1560. Most Visited Sector in a Circular Track

Simulation Easy <u>Array</u>

Problem Description

consisting of m rounds. Each round starts and ends at specified sectors, with the starting sector for each round given in an array rounds. Specifically, the ith round starts at sector rounds[i - 1] and ends at sector rounds[i]. The goal is to find out which sectors are the most visited throughout the marathon and return these sectors sorted in ascending order. It's important to notice that the sectors must be visited in the counter-clockwise direction and we wrap around the track each time we reach sector n.

In this problem, we have a circular track with n sectors, each labeled from 1 to n. A marathon is taking place on this track,

The solution to this problem relies on the observation that if the start sector of the marathon (rounds [0]) is less than or equal to

Intuition

round always moves from a lower to a higher sector number, covering all sectors in between. However, if the start sector is greater than the end sector, this indicates that the runners have passed the sector marked n and started a new lap, visiting the sectors from 1 to rounds [-1] again. Therefore, in this case, the most visited sectors will be from

the end sector (rounds [-1]), then all sectors in between are visited more frequently than any other sector. This is because each

sector 1 to rounds [-1] and from rounds [0] to sector n. The solution approach, therefore, is to check the relative positions of the start and end sectors and return the appropriate range

of sectors: 1. If rounds [0] is less than or equal to rounds [-1], return a list of sectors ranging from rounds [0] to rounds [-1]. 2. If rounds [0] is greater than rounds [-1], there are two ranges of sectors to consider. Combine the range from 1 to rounds [-1] with the range

- **Solution Approach**
- The solution for finding the most visited sectors on the circular track follows a straightforward approach based on conditional

Condition Check: First, we check if rounds [0], the starting sector of the first round, is less than or equal to rounds [-1], the

class Solution:

consider two ranges of sectors.

if rounds[0] <= rounds[-1]:</pre>

from rounds [0] to n into a single list and return it.

logic rather than elaborate algorithms or data structures.

Here's a step-by-step explanation of the implementation:

ending sector of the last round. o If this condition is True, it means that the runners have not crossed the sector marked n from the last round to the first round, so we only

○ We use the range function to generate a list starting from rounds [0] to rounds [-1], inclusive. This list represents the sectors that are covered in a direct path from the starting sector to the ending sector without any wrap-around.

This indicates that during the marathon, runners have wrapped around the track, passing by sector n and back to sector 1. We need to

def mostVisited(self, n: int, rounds: List[int]) -> List[int]:

return list(range(rounds[0], rounds[-1] + 1))

need to consider the sectors that lie between rounds [0] and rounds [-1].

Range Construction (No Wrap-Around): When rounds [0] <= rounds [-1]:

Range Construction (With Wrap-Around): When rounds [0] > rounds [-1]:

 These two ranges are combined to form a complete list of the most visited sectors. **List Concatenation**: To combine the two ranges when there is a wrap-around: • We utilize list concatenation to join the lists resulting from the two range calls into a single list, which is returned as the final result.

The code implementation uses only basic Python constructs such as conditional statements and list manipulations, thereby

• The second range is from rounds [0] to sector n, inclusive, which represents the sectors visited before the wrap-around occurs.

○ The first range is from sector 1 to rounds [-1], inclusive, which represents the sectors visited after the last wrap-around.

avoiding the need for complex algorithms or data structures. It leverages Python's range function to create lists directly

corresponding to the ranges of sectors that are most visited, as deduced from the initial condition checks.

else: return list(range(1, rounds[-1] + 1)) + list(range(rounds[0], n + 1)) The function mostVisited takes two parameters, n and rounds, which represent the number of sectors and the array of rounds,

respectively. Within the function, we apply the logic outlined above to determine the most visited sectors and return them as a

```
list.
  The beauty of this solution lies in its simplicity and efficiency. Since it avoids additional computation by directly identifying the
  range of most visited sectors, it executes in constant time, making it suitable for even large values of n and numbers of rounds.
Example Walkthrough
```

Let's walk through a small example to illustrate the solution approach using a circular track with $\frac{5}{5}$ sectors ($\frac{1}{5}$ and an array

of rounds [1, 3, 5, 2]. We apply the steps from the solution approach to determine the most visited sectors.

