10 for a profit of 10 - 3 + 2 = 9 dollars.

- Drive passenger 2 from point 10 to point 12 for a profit of 12 - 10 + 3 = 5 dollars.

- Drive passenger 5 from point 13 to point 18 for a profit of 18 - 13 + 1 = 6 dollars.

We earn 9 + 5 + 6 = 20 dollars in total.

**Constraints:**

1 <= n <= 10^5

1 <= rides.length <= 3 \* 10^4

rides[i].length == 3

1 <= starti < endi <= n

1 <= tipi <= 10^5

**Attempt 1: 2023-12-18**

**Solution 1: Native DFS (10 min, TLE 54/84)**

class Solution {

    public long maxTaxiEarnings(int n, int[][] rides) {

        Arrays.sort(rides, (a, b) -> a[0] - b[0]);

        return helper(rides, 0);

    }

    private long helper(int[][] rides, int index) {

        if(index >= rides.length) {

            return 0;

        }

        long notTakeCurrentRide = helper(rides, index + 1);

        int i;

        for(i = index + 1; i < rides.length; i++) {

            // The concatenation ride start label should only

            // at or after current ride end label

            if(rides[i][0] >= rides[index][1]) {

                break;

            }

        }

        long takeCurrentRide = rides[index][1] - rides[index][0] + rides[index][2] + helper(rides, i);

        return Math.max(notTakeCurrentRide, takeCurrentRide);

    }

}

Time Complexity: O(2^m), m is the number of ride intervals

Each recursion level we will take two branches, "not take current ride" and "take current ride", so worst case is we have to go through m depth recursion, then time complexity is O(m^2)

Space Complexity: O(m)

Recursion Stack: The depth of recursion may go up to m in the worst case if all rides are taken sequentially with no overlap, which contributes to O(m) space complexity.

**Solution 2: DFS + Memoization (10 min)**

class Solution {

    public long maxTaxiEarnings(int n, int[][] rides) {

        Arrays.sort(rides, (a, b) -> a[0] - b[0]);

        Long[] memo = new Long[rides.length];

        return helper(rides, 0, memo);

    }

    private long helper(int[][] rides, int index, Long[] memo) {

        if(index >= rides.length) {

            return 0;

        }

        if(memo[index] != null) {

            return memo[index];

        }

        long notTakeCurrentRide = helper(rides, index + 1, memo);

        int i;

        for(i = index + 1; i < rides.length; i++) {

            // The concatenation ride start label should only

            // at or after current ride end label

            if(rides[i][0] >= rides[index][1]) {

                break;

            }

        }

        long takeCurrentRide = rides[index][1] - rides[index][0] + rides[index][2] + helper(rides, i, memo);

        return memo[index] = Math.max(notTakeCurrentRide, takeCurrentRide);

    }

}

Time Complexity: O(m^2), m is the number of ride intervals

First m is for worst case may need m recursion on depth, second m is for each recursion internally we have a for loop to find the next recursion entry

Space Complexity: O(m), m is the number of ride intervals

Memoization: The dfs function uses a memoization, which will store results for each unique argument (i). This could result in up to m entries in the worst case, giving a space complexity of O(m).

Recursion Stack: The depth of recursion may go up to m in the worst case if all rides are taken sequentially with no overlap, which contributes to O(m) space complexity.

So both are O(m), final cost is O(m)

**Solution 3: DP (30 min)**

**Wrong Solution**

class Solution {

    public long maxTaxiEarnings(int n, int[][] rides) {

        Arrays.sort(rides, (a, b) -> a[0] - b[0]);

        int len = rides.length;

        long[] dp = new long[len + 1];

        // In Native DFS, the top is 0(recursion entry), bottom is

        // len(recursion base condition), in bottom up DP (tabulation),

        // we start from bottom and return at top

        dp[len] = 0;

        for(int index = len - 1; index >= 0; index--) {

            for(int i = index + 1; i < len; i++) {

                // The concatenation ride start label should only

                // at or after current ride end label

                if(rides[i][0] >= rides[index][1]) {

                    // 1. dp[index + 1] means not take current ride

                    // 2. rides[index][1] - rides[index][0] + rides[index][2] + dp[i] means take current ride

                    dp[index] = Math.max(dp[index + 1], rides[index][1] - rides[index][0] + rides[index][2] + dp[i]);

