1900. The Earliest and Latest Rounds Where Players Compete Memoization **Dynamic Programming Leetcode Link** Hard

Problem Description In this tournament, n players are lined up in a single row, and they are numbered from 1 to n. The players compete in rounds where

If there's an odd number of players in a round, the middle player automatically advances without competing. The objective of the problem is to determine, given two specific players (firstPlayer and secondPlayer), the earliest and latest

the ith player from the front of the row competes with the ith player from the end of the row. The winner goes on to the next round.

round in which these two players could potentially compete against each other. It is important to note that firstPlayer and secondPlayer are the strongest and will win against any other player. When other players compete, the outcome of who wins can be decided by us in order to find out the earliest and latest round these two specific players could compete.

while making decisions that would either hasten or delay their match.

The intuition behind finding the earliest and latest rounds firstPlayer and secondPlayer can compete is to simulate the tournament

Since we can control the outcome of matches between other players, we use this to our advantage to find extremes. For the earliest,

we try to eliminate players in such a way that our firstPlayer and secondPlayer meet as soon as possible. For the latest, we do the

Intuition

opposite, we pick winners such that firstPlayer and secondPlayer can avoid meeting until it's inevitable.

The simulation uses dynamic programming (DP) to keep track of the earliest and latest rounds given different conditions. The DP state is defined by the position of firstPlayer from the front, the position of secondPlayer from the end, and the total number of players k in the current round. Therefore, dp(1, r, k) keeps track of the earliest and latest rounds they can meet given those conditions.

The solution uses recursion with memoization (functools.lru_cache) to ensure that overlapping subproblems are calculated only

once, hence optimizing the overall run time of our DP approach. The base cases are clear: if 1 == r, it means the firstPlayer and secondPlayer are facing each other, and the round numbers for both earliest and latest would be 1. Alternatively, if l > r, we swap the values to keep positions consistent with our definition. The recursive step involves iterating through all possible positions the players could end up in after the round (i and j) and keeping track of the minimum and maximum rounds for all these scenarios. We also constrain our iterations based on the number of players

In the end, dp(firstPlayer, n - secondPlayer + 1, n) gives us the earliest and latest rounds firstPlayer and secondPlayer can meet.

Solution Approach

• Dynamic Programming (DP): The function dp(l: int, r: int, k: int) -> List[int] represents the DP state, where l is the position of firstPlayer from the front, r is the position of secondPlayer from the back, and k is the total number of players left. The function returns the earliest and latest rounds in which firstPlayer and secondPlayer can compete.

• Memoization: Implemented using the functools.lru_cache decorator, which caches the results of each unique call to dp based

• Recursive Cases: These are handled through the nested for loops where i and j loop through possible positions of firstPlayer

firstPlayer and secondPlayer.

Example Walkthrough

Matches: (1 vs. 4) and (2 vs. 3).

Remaining players for the next round: 1, 2.

Remaining players for next round: 1, 3.

• Competition: (1 vs. 2).

potentially face player 4.

player.

• Constraints on Positions: The check not l + r - k // $2 \le i + j \le (k + 1)$ // 2 enforces that players cannot jump over the middle. It ensures that the possible positions of the players adhere to the rules of the tournament.

during iteration. With each valid scenario, we update them to reflect the earliest and latest possible competition rounds for

• Updating Earliest and Latest Rounds: Temporary variables a and b hold the minimum and maximum round numbers respectively,

rounds are 1. When l > r, the positions are swapped for consistency. • Returning the Result: The initial call to the DP function dp(firstPlayer, n - secondPlayer + 1, n) provides the final answer,

• Base Cases: When l == r, it indicates a direct match between firstPlayer and secondPlayer, and both earliest and latest

Initially, the players are lined up as 1, 2, 3, 4. **Round 1**

Let's walk through a small example. Suppose there are n = 4 players: 1, 2, 3, 4. We want to find out the earliest and latest rounds in

which player 1 (firstPlayer) and player 4 (secondPlayer) could meet, assuming the strongest players always win against any other

player 3 will delay the match between player 1 and 4. **Earliest Scenario**

Since player 1 (firstPlayer) is the strongest, they win and this would be the earliest round (Round 2) that player 1 could

By selecting player 2 in the first scenario, we are setting up direct competition between 1 and 2 in the next round, whereas selecting

rounds.

