330. Patching Array

Hard Greedy Array

Leetcode Link

Problem Description

The problem is focused on a concept called `patching array`. Given a sorted array of integer numbers `nums` and an integer `n`, the task is to modify this array (by potentially adding new numbers to it), so that every number in the set {1, 2, ..., n} can be expressed as the sum of some elements in the array. You need to find the minimum number of new numbers (patches) that you have to add to the array to achieve this.

In simple terms, the question asks: "What is the smallest amount of numbers we can add to `nums` so that we can create every integer from 1 up to `n` using sums of numbers in `nums`?"

Intuition

The intuition behind this solution comes from the idea of exploring "ranges" of numbers that can be created using the current array elements and the numbers we add (patch). The strategy is to start with the smallest possible range `[1, 1]` and then expand this range as far as possible until we can reach `n`.

If the next number in the array (`nums[i]`) is within the current range we can reach (`<= x`), it helps us extend the range further without needing to patch the array (since every number up to `x` can be formed, adding `nums[i]` lets us now form every number up to `x + nums[i]`). Each time this happens, we take the next number in `nums` and adjust our range accordingly.

However, when the next number in `nums` is beyond the current reach (`nums[i] > x`), we can't use it to expand our range and must add (patch) a new number. The most efficient number to add is the next number just after the current reach (`x` itself), which maximally extends the range by doubling it (since before the patch we could reach `x - 1`, after adding `x`, we can reach `2x - 1`).

The algorithm keeps track of the patches added and the current range that can be formed. When the range expands to include `n` or more, we've added enough patches and can stop the process, returning the number of patches as our answer.

Solution Approach

The algorithm makes use of a greedy approach, as it always tries to extend the range as much as possible with each step by either using the next available number in `nums` or by patching the smallest number possible to increase the coverage range.

Here is the step-by-step explanation of the implementation:

1. Initialize two pointers: `x` starts at one, representing the current smallest number that we want to make sure can be formed by `nums`. `i` keeps track of the current position in the `nums` array, starting at zero.
2. Initialize `ans` to zero, which will keep track of the number of patches needed.
3. Start a `while` loop that continues as long as `x` is less than or equal to `n`. This is because we need to ensure that we can form all the numbers in the range `[1, n]`.
4. Inside the loop, we check if we have not yet used all numbers in `nums` and whether the current number at `nums[i]` can be used to extend the range. If `nums[i]` is less than or equal to `x`, then it means we can use `nums[i]` to create new sums up to `x + nums[i]`. Thus, we update `x` to `x + nums[i]` and increment `i`, moving to the next number in `nums`.
5. If, however, the current number in `nums` is larger than `x`, it cannot be used to extend our range. At this point, we must add a patch. The patch will be `x`, doubling the current range we can reach (`x <<= 1` is equivalent to `x = x * 2`). We also increment the `ans` since we've used a patch.
6. This process continues, either extending the range with the current numbers in `nums` or adding patches until `x` exceeds `n`.
7. When the loop finishes, `ans` holds the minimum number of patches added to achieve the goal, and we return `ans`.

It is important to note that this algorithm runs efficiently because:

- We only iterate through `nums` once.
- We use a while loop that runs in O(log n) time in the worst case scenario (when patches are added until `x` exceeds `n`).
- We do not use any additional data structures, which keeps our space complexity at O(1).

With this approach, we then can return the minimum amount of patches needed to `nums` to ensure that all numbers within the range `[1, n]` can be formed.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have an array `nums = [1, 3]` and we want to ensure we can form every number from 1 to `n = 7` using sums of numbers from `nums`. We want to find the minimum number of patches needed.

1. Initialize `x` to 1 (since we need to start forming sums from 1) and `i` to 0 (the index of the first number in `nums`). `ans` is also initialized to 0 as no patches have been added.
2. Start the `while` loop since `x = 1 <= n = 7`.
3. Inside the loop, check if `i` is within bounds of `nums` and if `nums[i] <= x`. For the first iteration, `nums[0] = 1` is equal to `x = 1`. Therefore, we can use `nums[0]` to form numbers up to `1 + 1 = 2`. Update `x` to 2 and increment `i` to 1.
4. Next, with `x = 2` and `nums[i] = 3`, we can see that `nums[1]` can be used to extend the range. We update `x` to `2 + 3 = 5` and increment `i` to 2 (which is now beyond the bounds of `nums`).
5. With `x = 5` and `i` out of bounds, we need to patch. The optimal patch is `x` itself, so we add 5 (the patch) to `nums`. The range extends to `5 + 5 = 10`, which is beyond `n = 7`, so we update `x = 10`. Increase `ans` by 1, since we added a patch.
6. The `while` loop ends because `x = 10` is now greater than `n = 7`. We can now form all the numbers from 1 to 7 using `[1, 3, 5]`.
7. The `ans` has counted a single patch (5). Therefore, we return `ans = 1`.

Through this process, we find that we only need to add one patch (the number 5) to the array `[1, 3]` to be able to form every number between 1 and 7 inclusively.