                }

            }

        }

        return dp[0];

    }

}

**Correct Solution**

class Solution {

    public long maxTaxiEarnings(int n, int[][] rides) {

        Arrays.sort(rides, (a, b) -> a[0] - b[0]);

        int len = rides.length;

        long[] dp = new long[len + 1];

        // In Native DFS, the top is 0(recursion entry), bottom is

        // len(recursion base condition), in bottom up DP (tabulation),

        // we start from bottom and return at top

        dp[len] = 0;

        // Exactly same logic as Native DFS

        for(int index = len - 1; index >= 0; index--) {

            int i;

            for(i = index + 1; i < len; i++) {

                // The concatenation ride start label should only

                // at or after current ride end label

                if(rides[i][0] >= rides[index][1]) {

                    break;

                }

            }

            // 1. dp[index + 1] means not take current ride

            // 2. rides[index][1] - rides[index][0] + rides[index][2] + dp[i] means take current ride

            dp[index] = Math.max(dp[index + 1], rides[index][1] - rides[index][0] + rides[index][2] + dp[i]);

        }

        return dp[0];

    }

}

Time Complexity: O(m^2), m is the number of ride intervals

First m is for worst case may need m recursion on depth, second m is for each recursion internally we have a for loop to find the next recursion entry

Space Complexity: O(m), m is the number of ride intervals

Recursion Stack: The depth of recursion may go up to m in the worst case if all rides are taken sequentially with no overlap, which contributes to O(m) space complexity.

**The problem for wrong solution happens at inner for loop not break out like the logic in Native DFS, in Native DFS, we have a recursion helper method and input parameter as 'index' which will recursively progress from 'index = 0' to 'index = n - 1', then we initialize a variable 'i' outside the for loop in helper method, and when we detect the first 'i' match condition "rides[i][0] >= rides[index][1]", we "break out" to keep the first 'i' value and continue further recursion, the physical meaning behind the "break out" to keep the first 'i' value is that will be the concatenation point for next ride, which means the next recursion entry. But in bottom up DP (tabulation) if we don't "break out" then if not only one 'i' satisfy "rides[i][0] >= rides[index][1]", the later 'i' will overwrite the first 'i' value, the physical meaning is we will miss the correct concatenation point for next ride, the next recursion will start with wrong entry 'i'.**

**Test case below, step by step:**

Example 2:

Input: n = 20, rides = [[1,6,1],[3,10,2],[10,12,3],[11,12,2],[12,15,2],[13,18,1]]

Output: 20

Explanation: We will pick up the following passengers:

- Drive passenger 1 from point 3 to point 10 for a profit of 10 - 3 + 2 = 9 dollars.

- Drive passenger 2 from point 10 to point 12 for a profit of 12 - 10 + 3 = 5 dollars.

- Drive passenger 5 from point 13 to point 18 for a profit of 18 - 13 + 1 = 6 dollars.

We earn 9 + 5 + 6 = 20 dollars in total.

Correct one:

for(int index = len - 1; index >= 0; index--) {

int i;

for(i = index + 1; i < len; i++) {

// The concatenation ride start label should only

// at or after current ride end label

if(rides[i][0] >= rides[index][1]) {

break;

}

}

// 1. dp[index + 1] means not take current ride

// 2. rides[index][1] - rides[index][0] + rides[index][2] + dp[i] means take current ride

dp[index] = Math.max(dp[index + 1], rides[index][1] - rides[index][0] + rides[index][2] + dp[i]);

}

index = 5 -> i = 6

index = 4 -> i = 6

index = 3 -> i = 4

index = 2 -> i = 4

index = 1 -> i = 2

index = 0 -> i = 2

=============================================================================

Wrong one:

for(int index = len - 1; index >= 0; index--) {

for(int i = index + 1; i < len; i++) {

// The concatenation ride start label should only

// at or after current ride end label

if(rides[i][0] >= rides[index][1]) {

// 1. dp[index + 1] means not take current ride

// 2. rides[index][1] - rides[index][0] + rides[index][2] + dp[i] means take current ride

dp[index] = Math.max(dp[index + 1], rides[index][1] - rides[index][0] + rides[index][2] + dp[i]);

