544. Output Contest Matches

Simulation

String

Recursion

In the given LeetCode problem, we are asked to simulate the matchups in an NBA-style playoff where stronger teams are paired

Problem Description

and n being the weakest.

Medium

The goal of the problem is to pair the strongest team with the weakest, the second strongest with the second weakest, and so forth, repeatedly until there is only one match left. This should be reflected in the final string in a nested format that visualizes the structure of the contests, with the matchups of each round encapsulated by parentheses and separated by commas.

with weaker teams in the order of their rankings. The teams are given an initial ranking from 1 to n with 1 being the strongest

For example, with 4 teams, the pairing process is as follows: 1. Round one pairings would be: (1,4) and (2,3).

2. Round two (the final) pairing would be: ((1,4),(2,3)).

This final string is what is expected as the output of the function.

Intuition

The intuition behind the solution approach is to use a simulation method, where we continually halve the number of teams by creating the pairs according to the problem statement in each iteration. It's clear we need to do this log2(n) times since each

so forth. These new pairs are stored at the corresponding beginning positions in the list.

manipulations, representing each team by its string equivalent simplifies the process.

time we pair up teams, we halve the total.

A key insight here is that after pairing, the number of teams in question is halved, so we end the process after each round by bitwise right shifting the number n by 1 which is equivalent to dividing n by 2. This is performed until we are left with just one pair, representing the final match. The iterative pairing and overwriting of the team list result in a tree-like structure in the string that delineates the progression of the matches.

The solution begins by initializing a list of teams from 1 to n, as strings because we need to return the output as a string. In each

iteration, we simulate the round by combining the first team in the list with the last, the second team with the second-to-last, and

the more error-prone string concatenation operations. Solution Approach

The f-string feature in Python is used to construct the new pairings efficiently, combining strings in a readable way that avoids

to create the desired nested pairing representation. Here is a breakdown of how the implementation works: Initialize the Team Representation: The implementation begins by creating a list named team, which contains the string

representation of team numbers from 1 to n. Since our final output needs to be a string and the operations are string

The solution to simulate NBA-style playoffs takes advantage of simple array manipulation and the concept of iterative reduction

n >>= 1

return team[0]

team = [str(i + 1) for i in range(n)]

weakest team (at the end of the team list) and so on.

denote the pairs, and commas separate the paired teams.

Simulating the Rounds: The solution uses a while loop to iteratively create matchups and reduce the number of teams until only one match is left (the final match). **Pairing Teams:** Inside the loop, a for loop ranges from 0 to n >> 1 (half of the current number of teams). This loop

represents the pairing process for the current round. The strongest team (at the start of the team list) is paired with the

The pairing syntax $f'({team[i]},{team[n - 1 - i]})'$ is used to ensure the required format is respected; parentheses

is executed by right-shifting n by 1 (n >>= 1). This operation makes sure that in the following iteration, a new round begins

for i in range(n >> 1): team[i] = $f'(\{team[i]\}, \{team[n - 1 - i]\})'$ Preparation for the Next Round: After each round of pairing, the number of teams still in the contest is halved. This reduction

and pairs together half as many teams, since each pair from the previous round is now treated as a single unit.

Completing the Matches: This process continues until there is only one match remaining, which implies that n becomes 1. At this point, we've built the full match tree in the team[0] element, representing the nested structure of all matchups until the final match.

and making use of Python's powerful string formatting features. **Example Walkthrough**

Initialize the Team Representation: We begin by initializing our team list to the string representations of numbers 1 through 8:

First Round Pairings: In the first iteration, we pair the first team with the last team, the second with the second-to-last, and

The given approach uses an iterative process, halving the dataset in each loop cycle, and array manipulation to build a nested

string that represents the playoff match pairings. It is an efficient implementation, limiting the operations to what is necessary

Simulate the Rounds: Our task is to simulate the rounds of matchups until only the final matchup remains.

team = ['(1,8)', '(2,7)', '(3,6)', '(4,5)', '5', '6', '7', '8']

team = ['1', '2', '3', '4', '5', '6', '7', '8']

team = ['1', '2', '3', '4', '5', '6', '7', '8']

Before pairing for the first round

Before pairing for the second round

Before pairing for the final round

After pairing for the final round

After pairing for the first round

The calculated final match representation is then returned:

Prepare for Next Round: We halve the number of teams by right-shifting n by 1:

round:

n >>= 1 # n is now 4

n >>= 1 # n is now 2

the nested string:

return team[0]

while n > 1:

n //= 2

return teams[0]

Java

C++

public:

class Solution {

class Solution {

for i in range(n // 2):

public String findContestMatch(int n) {

String[] teams = new String[n];

for (int i = 0; i < n; i++) {

for (; n > 1; n /= 2) {

string findContestMatch(int n) {

vector<string> teams(n);

for (int i = 0; i < n; ++i) {

teams[i] = to_string(i + 1);

// Create an array to store team names/strings.

// After each round the number of teams is halved.

// Method to find the contest match pairings given `n` teams

// Loop until we have only one match left (the final match)

// Loop for pairing teams up for the current round

// Continue pairing teams until one match is left (the final match)

// Each iteration halves 'n' since teams are paired into matches

// Ensures the team with the highest seed is matched with the lowest seed

// At this point, teams[0] contains the string representation of the final match

Create a list of team names as strings, starting from "1" up to "n".

Continue pairing teams together until we have only one match left.

them, but the number of string elements in the list decreases by half each time.

teams[i] = $f'(\{teams[i]\},\{teams[n-1-i]\})'$

Pair teams for the current round. The number of pairs is half the remaining teams.

After pairing, halve the number of teams to represent advancing to the next round.