Python Solution

```python
from typing import List

class Solution:
    def minPatches(self, nums: List[int], n: int) -> int:
        # Initialize the smallest number that cannot be formed
        # by summing elements in nums up to index i
        missing_number = 1

        # Initialize the count of numbers we need to patch (add) to nums
        patches_count = 0

        # Initialize index for iterating through nums
        index = 0

        # While the smallest missing number is not greater than n
        while missing_number <= n:
            # If current index is within bounds and nums[index] can be used to form missing_number
            if index < len(nums) and nums[index] <= missing_number:
                # Increase missing_number by nums[index],
                # as we can form numbers up to missing_number + nums[index]
                missing_number += nums[index]

                # Move to the next element in nums
                index += 1
            else:
                # Otherwise, no element in nums can form missing_number.
                # Therefore, we need to patch (add) missing_number itself
                patches_count += 1

                # When we add missing_number, we can now form numbers up to missing_number * 2 - 1
                # We use bit shifting as a fast way to multiply missing_number by 2
                missing_number <<= 1

        # Return the total count of numbers we needed to patch to nums
        return patches_count

```

Java Solution

```java
class Solution {
    public int minPatches(int[] nums, int n) {
        long coverage = 1; // This will keep track of the largest number that can be formed by the sum of subsets of sorted nums
        int patches = 0; // This will count the number of patches (numbers) we need to add
        int index = 0; // Index to iterate through the nums array

        // Loop until the coverage is less than or equal to n
        while (coverage <= n) {
            // If the current index is within the bounds of the nums array
            // and the current number at nums[index] can be used to extend coverage
            if (index < nums.length && nums[index] <= coverage) {
                // Increment coverage by nums[index] and move to the next number in nums
                coverage += nums[index++];
            } else {
                // Increment patches since we need to add a new number to extend coverage
                patches++;
                // Double the coverage, simulating the addition of the number equal to the current coverage
                coverage <<= 1;
            }
        }

        // Return the number of patches needed to ensure a complete range from 1 to n
        return patches;
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to calculate the minimum number of patches required
    int minPatches(vector<int>& nums, int n) {
        long long coverLimit = 1; // Start with a coverage limit of 1
        int patchesCount = 0; // Initialize the count of patches needed
        size_t idx = 0; // Initialize the current index for the nums vector

        // Loop until the coverage limit exceeds n
        while (coverLimit <= n) {
            // Check if the current index is within bounds and the current number is within the coverage limit
            if (idx < nums.size() && nums[idx] <= coverLimit) {
                // The current number can extend the coverage limit
                coverLimit += nums[idx++]; // Extend the coverage range and move to the next number
            } else {
                // The current coverage limit cannot be extended with the existing numbers
                ++patchesCount; // Increment the number of patches
                coverLimit <<= 1; // Double the coverage limit
            }
        }

        // Return the number of patches needed to cover the range [1, n]
        return patchesCount;
    }
};

```

Typescript Solution

```typescript
function minPatches(nums: number[], n: number): number {
    let currentMax = 1; // Initialize current maximum possible sum
    let patchesNeeded = 0; // Initialize count of patches needed
    let currentIndex = 0; // Initialize current index in the input array `nums`

    // Loop until the current maximum sum is less than or equal to `n`
    while (currentMax <= n) {
        // If we haven't reached the end of the input array `nums`
        // and the current number is less than or equal to the current max sum,
        // we can use this number to extend the range of representable sums.
        if (currentIndex < nums.length && nums[currentIndex] <= currentMax) {
            currentMax += nums[currentIndex++]; // Add the current number to the max sum and increment index
        } else {
            // If the condition above is not true, we need to add a patch (a new number)
            // That number is always the current maximum sum itself, which doubles the range of representable sums.
            patchesNeeded++; // Increment the patches needed
            currentMax *= 2; // Double the current max sum
        }
    }

    // Return the total number of patches needed
    return patchesNeeded;
}

```

Time and Space Complexity

Time Complexity:

The algorithm loops while `x` is less than or equal to `n`. Within each loop, the code either adds a number from `nums` to `x` (if the current element in `nums` is less than or equal to `x`), or it doubles `x` (if there is no such element or all elements have been used). The key point is that `x` grows exponentially (either by adding a number from `nums` that's less than or equal to `x` or by doubling through patches). Hence, the number of times the loop can iterate is logarithmic in relation to `n`. Specifically, it will iterate `O(log(n))` times because `x` can double at most `O(log(n))` times to exceed `n`.

In each iteration, the algorithm performs a constant amount of work unless it's adding a number from `nums`, in which case it works in `O(1)` time to add a number and move the `i` pointer. Since it moves through each element of `nums` at most once, the number of these operations is bounded by the length of `nums`. Therefore, the total time complexity is `O(len(nums) + log(n))`.

Space Complexity:

The extra space used by the algorithm is constant, as it only uses a few variables (`x`, `ans`, `i`) and does not allocate any additional space that grows with the size of the input. Thus, the space complexity is `O(1)`.