Latest Scenario

- This walkthrough illustrates how dynamic programming with memoization, coupled with logical rules of the tournament, can be used to find the earliest and latest possible rounds for any two given players to compete.
- # Define the memoization (dynamic programming) function with caching @functools.lru_cache(None) 8 def dp(left_index: int, right_index: int, num_players: int) -> List[int]: 9 10 # left_index is the position of the first player from the start (front), # right_index is the position of the second player from the end, 11

num_players is the total number of players in the current round.

Enumerate all possible new positions for players after matching

for j in range(left_index - i + 1, right_index - i + 1):

latest_round = max(latest_round, new_latest + 1)

Base case: if both players are about to meet

Ensure that left_index is always before right_index

return dp(right_index, left_index, num_players)

Initialize earliest and latest rounds as infinity

if left_index == right_index:

if left_index > right_index:

earliest_round = math.inf

latest_round = -math.inf

and negative infinity, respectively

for i in range(1, left_index + 1):

return [earliest_round, latest_round]

return dp(firstPlayer, n - secondPlayer + 1, n)

Adjust indices based on player positions and invoke dp

private int[] dp(int leftIndex, int rightIndex, int numPlayers) {

return dp(rightIndex, leftIndex, numPlayers);

if (leftIndex > rightIndex) {

if (memo.containsKey(key)) {

return memo.get(key);

if (leftIndex == rightIndex) {

return new int[]{1, 1};

int earliestRound = INT_MAX;

for (int i = 1; i <= leftIndex; ++i) {</pre>

continue;

// Ignore invalid matchings.

int newEarliest = result[0];

int newLatest = result[1];

return {earliestRound, latestRound};

const cache: Map<string, RoundResults> = new Map();

return function(...args: any[]): RoundResults {

if (cache.has(key)) return cache.get(key)!;

if (leftIndex >= rightIndex) return [1, 1];

// Calculate the sum of the new positions.

return dp(firstPlayer, n - secondPlayer + 1, n);

// Iterate over all possible new positions for players after matching.

let [newEarliest, newLatest] = dp(i, j, Math.floor((numPlayers + 1) / 2));

// Global function to find the earliest and latest rounds where two players could meet.

function earliestAndLatest(n: number, firstPlayer: number, secondPlayer: number): RoundResults {

// Adjust indices based on player positions and invoke the memoized dynamic programming function.

for (let j = leftIndex - i + 1; j <= rightIndex - i; j++) {</pre>

// Ignore invalid matchings based on the constraints.

latestRound = Math.max(latestRound, newLatest + 1);

innermost statements will run 0(1 * r) times for a particular value of k.

functools.lru_cache(None) ensures that each state is computed only once.

each recursive step, and 1 and r contribute a factor of n^2 due to the nested loops.

earliestRound = Math.min(earliestRound, newEarliest + 1);

let earliestRound: number = Infinity;

let latestRound: number = -Infinity;

for (let i = 1; i <= leftIndex; i++) {</pre>

return [earliestRound, latestRound];

Time and Space Complexity

let sum = i + j;

const result = fn(...args);

cache.set(key, result);

return result;

return dp(firstPlayer, n - secondPlayer + 1, n);

// Enumerate all possible new positions for players after matching.

for (int $j = leftIndex - i + 1; j \ll rightIndex - i + 1; ++j) {$

// Recurse to find the new earliest and latest rounds.

std::vector<int> result = dp(i, j, (numPlayers + 1) / 2);

earliestRound = std::min(earliestRound, newEarliest + 1);

type RoundResults = [number, number]; // Tuple type for holding the earliest and latest rounds.

function memoize(fn: (...args: any[]) => RoundResults): (...args: any[]) => RoundResults {

const key = JSON.stringify(args); // Serialize arguments array to use as a cache key.