At the end, teams[0] contains the final match as a string in the desired format.

Construct the match pairing for team i and the corresponding team from the end of the list.

Pair the first team with the last team, the second team with the second-to-last, and so on.

// Pair each team 'i' with its opposite seed in the list

teams[i] = "(" + teams[i] + "," + teams[n - 1 - i] + ")";

// Initialize a vector to store the team pairings

teams[i] = String.valueOf(i + 1);

for (int i = 0; i < n / 2; i++) {

Second Round Pairings: We now perform the next iteration of pairings with the team list reflecting the results of the first

Let's illustrate the solution approach with a small example where n = 8 representing 8 teams.

so on until we have 4 pairs. Using the for loop, we update the team array with these new pairs:

team = ['(1,8)', '(2,7)', '(3,6)', '(4,5)', '5', '6', '7', '8']# After pairing for the second round team = ['((1,8),(4,5))', '((2,7),(3,6))', '(3,6)', '(4,5)', '5', '6', '7', '8']

Note that we only need to update the first half of the team array. The second half will be ignored in later iterations.

team = ['(((1,8),(4,5)),((2,7),(3,6)))', '((2,7),(3,6))', '(3,6)', '(4,5)', '5', '6', '7', '8']Completion: With n now equal to 1, we have our final matchup set in team[0]. The final representation of all the matchups is

and string manipulation, yielding the correct nested representation of matchups.

Continue pairing teams together until we have only one match left.

teams[i] = $f'(\{teams[i]\}, \{teams[n - 1 - i]\})'$

Pair teams for the current round. The number of pairs is half the remaining teams.

After pairing, halve the number of teams to represent advancing to the next round.

At the end, teams[0] contains the final match as a string in the desired format.

// Initialize the array with team names represented as strings from "1" to "n".

teams[i] = "(" + teams[i] + "," + teams[n - 1 - i] + ")";

// Assign each team a string representation of their initial seeding number

for (; n > 1; n >>= 1) { // we half 'n' each time since teams are paired up

// Pair team 'i' with its corresponding team in this round,

// 'n - 1 - i' ensures that the team with the highest seed remaining

for (int i = 0; i < n / 2; ++i) { // only need to iterate through the first half of teams

// Repeatedly pair teams and update the array until we are left with only one match.

// Form pairs in this round and store them as strings in the form "(team1,team2)".

Which evaluates to (((1,8),(4,5)),((2,7),(3,6))).

team = ['((1,8),(4,5))', '((2,7),(3,6))', '(3,6)', '(4,5)', '5', '6', '7', '8']

Third Round Pairing (Final Round): Finally, we perform the last pairing with the updated team list:

Prepare for Next Round Again: Halve the number of teams once more:

```
Solution Implementation
Python
class Solution:
    def find contest match(self, n: int) -> str:
       # Create a list of team names as strings, starting from "1" up to "n".
       teams = [str(i + 1) for i in range(n)]
```

Construct the match pairing for team i and the corresponding team from the end of the list.

Pair the first team with the last team, the second team with the second-to-last, and so on.

This walkthrough demonstrates how the given solution approach efficiently simulates an NBA-style playoff using iterative pairing

// The final match is the first element in the array. return teams[0];

```
// is matched with the team with the lowest seed
                teams[i] = "(" + teams[i] + "," + teams[n - 1 - i] + ")";
        // In the end, teams[0] will contain the string representation of the final match
        return teams[0];
};
TypeScript
// Function to convert an integer to a string in TypeScript
function intToString(num: number): string {
  return num.toString();
// Function to find the contest match pairings given `n` teams
function findContestMatch(n: number): string {
  // Initialize an array to store the team pairings as strings
  let teams: string[] = new Array(n);
  // Assign each team a string representation of their initial seeding number
  for (let i = 0; i < n; i++) {
    teams[i] = intToString(i + 1);
```

The given function generates the matches by pairing teams and reducing n by half in each iteration. Since this operation is performed until n becomes 1, the number of iterations needed is 0(log n) because we're continually halving the number of teams.

Time and Space Complexity

for (; n > 1; n >>= 1) {

return teams[0];

while n > 1:

n //= 2

return teams[0]

class Solution:

for (let i = 0; i < n / 2; i++) {

def find contest match(self. n: int) -> str:

for i in range(n // 2):

teams = [str(i + 1) for i in range(n)]

remaining list of teams, and updating the team list with the new pairings. This loop operation has a complexity of O(n/2) which simplifies to O(n) for each level of iteration.

Time Complexity

Combining these two observations, the total time complexity is $O(n) * O(\log n)$, which gives us $O(n \log n)$. **Space Complexity** The space complexity of the function is primarily dependent on the team list that's updated in-place. Initially, the list is of size n,

holding all team numbers as strings. As the algorithm progresses, the strings within the team list grow as we keep re-pairing

In each iteration, the loop runs n/2 times (n >> 1 is the same as dividing n by 2), pairing off the first and last team in the

The largest amount of space will be used right at the start, when there are n strings. However, because the strings grow in length, we have to consider the storage used by these strings, which will be the sum of the lengths of all rounds.

At each level of iteration, the total number of characters in all strings combined essentially doubles (each string from the previous iteration is included in a pair, with two additional characters for parentheses and one for the comma), but we cannot simply state the space complexity is O(n) based on the initial list size due to the increasing size of the strings themselves.

If we account for the maximum length of the string after all match pairings are complete, we will have a string that is <code>0(n)</code>

characters long, because this string represents the pairing of all n teams with parenthesis and commas added.

- Hence, the space complexity, accounting for the maximum length of the final string, is O(n). To summarize:
 - Time Complexity: 0(n log n) Space Complexity: 0(n)