}

}

}

index = 5 -> i NA, 1st error

index = 4 -> i = 5

index = 3 -> i = 4,5, i = 4 result overwrite by i = 5, 2nd error

index = 2 -> i = 4,5, i = 4 result overwrite by i = 5, same 2nd error

index = 1 -> i = 2,3,4,5, i = 2 result overwrite by i = 3,4,5, same 2nd error

index = 0 -> i = 2,3,4,5, i = 2 result overwrite by i = 3,4,5, same 2nd error

**Solution 4: DP + Binary Search (10 min)**

class Solution {

    public long maxTaxiEarnings(int n, int[][] rides) {

        Arrays.sort(rides, (a, b) -> a[0] - b[0]);

        int len = rides.length;

        long[] dp = new long[len + 1];

        // In Native DFS, the top is 0(recursion entry), bottom is

        // len(recursion base condition), in bottom up DP (tabulation),

        // we start from bottom and return at top

        dp[len] = 0;

        // Exactly same logic as Native DFS

        for(int index = len - 1; index >= 0; index--) {

            // Note: rides is already sorted

            // Find lower boundary, the first index 'i' satisfy

            // "rides[i][0] >= rides[index][1] (= val)"

            int i = binarySearch(rides, rides[index][1], index + 1);

            // 1. dp[index + 1] means not take current ride

            // 2. rides[index][1] - rides[index][0] + rides[index][2] + dp[i] means take current ride

            dp[index] = Math.max(dp[index + 1], rides[index][1] - rides[index][0] + rides[index][2] + dp[i]);

        }

        return dp[0];

    }

    // Find lower boundary

    private int binarySearch(int[][] rides, int val, int lo) {

        int hi = rides.length - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(rides[mid][0] >= val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

}

Time Complexity: O(mlogm), m is the number of ride intervals

Sorting the Rides: The initial step in the code is to sort the rides list, which consists of m ride intervals. This has a time complexity of O(m log m), where m is the number of rides.

Dynamic Programming with Memoization (dfs function): The dfs function is used here to implement dynamic programming with memoization. The memoization ensures that each possible starting point of a ride is computed at most once. Since memoization caches the results, the maximum number of distinct states to consider is m, the length of the sorted rides list.

Binary Search (bisect\_left function): Inside the dfs function, binary search is used to find the next non-overlapping ride, which has a time complexity of O(log m) per call.

Combining the dynamic programming computation with the binary search, for each of the m calls to dfs, a binary search operation is involved. Thus, each call may contribute up to O(log m) complexity. Given that there are m such calls, the total time complexity from the dynamic programming along with the binary searches is O(m log m).

Considering both sorting and the memoized dfs, the overall time complexity of the algorithm is O(m log m).

Space Complexity: O(m), m is the number of ride intervals

Sorting: Sorting is done in-place in Python (Timsort), but it may still require O(log m) space for the stack frames used in the sort implementation.

Memoization: The dfs function uses a memoization table (implicitly through the @cache decorator), which will store results for each unique argument (i). This could result in up to m entries in the worst case, giving a space complexity of O(m).

Recursion Stack: The depth of recursion may go up to m in the worst case if all rides are taken sequentially with no overlap, which contributes to O(m) space complexity.

Thus, combining the space requirements for the sorting, memoization, and recursion stack, the total space complexity is O(m).

In conclusion, the time complexity of the code is O(m log m) and the space complexity is O(m), where m is the number of rides.

**Refer to**

<https://leetcode.com/problems/maximum-earnings-from-taxi/solutions/1613010/c-brute-recursion-better-dp-optimal-dp-binary-search-clear-and-concise/>

**1. Brute Force (Recursion)**

In brute force we use simple recursion technique and opt between two choices:

Opt for an ride and add value to the answer

Skip the ride and move ahead

class Solution {

public:

long long recur(vector<vector<int>>&rides,int nn,int idx)

{

if(idx>=nn)

return 0;

int i;

for(i=idx+1;i<nn;i++)

{

if(rides[i][0]>=rides[idx][1])

break;

}

long long op1=recur(rides,nn,idx+1);

long long op2=rides[idx][1]-rides[idx][0]+rides[idx][2]+recur(rides,nn,i);

return max(op1,op2);

}

long long maxTaxiEarnings(int n, vector<vector<int>>& rides) {

sort(rides.begin(),rides.end());

int nn=rides.size();

return recur(rides,nn,0);

}

};

**2. Better Approach (Recursion + Memoization)**

class Solution {

public:

long long dp[(int)1e5];

long long recur(vector<vector<int>>&rides,int nn,int idx)