// Decorator-like function to simulate memoization in TypeScript (Python's functools.lru_cache equivalent).

latestRound = std::max(latestRound, newLatest + 1);

// and invoke the dp (dynamic programming) function with caching.

int latestRound = INT_MIN;

// Ensure that leftIndex is always before rightIndex for consistency

// Convert the state to a string to use as a key for memorization

String key = leftIndex + "," + rightIndex + "," + numPlayers;

// If both players are about to meet, this is the end case

return [1, 1]

Eventually, we found that the earliest round they could meet was Round 2 and the latest was Round 3.

- Python Solution from typing import List import functools import math class Solution:
- # Ignore invalid matchings 30 if not left_index + right_index - num_players // 2 <= i + j <= (num_players + 1) // 2:</pre> 31 32 continue 33 # Recurse to find the new earliest and latest rounds 34 new_earliest, new_latest = dp(i, j, (num_players + 1) // 2) earliest_round = min(earliest_round, new_earliest + 1) 35
- public class Solution { private Map<String, int[]> memo = new HashMap<String, int[]>(); public int[] earliestAndLatest(int n, int firstPlayer, int secondPlayer) { 6 return dp(firstPlayer, n - secondPlayer + 1, n); 8

```
27
           // Initialize earliest and latest rounds
28
           int earliestRound = Integer.MAX_VALUE;
           int latestRound = -1;
           // Iterate over all possible new positions for players after matching
           for (int i = 1; i <= leftIndex; ++i) {</pre>
                for (int j = leftIndex - i + 1; j \le rightIndex - i + 1; ++j) {
                   // Skip invalid matchings
                   if (!(leftIndex + rightIndex - numPlayers / 2 \le i + j \& i + j \le (numPlayers + 1) / 2)) {
                        continue;
                   // Recursive call to find the new earliest and latest rounds
                    int[] rounds = dp(i, j, (numPlayers + 1) / 2);
                    earliestRound = Math.min(earliestRound, rounds[0] + 1);
                    latestRound = Math.max(latestRound, rounds[1] + 1);
           // Memorize the calculated rounds
           memo.put(key, new int[]{earliestRound, latestRound});
           return memo.get(key);
 1 #include<vector>
2 #include<functional>
   #include<climits>
   class Solution {
   public:
       std::vector<int> earliestAndLatest(int n, int firstPlayer, int secondPlayer) {
           // Use a lambda function to replace the functools.lru_cache in Python.
           // The lambda function will serve as the memoization function with caching.
 9
           std::function<std::vector<int>(int, int, int)> dp;
           dp = [&](int leftIndex, int rightIndex, int numPlayers) -> std::vector<int> {
```

12 **}**; 13 14 // Global recursive function for dynamic programming with memoization. const dp: (...args: any[]) => RoundResults = memoize(function (leftIndex: number, rightIndex: number, numPlayers: number): RoundRes // Base case: if both players are about to meet or have already met.

tournament bracket. The state of the dynamic programming dp[i][j][k] maintains the earliest and latest rounds where the i-th player from the front can meet the j-th player from the back out of k total players. **Time Complexity** To analyze the time complexity, we should consider three parameters 1, r, and k corresponding to the first player's position, the second player's position from the end, and the total number of players, respectively. The maximum range of 1 and r is n since n is the total number of players. The range of k is also 0(n) because, in the worst case, k takes on half the previous value, leading to a logarithmic number of states in terms of k.

