278. First Bad Version

Easy Binary Search Interactive

Problem Description

You are a product manager overseeing the development of a new product, which unfortunately has some versions failing the quality check. Versions of this product are built sequentially where each version builds upon the previous one. If any version is discovered to be faulty, all subsequent versions will inherit this fault.

Your task is to efficiently determine the first version where the defect was introduced, given that subsequent versions will be defective as well. You have `n` versions labeled from `1` to `n`. To assist in identifying the defective version, you have access to a provided API function `isBadVersion(version)` that will return `true` if the version is bad, and `false` otherwise.

The goal is to minimize the number of API calls you make to `isBadVersion`.

Intuition

To solve this problem, a binary search algorithm is an ideal approach. Binary search is applied because we know the versions are ordered from good to bad and the first bad version triggers all the following versions to be bad as well. This creates a sorted pattern of good and bad versions, which is perfect for binary search.

Initially, we know the right boundary (`right`) of our search space is `n` (the last version) and the left boundary (`left`) is `1` (the first version). To minimize the calls to the API, we avoid searching versions sequentially. Instead, we calculate the middle point (`mid`) of the current search space and use the `isBadVersion` API to check if `mid` is a bad version.

If `mid` is bad, we know that the first bad version must be at `mid` or before `mid`, so we set `right` to `mid`. If not, we know that the first bad version must be after `mid`, so we update `left` to be `mid + 1`. This halves the search space with each iteration.

Repeatedly narrowing the search space like this leads us to converge on the first bad version without checking every version. The loop terminates when `left` and `right` meet, which is when we have found the first bad version. By utilizing this method, the number of calls we make to the API is reduced to $O(\log n)$, which is the time complexity of a binary search.

Solution Approach

The solution implements a classic binary search strategy to find the first bad version. This method significantly reduces the time complexity from $O(n)$ to $O(\log n)$. Let's walk through the implementation of the solution:

1. We start by initializing two pointers, `left` and `right`. `left` starts at 1, which represents the first version, and `right` starts at `n`, which represents the last version. They mark the boundaries of our search space.
2. We enter a while loop that will continue running as long as `left` is less than `right`. Inside this loop, we will repeatedly narrow down our search space by inspecting the midpoint of `left` and `right`.
3. To calculate the midpoint, we use the expression `(left + right) >> 1`, which is equivalent to `(left + right) / 2` but faster computationally as it involves bit-shifting to the right by one bit to do integer division by two. We assign this value to the variable `mid`.
4. The `isBadVersion(mid)` call tells us whether the midpoint version is bad. If it is, we know the target bad version must be less than or equal to `mid`. So, we set `right` to `mid`, indicating our search space has shifted to the left half from `left` to `mid`.
5. Conversely, if `isBadVersion(mid)` returns `false`, it means that the first bad version is somewhere after `mid`. Hence, we set `left` to `mid + 1`, indicating our search space has shifted to the right half excluding `mid`.
6. The loop continues, and with each iteration, the range `[left, right]` narrows down until `left` equals `right`. At this point, `left` (or equivalently `right`) points to the first bad version.

With the binary search pattern applied, we not only find the correct answer but also optimize performance by making the least number of calls necessary to `isBadVersion`, satisfying the problem's constraint to minimize API calls. Once the loop finishes, we return `left` as it will be the index of the first bad version.

Example Walkthrough

Suppose we have 5 versions of a product and we know that one of the middle versions introduced a defect. Our `isBadVersion` API functions as follows for these 5 versions:

- `isBadVersion(1)` returns `false`
- `isBadVersion(2)` returns `false`
- `isBadVersion(3)` returns `true`
- `isBadVersion(4)` returns `true`
- `isBadVersion(5)` returns `true`

Using the binary search approach, we start with `left = 1` and `right = 5`. We perform the following steps to identify the first bad version:

1. First, take the midpoint of `left` (1) and `right` (5). The midpoint `mid` is calculated as `(1 + 5) >> 1` which equates to 3.
2. We check `isBadVersion(3)` and it returns `true`. Because `mid` is bad, we know all versions after 3 are also bad. So, we set `right` to `mid`, changing our range from `[1, 5]` to `[1, 3]`.
3. Next, we recalculate the midpoint of the new range. With `left` still at 1 and `right` at 3, the new `mid` is `(1 + 3) >> 1`, which equals 2.
4. We call `isBadVersion(2)` and it returns `false`. Now we know the first bad version must be greater than `mid`. So we update `left` to `mid + 1`, changing our range from `[1, 3]` to `[3, 3]`.
5. Since `left` and `right` meet and the loop terminates, we conclude that the first bad version is 3.

By following this example, the approach narrows down the search space efficiently and we find the first bad version which is 3, with just two API calls to `isBadVersion` instead of five, which would have been the case if we had checked each version sequentially. This demonstrates the effectiveness of the binary search technique.

