Solution: First Bad Version

Let's solve the First Bad Version problem using the Modified Binary Search pattern.

We'll cover the following

Statement

The latest version of a software product fails the quality check. Since each version is developed upon the previous one, all the versions created after a bad version are also considered bad.

Suppose you have `n` versions with the IDs $[1, 2, ..., n]$, and you have access to an API function that returns TRUE if the argument is the ID of a bad version.

Find the first bad version that is causing all the later ones to be bad. Additionally, the solution should also return the number of API calls made during the process and should minimize the number of API calls too.

Constraints:

- $1 \leq$ first bad version $\leq$ `n` $\leq 2^{31} - 1$

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach is to find the first bad version in the versions range by linear search. We traverse the whole version range one element at a time until we find the target version.

The time complexity of this approach is $O(n)$, because we may have to search the entire range in this process. This approach ignores an important piece of information: the range of version numbers is sorted from 1 to n. Let's see if we can use this information to design a faster solution.

Optimized approach using modified binary search

Since the versions range is sorted, binary search is the most efficient approach for finding the first bad version. It executes in $O(\log n)$ time, which is faster than the alternate linear scanning method.

Our goal is to find the first bad version by making the minimum number of calls to the API function. We use the binary search algorithm. The idea is to check the middle version to see if it's bad. If it's good, this means that the first bad version occurs later in the range, allowing us to entirely skip checking the first half of the range. If it's bad, we need to check the first half of the range to find the first bad version. Therefore, we can skip checking the second half of the range.

When we go to check the identified half of the range, we use the same approach again. We check the middle version to figure out which half of the current range to check.

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

Let's start with the simplest step. Since we know the number of versions and also the versions range is sorted, we use two pointers, `first` and `last`. `first` will point to the first version, which is 1, and `last` will point to the last version which is `n`. Additionally, we initialize a variable, `calls`, with zero that counts the number of API calls made by the solution to find the first bad version.

Java

```java
import java.util.*;

class FBVersion {
    public static int v;

    static int[] firstBadVersion(int n) {
        int[] result = new int[2];
        // Assigning first pointer with the first version that is 1
        int first = 1;
        // Assigning last pointer with n that is the number of versions
        int last = n;
        int calls = 0;
        result[0] = first;
        result[1] = calls;
        System.out.print("\n\tVersions:  ");
        for (int x = first; x < last; x++) {
            System.out.print(x + ", ");
        }
        System.out.println(last + "\n");
        System.out.println("\tfirst: " + first);
        System.out.println("\tlast: " + last);
        return result;
    }

    // Driver code
    public static void main(String args[]) {
        int[] testCaseVersions = new int[]{38, 13, 29, 40, 23};
        int[] firstBadVersion = new int[]{28, 10, 10, 28, 19};
```

First Bad Version

Next, we can find the middle version's ID. We do it in a `while` loop because we repeat this step until the `first` becomes equal to the `last`. We will pass the middle to the API function, which will tell us whether or not the middle version is bad.

Java

```java
import java.util.*;

class FBVersion {
```

```
class FBVersion {
    public static int v;

    static boolean isBadVersion(int s) {
        return s >= v;
    }

    static int[] firstBadVersion(int n) {
        int[] result = new int[2];
        // Assigning first pointer with the first version that is 1
        int first = 1;
        // Assigning last pointer with n that is the number of versions
        int last = n;
        int calls = 0;
        result[0] = first;
        result[1] = calls;
        System.out.print("\n\tVersions:  ");
        for (int x = first; x < last; x++) {
            System.out.print(x + ", ");
        }
        System.out.println(last + "\n");
        // Binary search to find the target version
        while (first < 2) // Now we will search the entire list
        {
            int mid = first + (last - first) / 2;
            first += 1;
```