• rounds = [1, 3, 5, 2]We start by applying the first step in our solution approach.

• In this case, the condition is True because 1 is less than 2. Normally, this would imply that we only need to consider the sectors that lie between 1 and 2. However, since there is a wrap-around

• n = 5 (sectors are 1, 2, 3, 4, 5)

Given:

Range Construction (With Wrap-Around): Since rounds [0] is not greater than rounds [-1] but there is a wrap-around from the third to the fourth round (from 5 back to 2), we consider the entire sequence of sectors visited.

• The sectors visited are initially [1, 2, 3, 4, 5], and following the wrap-around, the sectors [1, 2] are visited again, leading to the

implied by the progression of sectors in the rounds from 5 back to 2, we need to consider the full marathon sequence.

 Since sectors 1 and 2 were visited in the last part of the marathon, they would be added to the most visited sectors list. Hence, after evaluating the rounds in order, we determine that the most visited sectors would be [1, 2, 3, 4, 5], but [1, 2]

List Concatenation: We combine the ranges from the first round and after the wrap-around:

have the highest visit count due to the wrap-around. We need to sort them to return the result as [1, 2].

The first range is from sector 1 to sector 5, inclusive.

sequence [1, 2, 3, 4, 5, 1, 2].

• The second range starts again at 1 after wrapping around and ends at 2.

def mostVisited(self, sector count: int, rounds: List[int]) -> List[int]:

or if the race direction was such that there was no wraparound

public List<Integer> mostVisited(int sectors, int[] rounds) {

List<Integer> mostVisitedSectors = new ArrayList<>();

int finalRoundIndex = rounds.length - 1;

// List to store the most visited sectors

// The final round (m is now finalRoundIndex for clarity)

The first sector visited in the first round

The last sector visited in the last round

Condition Check: We compare rounds [0] to rounds [-1] (we check whether 1 <= 2).

2, 3, 4, 5] since we are asked to return unique sectors visited. After sorting, the final answer remains [1, 2, 3, 4, 5], since all sectors are most visited due to the marathon sequence that was provided in the rounds array.

Using the given solution code, the mostVisited function would thus return [1, 2, 3, 4, 5] + [1, 2], which simplifies to [1,

if start sector <= end sector:</pre> # The most visited sectors are all sectors from start_sector to end_sector, inclusive return list(range(start_sector, end_sector + 1)) else: # If there was a wraparound, the most visited sectors are from the beginning (sector 1) # to the end sector, and from the start sector to the end of the track (sector count) return list(range(1, end_sector + 1)) + list(range(start_sector, sector_count + 1))

If the race started (start sector) and ended (end sector) on the same or a directly subsequent sector,

Note: The List type needs to be imported from the typing module, so we should add the following line

Make sure to import the `List` type from the `typing` module for type annotations if it is not already imported:

from typing import List Java

import java.util.ArrayList;

import java.util.List;

class Solution {

at the beginning of the file:

from typing import List

```python

Solution Implementation

start sector = rounds[0]

end\_sector = rounds[-1]

**Python** 

class Solution:

```
// If the first round is less than or equals to the last round, it means the path doesn't cross sector 1
 if (rounds[0] <= rounds[finalRoundIndex]) {</pre>
 // Iterate from start sector to end sector and add to the list
 for (int i = rounds[0]; i <= rounds[finalRoundIndex]; ++i) {</pre>
 mostVisitedSectors.add(i);
 } else { // The path crosses sector 1
 // Add sectors from 1 to the final round's sector
 for (int i = 1; i <= rounds[finalRoundIndex]; ++i) {</pre>
 mostVisitedSectors.add(i);
 // Add sectors from the starting round's sector to the last sector (wrapping around)
 for (int i = rounds[0]; i <= sectors; ++i) {</pre>
 mostVisitedSectors.add(i);
 // Return the list of most visited sectors
 return mostVisitedSectors;
C++
#include <vector>
class Solution {
public:
 // This function returns the most visited sectors in a circular track after all rounds are completed.
 std::vector<int> mostVisited(int totalSectors, std::vector<int>& rounds) {
 int lastRoundIndex = rounds.size() - 1; // Get the index of the last round
 std::vector<int> mostVisitedSectors; // Vector to store the most visited sectors
 // Check if the starting sector number is less than or equal to the ending sector number
 if (rounds[0] <= rounds[lastRoundIndex]) {</pre>
```

// If it is, add all sectors from the start sector to the end sector to the answer

// Also add all sectors from the start sector to the total number of sectors

// Check if the starting sector number is less than or equal to the ending sector number

// The path has lapped around the track, so cover both segments of the lap

// Add all sectors from the start sector to the end sector to the most visited sectors

for (int i = rounds[0]; i <= rounds[lastRoundIndex]; ++i) {</pre>

// Return the list of the most visited sectors after all the rounds

// If it's not, the path has lapped around the track

for (int i = 1; i <= rounds[lastRoundIndex]; ++i) {</pre>

for (int i = rounds[0]; i <= totalSectors; ++i) {</pre>

mostVisitedSectors.push\_back(i);

mostVisitedSectors.push\_back(i);

mostVisitedSectors.push\_back(i);

// Define the function to calculate the most visited sectors.

// Initialize an array to store the most visited sectors

// Take the total number of sectors and an array of rounds as inputs.

function mostVisited(totalSectors: number, rounds: number[]): number[] {

for (let i = rounds[0]; i <= rounds[lastRoundIndex]; i++) {</pre>

return mostVisitedSectors;

// Get the index of the last round

const lastRoundIndex = rounds.length - 1;

if (rounds[0] <= rounds[lastRoundIndex]) {</pre>

mostVisitedSectors.push(i);

let mostVisitedSectors: number[] = [];

// Add all sectors from 1 to the end sector

} else {

**}**;

**TypeScript** 

} else {

```
// Add all sectors from 1 to the end sector number
 for (let i = 1; i <= rounds[lastRoundIndex]; i++) {</pre>
 mostVisitedSectors.push(i);
 // Add all sectors from the start sector to the total number of sectors
 for (let i = rounds[0]; i <= totalSectors; i++) {</pre>
 mostVisitedSectors.push(i);
 // Return the array of the most visited sectors
 return mostVisitedSectors;
class Solution:
 def mostVisited(self, sector count: int, rounds: List[int]) -> List[int]:
 # The first sector visited in the first round
 start sector = rounds[0]
 # The last sector visited in the last round
 end_sector = rounds[-1]
 # If the race started (start sector) and ended (end sector) on the same or a directly subsequent sector,
 # or if the race direction was such that there was no wraparound
 if start sector <= end sector:</pre>
 # The most visited sectors are all sectors from start_sector to end_sector, inclusive
 return list(range(start_sector, end_sector + 1))
 else:
 # If there was a wraparound, the most visited sectors are from the beginning (sector 1)
 # to the end sector, and from the start sector to the end of the track (sector count)
 return list(range(1, end_sector + 1)) + list(range(start_sector, sector_count + 1))
Note: The List type needs to be imported from the typing module, so we should add the following line
at the beginning of the file:
from typing import List
```

# Time and Space Complexity

from typing import List

```python

The time complexity of the code is primarily determined by the creation of the list that gets returned. This involves generating a range of integers, which can be done in constant time for each integer in the range, and then converting that range into a list.

In the best-case scenario, where rounds[0] <= rounds[-1], we create a single range from rounds[0] to rounds[-1], which

Make sure to import the `List` type from the `typing` module for type annotations if it is not already imported:

- In the worst-case scenario, where rounds[0] > rounds[-1], we create two ranges. The first is from 1 to rounds[-1] and the second is from rounds [0] to n. In the worst case, these two ranges can also include up to n elements combined, so the time
- complexity remains O(n). The space complexity of the code is determined by the space needed to store the output list.

• In both cases mentioned above, the space complexity depends on the number of elements in the output list, which can be at most n. Thus, the

space complexity for the algorithm is O(n). Therefore, the final time complexity is O(n) and the space complexity is O(n).

includes at most n elements. Thus, the time complexity for this case is O(n).