Solution Implementation

Python

```python
# Assume the isBadVersion API is already defined.
# @param version: int => An integer representing the version
# @return bool => True if the version is bad, False otherwise
# def isBadVersion(version):


class Solution:
    def firstBadVersion(self, n: int) -> int:
        """
        Utilize binary search to find the first bad version out of 'n' versions.
        If a bad version is found, then all subsequent versions are also bad.

        :type n: int The total number of versions to check.
        :rtype: int The first bad version number.
        """
        # Initialize the search space
        left, right = 1, n
        # Loop until the search space is reduced to one element
        while left < right:
            # Calculate the middle point to divide the search space
            mid = left + (right - left) // 2
            # If mid is a bad version, the first bad version is in the left half
            if isBadVersion(mid):
                right = mid  # Narrow the search to the left half
            else:
                left = mid + 1  # Narrow the search to the right half

        # At this point, left is the first bad version
        return left
```

Java

```java
public class Solution extends VersionControl {
    /**
     * Finds the first version that is bad.
     *
     * @param n Total number of versions.
     * @return The first bad version number.
     */
    public int firstBadVersion(int n) {
        // Initialize pointers for the range start and end.
        int start = 1;
        int end = n;

        // Continue searching while the range has more than one version.
        while (start < end) {
            // Calculate the middle version of the current range.
            // Using unsigned right shift operator to avoid integer overflow.
            int mid = start + (end - start) / 2;

            // Check if the middle version is bad.
            if (isBadVersion(mid)) {
                // If the middle version is bad, the first bad version
                // is before it or it is the first bad version itself.
                end = mid;
            } else {
                // If the middle version is good, the first bad version
                // must be after it.
                start = mid + 1;
            }
        }

        // At this point, start == end and it is the first bad version.
        return start;
    }
}
```

C++

```cpp
// The API isBadVersion is defined for you.
// bool isBadVersion(int version);

class Solution {
public:
    // Function to find the first bad version in a sequence of versions
    int firstBadVersion(int n) {
        int left = 1;           // Initialize the left boundary of the search range
        int right = n;          // Initialize the right boundary of the search range

        // Perform a binary search to narrow down the first bad version
        while (left < right) {
            // Use bitwise shift to avoid the potential overflow of (left + right)
            int mid = left + ((right - left) >> 1);

            // Check if the middle version is bad
            if (isBadVersion(mid)) {
                // If it is bad, we know that the first bad version must be at or before 'mid'
                right = mid;    // Reset the right boundary to 'mid'
            } else {
                // If it is not bad, the first bad version must be after 'mid'
                left = mid + 1; // Reset the left boundary to one after 'mid'
            }
        }

        // At the end of the loop, 'left' will be the first bad version
        return left;
    }
};
```

TypeScript

```typescript
/**
 * The `isBadVersion` function is a predicate that determines whether
 * a given software version is bad. The actual implementation is unspecified.
 *
 * @param version - The version number to check.
 * @returns `true` if the specified version is bad, otherwise `false`.
 */
type IsBadVersion = (version: number) => boolean;

/**
 * This function is a higher-order function that takes in the `isBadVersion` function
 * and returns a nested function that can be used to find the first bad version.
 *
 * @param isBadVersion - A function to determine whether a given version is bad.
 * @returns A function that accepts the total number of versions `n` and returns the
 * first bad version found.
 */
const solution = (isBadVersion: IsBadVersion) => (n: number) => {
  /**
   * This inner function uses binary search to efficiently find the first bad version
   * within the total number of versions `n`.
   *
   * @param n - The total number of versions to search through.
   * @returns The first bad version number within the total versions.
   */
  return (n: number): number => {
    let left: number = 1;
    let right: number = n;
    while (left < right) {
      // Use Math.floor to ensure the result is an integer after the division.
      const midpoint: number = left + Math.floor((right - left) / 2);
      if (isBadVersion(midpoint)) {
        right = midpoint;
      } else {
        left = midpoint + 1;
      }
    }
    // At this point, 'left' is the first bad version.
    return left;
  };
};


# Assume the isBadVersion API is already defined.
# @param version: int => An integer representing the version
# @return bool => True if the version is bad, False otherwise
# def isBadVersion(version):


class Solution:
    def firstBadVersion(self, n: int) -> int:
        """
        Utilize binary search to find the first bad version out of 'n' versions.
        If a bad version is found, then all subsequent versions are also bad.

        :type n: int The total number of versions to check.
        :rtype: int The first bad version number.
        """
        # Initialize the search space
        left, right = 1, n
        # Loop until the search space is reduced to one element
        while left < right:
            # Calculate the middle point to divide the search space
            mid = left + (right - left) // 2
            # If mid is a bad version, the first bad version is in the left half
            if isBadVersion(mid):
                right = mid  # Narrow the search to the left half
            else:
                left = mid + 1  # Narrow the search to the right half

        # At this point, left is the first bad version
        return left
```

Time and Space Complexity

The given code snippet uses a binary search algorithm to find the first bad version among `n` versions. The binary search successively divides the range of the search in half, narrowing down on the first bad version.

Time Complexity

The time complexity of this algorithm is $O(\log n)$. This is because with each comparison, it effectively halves the search space, which is a characteristic of binary search. As such, the number of comparisons needed to find the target is proportional to the logarithm of the total number of versions `n`.

Space Complexity

The space complexity of this algorithm is $O(1)$. This is because it uses a fixed amount of space; only two variables `left` and `right` are used to keep track of the search space, and the `mid` variable is used for the computations. No additional space is dependent on the input size `n`.