{

if(idx>=nn)

return 0;

int i;

if(dp[idx]!=-1)

return dp[idx];

for(i=idx+1;i<nn;i++)

{

if(rides[i][0]>=rides[idx][1])

break;

}

long long op1=recur(rides,nn,idx+1);

long long op2=rides[idx][1]-rides[idx][0]+rides[idx][2]+recur(rides,nn,i);

return dp[idx]=max(op1,op2);

}

long long maxTaxiEarnings(int n, vector<vector<int>>& rides) {

sort(rides.begin(),rides.end());

int nn=rides.size();

memset(dp,-1,sizeof dp);

return recur(rides,nn,0);

}

};

**3. Optimal Approach (Recursion + Memoization + Binary Search)**

class Solution {

public:

long long dp[(int)1e5];

int find(vector<vector<int>>&events,int start,int toFind)

{

int low=start;

int high=events.size()-1;

int ans=-1;

while(low<=high)

{

int mid=(low+high)/2;

if(events[mid][0]>=toFind)

{

ans=mid;

high=mid-1;

}

else

{

low=mid+1;

}

}

return ans;

}

long long recur(vector<vector<int>>&rides,int nn,int idx)

{

if(idx>=nn)

return 0;

if(idx<0)

return 0;

if(dp[idx]!=-1)

return dp[idx];

/\* O(n) time in worst case \*/

// for(i=idx+1;i<nn;i++)

// {

// if(rides[i][0]>=rides[idx][1])

// break;

// }

/\* O(logn) time in worst case Binary Search \*/

int i=find(rides,idx+1,rides[idx][1]);

long long op1=recur(rides,nn,idx+1);

long long op2=rides[idx][1]-rides[idx][0]+rides[idx][2]+recur(rides,nn,i);

return dp[idx]=max(op1,op2);

}

long long maxTaxiEarnings(int n, vector<vector<int>>& rides) {

sort(rides.begin(),rides.end());

int nn=rides.size();

memset(dp,-1,sizeof dp);

return recur(rides,nn,0);

}

};

**Refer to**

<https://algo.monster/liteproblems/2008>

**Problem Description**

In this problem, we are given an array called rides, which represents passengers looking for taxi rides along a road. The taxi can only move in one direction, from point 1 to point n. Each passenger request is described by three numbers: [start\_i, end\_i, tip\_i], where start\_i is the point where the passenger wants to start the ride, end\_i is the point where the passenger wants to end the ride, and tip\_i is the tip the passenger will give for the ride.

The revenue earned from a single passenger is equal to the distance from the start to the end of the ride plus the tip given by the passenger. The goal is to pick up the passengers in such a way as to maximize earnings. It’s important to note that a taxi driver can have only one passenger at a time in the taxi but can pick up another passenger at the same point where the last passenger was dropped off.

The question is to find out the maximum revenue that can be earned by the taxi driver by fulfilling these ride requests optimally.

**Intuition**

To approach this problem, an intuitive way to start thinking is through dynamic programming or recursive solutions, where we decide for each passenger whether to take them or not. What makes it challenging is that we must ensure that the choices we make are optimal at every stage.

One way to break down the problem is to sort the ride requests by their start points. This way, we can perform our decision-making process sequentially. At each step, we have two options: either pick up the current passenger or skip to the next possible ride that can be taken. If we choose to take a passenger, we must find the next ride that starts at or after our current ride ends, then calculate the earnings from the current ride plus the maximum earnings from all possible subsequent rides.

The key to implementing this solution efficiently is to use a technique called memoization, which stores the results of expensive function calls and returns the cached result when the same inputs occur again. Additionally, to avoid manually searching for the next possible ride to take after the current one ends, we can use an algorithm called binary search, which allows us to quickly find the position where the next ride starts.

By combining these techniques, we recursively explore the decision at each ride, either picking the passenger and using binary search to find our next decision point or skipping to the next passenger, and we use memoization to remember our calculated earnings to avoid redundant calculations and speed up the process. This leads to an optimized solution that can handle the given problem efficiently.

**Solution Approach**

The solution uses recursion and binary search along with memoization. Here's a step-by-step explanation of the implementation:

**Sorting the Rides**: First, we sort the rides array based on the starting point of each ride. This ensures that we make decisions in the sequence of the road.