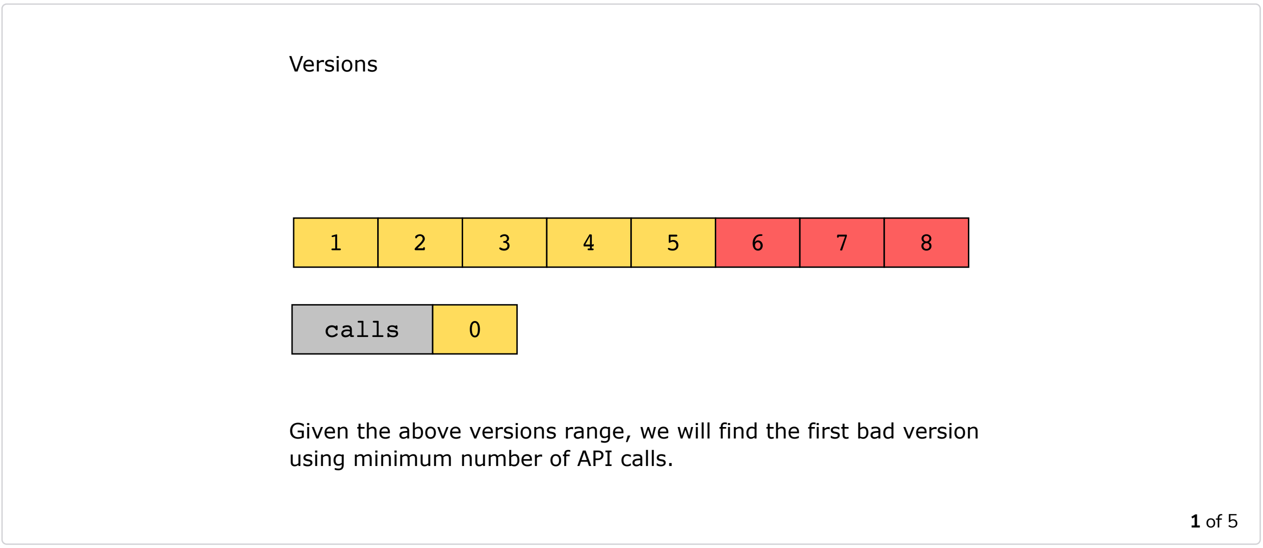
First Bad Version

Call the API fuction, `isBadVersion`, to check if whether or not that version is bad. Increment the `calls` variable on every API call so that at the end, we can return the number of API calls. In this way, each API call helps us divide our data set in half and narrow down the search space. Now, we just have to keep iterating and narrowing down until the version is found. Here's how we divide the search space:

- If the middle version is bad, we know that this is either the first bad version or that the first bad version occurred prior to this version. Therefore, we search for the first bad version from `1` to `mid`.
- If the middle version is not bad, we need to search for the first bad version from `mid + 1` to `n`.

After we've found the first bad version, we will return a tuple containing first bad version and the minimum number of API calls.

The slides below illustrate how we would like the algorithm to run:



Given the above versions range, we will find the first bad version using minimum number of API calls.

1 of 5

Java

```java
import java.util.*;

class FBVersion {
    public static int v;

    static boolean isBadVersion(int s) {
        return s >= v;
    }

    static int[] firstBadVersion(int n) {
        int[] result = new int[2];
        // Assigning first pointer with the first version that is 1
        int first = 1;
        // Assigning last pointer with n that is the number of versions
        int last = n;
        int calls = 0;
        System.out.print("\n\tVersions:  ");
        for (int x = first; x < last; x++) {
            System.out.print(x + ", ");
        }
        System.out.println(last + "\n");

        // Binary search to find the target version
        while (first < last) // Last should be equal to n. But here last = 2. Because we are calculating
        {
            int mid = first + (last - first) / 2;
            // isBadVersion() is the API function that returns true
            // or false depending upon whether the provided version ID
```

First Bad Version

Just the code

Here's the complete solution to this problem:

Java

```java
import java.util.*;

class FBVersion {
    public static int v;

    static boolean isBadVersion(int s) {
        return s >= v;
    }

    static int[] firstBadVersion(int n) {
        int[] result = new int[2];
        int first = 1;
        int last = n;
        int calls = 0;

        while (first < last) {
            int mid = first + (last - first) / 2;

            if (isBadVersion(mid)) {
                last = mid;
            } else {
                first = mid + 1;
            }

            calls += 1;
        }

        result[0] = first;
```

Solution summary

To recap, the solution to this problem can be divided into the following parts:

1. Assign two pointers at the IDs of the first and last versions.
2. Calculate the mid and check if that mid version is bad.
3. If the mid version is bad, search for the first bad version in the first half of the range of versions.
4. If the mid version is good, search in the second half of the range of versions.
5. Repeat the steps to check the mid version and to divide the range of versions in two until the first and last converge.

Time complexity

This algorithm takes $O(\log n)$ time, because we keep dividing the searching space in half at each iteration, where n is the number of versions.

Space complexity

The algorithm uses $O(1)$ space.

Back
First Bad Version

Next
Random Pick with We...

✓ Mark as Completed