website pagination hackerrank solution java

Website Pagination HackerRank Solution Java: A Comprehensive Guide

Are you wrestling with the HackerRank "Website Pagination" challenge using Java? Feeling lost in a sea of loops and conditional statements? You're not alone! This comprehensive guide will walk you through a clear, concise, and efficient Java solution for the Website Pagination problem on HackerRank, equipping you with not only a working code but also a deeper understanding of the underlying pagination logic. We'll cover various approaches, optimize for performance, and provide detailed explanations to help you ace this coding challenge.

Understanding the HackerRank Website Pagination Problem

Before diving into the code, let's clearly define the problem. The HackerRank Website Pagination challenge typically presents you with a scenario where you have a large dataset (e.g., a list of website pages) that needs to be displayed across multiple pages. The challenge requires you to implement a function that takes the total number of pages, the page number the user wants to view, and the number of pages to display per page (page size) as input. Your function should then return the range of pages to display on that specific page.

For instance, if you have 100 pages, the user wants to view page 3, and you display 10 pages per page, your function should return the range [11, 20] because that's the set of pages shown on the third page.

Naive Approach: Direct Calculation

One straightforward (though not always the most efficient) approach is to directly calculate the start and end indices using simple arithmetic. Here's a basic Java function illustrating this:

```java
class Pagination {

static int[] getPaginationRange(int totalPages, int pageNumber, int pageSize) {
int startIndex = (pageNumber - 1) pageSize + 1;
int endIndex = Math.min(startIndex + pageSize - 1, totalPages);
return new int[]{startIndex, endIndex};
}
```

```
public static void main(String[] args) {
int[] result = getPaginationRange(100, 3, 10);
System.out.println("[" + result[0] + ", " + result[1] + "]"); // Output: [11, 20]

result = getPaginationRange(5,2,3);
System.out.println("[" + result[0] + ", " + result[1] + "]"); //Output: [4,5]
}
}
```

This code directly calculates the `startIndex` and `endIndex` based on the provided inputs. The `Math.min` function ensures that the `endIndex` doesn't exceed the `totalPages`. This approach is easy to understand, but it might not be the most optimal for extremely large datasets.

A More Robust and Efficient Approach: Handling Edge Cases

While the naive approach works for many cases, it's crucial to consider edge cases. What happens if the `pageNumber` is less than 1 or greater than the total number of pages? What if the `pageSize` is 0 or negative? A more robust solution should handle these scenarios gracefully.

```java
class PaginationRobust {

static int[] getPaginationRange(int totalPages, int pageNumber, int pageSize) {
//Error Handling for invalid inputs
if(totalPages <=0 || pageNumber <=0 || pageSize <=0){
return new int[]{-1,-1}; //Indicates an error
}

int startIndex = (pageNumber - 1) pageSize + 1;
int endIndex = Math.min(startIndex + pageSize - 1, totalPages);

//Additional check to handle edge case where pageNumber exceeds totalPages.
if(startIndex > totalPages){
return new int[]{-1,-1};
}
return new int[]{startIndex, endIndex};
}

public static void main(String[] args) {
int[] result = getPaginationRange(100, 3, 10);
System.out.println("[" + result[0] + ", " + result[1] + "]"); // Output: [11, 20]

result = getPaginationRange(5, 2, 3);
System.out.println("[" + result[0] + ", " + result[1] + "]"); // Output: [4, 5]

result = getPaginationRange(100, 15,10);
```

```
System.out.println("[" + result[0] + ", " + result[1] + "]"); //Output:[141,150]

result = getPaginationRange(100, 0, 10); //Error Handling
System.out.println("[" + result[0] + ", " + result[1] + "]"); //Output:[-1,-1]
}
}
```

This improved version adds input validation to prevent errors and unexpected behavior. Returning `[-1,-1]` in case of invalid inputs provides a clear signal of failure.

Optimization for Extremely Large Datasets

For incredibly large datasets, even the robust approach might have performance implications. However, for the scale of problems typically presented in HackerRank challenges, the above solutions are perfectly adequate. The focus here should be on correctness and readability rather than premature optimization.

Testing Your Solution

Thorough testing is crucial. Test your code with various inputs, including edge cases (e.g., `pageNumber` = 1, `pageNumber` = `totalPages`, `pageSize` = 1, `pageSize` = `totalPages`). Use a testing framework like JUnit to automate this process and ensure your solution is robust and accurate.

Conclusion

Solving the HackerRank Website Pagination problem in Java involves understanding the core pagination logic and implementing a function that correctly calculates the page range. While a straightforward approach using direct calculation is sufficient for many scenarios, a more robust solution should handle edge cases and invalid inputs gracefully. Remember to test your solution thoroughly to ensure its correctness and reliability. This guide provides you with both a basic and a more robust solution, helping you tackle this challenge effectively.

FAQs

1. What happens if `pageNumber` is greater than the total number of pages? The robust solution presented above will return `[-1, -1]` indicating an error. It's crucial to handle this edge case to avoid unexpected behavior.

1. Can I use other data structures (e.g., ArrayList) to solve this problem? While you can use other data structures, it's not strictly necessary for this particular problem. The solution primarily involves arithmetic calculations, and using a more complex data

structure would add unnecessary overhead.

1. How can I improve the performance of my solution for extremely large datasets? For extremely large datasets, consider exploring more advanced techniques, but for HackerRank challenges, the provided solutions are sufficient. The focus is on clarity and correctness, not premature optimization.

1. What is the time and space complexity of the provided solutions? Both solutions have a time complexity of O(1) (constant time) as the calculations are performed in constant time regardless of the input size. The space complexity is also O(1) (constant space) as it uses a fixed amount of memory.

1. Where can I find more HackerRank challenges to practice? HackerRank's website provides a vast library of coding challenges categorized by difficulty and topic. Regular practice is key to improving your coding skills and problem-solving abilities.

Additional Resources

website pagination hackerrank solution java

Website Pagination Hackerrank Solution Java

Website Pagination Hackerrank Solution Java

Related Articles

- who owns the future by jaron lanier
- whos afraid of solarin a play in honour of dr tai solarin
- visible body anatomy and physiology

Back to Home