**Recursive Function** (**dfs)**: We define a recursive function dfs(i) that operates on the index i of the rides array. The purpose of this function is to calculate the maximum earnings starting from the i-th ride.

**Base Case of Recursion:** The base case occurs when i is greater than or equal to the length of the rides list, indicating that there are no more rides to consider, and thus the earnings would be 0.

**Exploring Choices**: For each ride at index i, we consider two possibilities:

Skip the i-th ride and move to the next one: We simply make a recursive call to dfs(i + 1).

Take the i-th ride: To find the next compatible ride (which starts after the current ride ends), we use a binary search to find the index j of the next ride whose start is at least as large as the end of the current ride. We then make a recursive call to dfs(j) to get the maximum earnings from rides starting at j or later. The earning of taking the i-th ride is the sum of end\_i - start\_i + tip\_i and the earnings from the rides starting from j.

**Making the Optimal Choice**: At each step, we must decide which choice leads to greater earnings—either skipping or taking the ride. We use the max function to determine the greater of the two earnings from the choices above.

**Memoization with @cache**: The decorator @cache from the functools module is used on the dfs function to memoize results. This means that once we have computed the maximum earnings for a given starting index of rides, we do not have to compute it again; we can reuse the saved result. This significantly reduces the number of calculations and therefore the overall runtime of the algorithm.

**Binary Search:** bisect\_left from the bisect module is used to perform a binary search to find the next ride to consider. This search is O(log n) and is much faster than a linear search for large lists.

**Final Call and Answer**: After defining the recursive function with all of its logic and memoization, we make the initial call to dfs(0). Because the rides array is sorted by starting point, this will consider all possible combinations starting from the first ride and return the maximum earnings possible.

Here is the key function extracted from the solution code for quick reference:

@cache

def dfs(i):

if i >= len(rides):

return 0

s, e, t = rides[i]

j = bisect\_left(rides, e, lo=i + 1, key=lambda x: x[0])

return max(dfs(i + 1), dfs(j) + e - s + t)

This function is the cornerstone of our dynamic programming approach, utilizing memorization and recursion to solve the problem efficiently.

**Example Walkthrough**

Let's consider a small example to illustrate the solution approach. Suppose we have the following rides array:

rides = [[2, 5, 4], [3, 7, 2], [5, 9, 1], [6, 8, 6]]

Here are the steps we’ll take following the solution approach:

**Sorting the Rides**: We sort the array rides:

rides.sort() # [[2, 5, 4], [3, 7, 2], [5, 9, 1], [6, 8, 6]]

**Recursive Function (dfs)**: Define the dfs function:

@cache

def dfs(i):

if i >= len(rides):

return 0

s, e, t = rides[i]

j = bisect\_left(rides, e, lo=i + 1, key=lambda x: x[0])

return max(dfs(i + 1), dfs(j) + e - s + t)

Base Case of Recursion: When i is out of the bounds of rides, 0 is returned.

Exploring Choices: For each ride at index i, we look at two choices, skip or take the ride.

Making the Optimal Choice: Use the max function to decide the best choice, which results in maximum earnings.

Memoization with**@cache**: This ensures we do not repeat expensive calculations.

Binary Search: The bisect\_left function finds the next possible ride index efficiently.

Final Call and Answer: We start the process by calling dfs(0).

Now, let's walk through the dfs function with this rides array:

We start with dfs(0). Taking the first ride [2, 5, 4], we would earn 5 - 2 + 4 = 7. We need to find the next compatible ride that starts at or after 5. Using binary search, we find that the next ride we can take is at index 2, which is [5, 9, 1].

We explore dfs(2) because it's the next compatible ride. If we take this ride, we'll earn 9 - 5 + 1 = 5. Now we look for the next ride after 9. No more rides can be taken, so dfs(3) will return 0. The total earning from taking rides at index 0 and 2 will be 7 + 5 = 12.

We now explore dfs(1) which refers to ride [3, 7, 2]. The earnings would be 7 - 3 + 2 = 6. The binary search tells us the next ride starts at index 3. If we take this ride [6, 8, 6], we earn 8 - 6 + 6 = 8. There are no more rides after this, so the total earning by taking rides at index 1 and 3 will be 6 + 8 = 14.

Since we want to maximize our earnings, we compare the total earnings from different sequences of rides. The maximum earning is 14 which comes from choosing rides at indices 1 and 3.