The nested loops in the code iterate and process each combination of 1 and r within their bounds, so in the worst case the

Therefore, the overall time complexity is $0(n^3 \log n)$, since the range of k contributes a logarithmic factor due to the halving in

Considering the recursive nature, dp(i, j, (k + 1) // 2) will be invoked multiple times, but memoization using

The given code is a dynamic programming solution that calculates the possible rounds in which two players would meet in a

if (!(leftIndex + rightIndex - Math.floor(numPlayers / 2) <= sum && sum <= Math.floor((numPlayers + 1) / 2))) continue;</pre>

Space Complexity The space complexity is determined by the number of items that can be stored in the cache (memoization) and the recursion stack depth.

The cache size corresponds to the number of distinct states that the dynamic programming algorithm must keep track of, which involves the ranges of 1, r, and k. As concluded before, 1 and r have a maximum range of n, and k can take on 0(log n) different

values. Hence, the space complexity of the cache is $0(n^2 \log n)$.

The recursion stack depth will be proportional to the number of recursive calls we can have in the worst case, which is 0(log n) since in each recursive call k is approximately halved until it reaches the base case. In conclusion, the overall space complexity of the solution is $0(n^2 \log n)$ when considering the cache size as the dominant factor.

The solution is implemented in Python using a recursive function with memoization. The algorithm utilizes dynamic programming (DP) to store the outcome of previously computed scenarios, thus avoiding redundant calculations. Here is how each part of the

solution contributes to the overall approach:

on the parameter values to eliminate redundant calculations.

left (k).

and secondPlayer after each round. The if-statement ensures that only feasible matchups are considered. The recursive call dp(i, j, (k + 1) // 2) computes the result for the next round considering the current scenario.

- which takes into account the total number of players n and the positions of firstPlayer and secondPlayer. The combination of these approaches yields an optimized algorithm that computes the desired range of rounds efficiently through recursive exploration of the tournament bracket while maintaining the logical constraints of the problem.
- Possible Outcomes: • First match: Player 1 wins (since player 1 is firstPlayer), player 4 wins (since player 4 is secondPlayer). Second match: We can decide the outcome since these are not the players we are tracking. Let's assume player 2 wins for

the earliest possible match and player 3 wins for the latest possible match.

Competition: Since there's an odd number of players, player 1 advances without competing.

for the number of players counting from the back, which results in dp(1, 4 - 4 + 1, 4).

• The DP and memoization help us avoid recalculating the same scenario twice.

This would be the latest round (Round 3) that player 1 could potentially face player 4.

In the following round, the remaining players would be 1 and 4, and they would compete then.

Application of the Solution Approach In this example, we applied the solution approach as follows:

• We invoked the dp function to calculate the earliest and latest rounds for players 1 (firstPlayer) and 3 (secondPlayer), adjusted

• The base cases ensure that if firstPlayer (1) and secondPlayer (r) meet (1 == r), we return 1 for both earliest and latest

• The recursive calls and updates to a and b are used to determine the rounds while adhering to the rules of the tournament.

def earliestAndLatest(self, n: int, firstPlayer: int, secondPlayer: int) -> List[int]:

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

36

37

38

39

40

41

42

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

50

8

9

10

11

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

44

46

45 }

});

49 };

};

Typescript Solution

Java Solution

import java.util.*;

```
29
 30
 31
 32
 33
 34
 35
 36
 37
 38
 39
 40
 41
 42
 43
 44
 45
 46
 47
 48
 49
 50 }
 51
C++ Solution
 10
 11
                 // If both players are about to meet, return [1, 1] because both earliest and latest are at this round.
 12
                 if (leftIndex == rightIndex) {
 13
                     return {1, 1};
 14
 15
 16
                 // Ensure that leftIndex is always before rightIndex.
 17
                 if (leftIndex > rightIndex) {
 18
 19
                     return dp(rightIndex, leftIndex, numPlayers);
 20
 21
 22
                 // Initialize earliest and latest rounds.
```

if (!(leftIndex + rightIndex - numPlayers / $2 \le i + j \& i + j \le (numPlayers + 1) / 2)) {$

// Adjust indices based on player positions (firstPlayer stays the same, secondPlayer is adjusted from the end)