



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

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.

To solve this problem, a binary search algorithm is an ideal approach. Binary search is applied because we know the versions are

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

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

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 $0(\log n)$, 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:

mid.

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
- 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

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.

mid + 1, indicating our search space has shifted to the right half excluding mid.

return left as it will be the index of the first bad version.

mid + 1, changing our range from [1, 3] to [3, 3].

demonstrates the effectiveness of the binary search technique.

1 # Assume the isBadVersion API is already defined.

def firstBadVersion(self, n: int) -> int:

Initialize the search space

while (start < end) {</pre>

// Calculate the middle version of the current range.

int mid = start + (end - start) / 2;

if (isBadVersion(mid)) {

// must be after it.

start = mid + 1;

end = mid;

} else {

// Check if the middle version is bad.

// Using unsigned right shift operator to avoid integer overflow.

// If the middle version is bad, the first bad version

// is before it or it is the first bad version itself.

// If the middle version is good, 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

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

• isBadVersion(1) returns false

functions as follows for these 5 versions:

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.

- 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
- 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
- 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

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

Python Solution

Utilize binary search to find the first bad version out of 'n' versions.

2 # @param version: int => An integer representing the version 3 # @return bool => True if the version is bad, False otherwise # def isBadVersion(version): class Solution:

If a bad version is found, then all subsequent versions are also bad. 11 12 13 :type n: int The total number of versions to check. :rtype: int The first bad version number. 14

9

10

15

16

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

```
17
           left, right = 1, n
18
           # Loop until the search space is reduced to one element
           while left < right:</pre>
19
               # Calculate the middle point to divide the search space
20
               mid = left + (right - left) // 2
21
22
               # If mid is a bad version, the first bad version is in the left half
23
               if isBadVersion(mid):
24
                   right = mid # Narrow the search to the left half
25
               else:
26
                   left = mid + 1 # Narrow the search to the right half
27
28
           # At this point, left is the first bad version
29
           return left
30
Java Solution
   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.
10
           int start = 1;
           int end = n;
11
13
           // Continue searching while the range has more than one version.
```

30

```
// At this point, start == end and it is the first bad version.
31
32
           return start;
33
34 }
35
C++ Solution
1 // The API isBadVersion is defined for you.
2 // 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
10
           // Perform a binary search to narrow down the first bad version
11
12
           while (left < right) {</pre>
13
               // Use bitwise shift to avoid the potential overflow of (left + right)
               int mid = left + ((right - left) >> 1);
14
15
               // Check if the middle version is bad
16
               if (isBadVersion(mid)) {
17
                   // 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'
19
20
               } else {
21
                   // If it is not bad, the first bad version must be after 'mid'
22
                   left = mid + 1; // Reset the left boundary to one after 'mid'
23
24
25
26
           // At the end of the loop, 'left' will be the first bad version
27
           return left;
28
29 };
30
```

Typescript Solution

```
/**
     * 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;
9
10
   /**
     * 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.
13
14
     * @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.
16
17
   const solution = (isBadVersion: IsBadVersion): ((n: number) => number) => {
     /**
20
      * This inner function uses binary search to efficiently find the first bad version
      * within the total number of versions `n`.
23
      * @param n - The total number of versions to search through.
      * @returns The first bad version number within the total versions.
24
25
26
      return (n: number): number => {
27
       let left: number = 1;
28
       let right: number = n;
       while (left < right) {</pre>
29
30
         // Use Math.floor to ensure the result is an integer after the division.
31
         const midpoint: number = left + Math.floor((right - left) / 2);
32
         if (isBadVersion(midpoint)) {
33
           right = midpoint;
         } else {
34
35
            left = midpoint + 1;
36
37
38
       // At this point, 'left' is the first bad version.
       return left;
39
40
     };
41 };
42
```

The given code snippet uses a binary search algorithm to find the first bad version among n versions. The binary search successively

The time complexity of this algorithm is $0(\log n)$. 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

Time Complexity

Time and Space Complexity divides the range of the search in half, narrowing down on the first bad version.

Space Complexity

the input size n.

logarithm of the total number of versions n.

The space complexity of this algorithm is 0(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