Therefore, the maximum revenue the taxi driver can earn is $14 by following the optimal path of picking up passengers from the sorted positions [3, 7, 2] and [6, 8, 6]. The implementation ensures that we are only making necessary calculations, thus optimizing our solution.

import java.util.Arrays;

class Solution {

private int[][] rides; // Array to hold the information about the rides

private long[] memo; // Memoization array to store the maximum earnings until the ith ride

private int numRides; // Total number of rides

// Method to find the maximum taxi earnings

public long maxTaxiEarnings(int n, int[][] rides) {

numRides = rides.length; // Getting the number of rides

memo = new long[numRides]; // Initializing the memoization array

// Sorting rides based on their start time

Arrays.sort(rides, (a, b) -> a[0] - b[0]);

this.rides = rides; // Assigning the argument rides to class variable for use in other methods

return findMaxEarnings(0); // Starting the search for max earnings from the first ride

}

// Helper method using Depth First Search (DFS) to find the max earnings starting from ride i

private long findMaxEarnings(int i) {

if (i >= numRides) { // Base case: when no more rides are left to consider

return 0;

}

if (memo[i] != 0) { // If we have already computed this subproblem, return the stored value

return memo[i];

}

// Extract the details of the current ride

int start = rides[i][0], end = rides[i][1], tip = rides[i][2];

// Find the next ride index that can be taken after finishing the current one

int nextRideIndex = findNextRideIndex(end, i + 1);

// Recursive calls to find the maximum of either taking the current ride

// and the best of what follows or skipping to the next ride

long earningsWithCurrentRide = end - start + tip + findMaxEarnings(nextRideIndex);

long earningsSkippingCurrentRide = findMaxEarnings(i + 1);

long maxEarnings = Math.max(earningsWithCurrentRide, earningsSkippingCurrentRide);

memo[i] = maxEarnings; // Storing the result in memoization array

return maxEarnings; // Returning the max earnings from current ride i onwards

}

// Helper method to find the next index of the ride which starts after the end time of the current ride

private int findNextRideIndex(int end, int startIndex) {

int left = startIndex, right = numRides;

// Binary search to find the first ride that starts after the end of the current ride

while (left < right) {

int mid = (left + right) >> 1; // Equivalent to (left + right) / 2

if (rides[mid][0] >= end) {

right = mid;

} else {

left = mid + 1;

}

}

return left; // Returning the index of the next ride

}

}

**Time and Space Complexity**

The given Python code represents a solution to a problem related to taxi earnings, where an input list rides consists of rides information, and each ride includes a start time, an end time, and a tip. The solution computes the maximum earnings possible by following a specific strategy using dynamic programming and binary search.

Let's analyze the time complexity and space complexity of the code:

**Time Complexity:**

Sorting the Rides: The initial step in the code is to sort the rides list, which consists of m ride intervals. This has a time complexity of O(m log m), where m is the number of rides.

Dynamic Programming with Memoization (dfs function): The dfs function is used here to implement dynamic programming with memoization. The memoization ensures that each possible starting point of a ride is computed at most once. Since memoization caches the results, the maximum number of distinct states to consider is m, the length of the sorted rides list.

Binary Search (bisect\_left function): Inside the dfs function, binary search is used to find the next non-overlapping ride, which has a time complexity of O(log m) per call.

Combining the dynamic programming computation with the binary search, for each of the m calls to dfs, a binary search operation is involved. Thus, each call may contribute up to O(log m) complexity. Given that there are m such calls, the total time complexity from the dynamic programming along with the binary searches is O(m log m).

Considering both sorting and the memoized dfs, the overall time complexity of the algorithm is O(m log m).

**Space Complexity:**

Sorting: Sorting is done in-place in Python (Timsort), but it may still require O(log m) space for the stack frames used in the sort implementation.

Memoization: The dfs function uses a memoization table (implicitly through the @cache decorator), which will store results for each unique argument (i). This could result in up to m entries in the worst case, giving a space complexity of O(m).

Recursion Stack: The depth of recursion may go up to m in the worst case if all rides are taken sequentially with no overlap, which contributes to O(m) space complexity.

Thus, combining the space requirements for the sorting, memoization, and recursion stack, the total space complexity is O(m).

In conclusion, the time complexity of the code is O(m log m) and the space complexity is O(m), where m is the number of rides.