313. Super Ugly Number

Medium Array Math Dynamic Programming Leetcode Link

Problem Description

A **super ugly number** is defined as a positive integer where all of its prime factors are part of a given list of prime numbers. The problem asks for the `n`th super ugly number, given `n` and a list of these prime factors, which are in an array called `primes`.

More concretely, you need to generate super ugly numbers in ascending order and then return the number that is in the `n`th position in this sequence.

For example, if `primes = [2, 7, 13, 19]`, the sequence of super ugly numbers starts as `1, 2, 4, 7, 8, 13, 14, 16, 19, ...`, and so on. In this sequence, `1` is always included and is considered the first super ugly number.

An important constraint is that the `n`th super ugly number is guaranteed to fit in a 32-bit signed integer. This means the result should be less than or equal to `2^31 - 1`.

Intuition

The intuition for solving this problem relies on building the sequence of super ugly numbers one by one, efficiently. The method used to keep the sequence sorted and avoid duplicates is a priority queue, commonly implemented with a heap in many programming languages.

The key insight is that every new super ugly number can be formed by multiplying a previously found super ugly number with one of the prime factors from the given `primes` array. This process will ensure we're covering all possible combinations while maintaining the sequence sorted.

As we're interested only in the `n`th super ugly number, we can use a min-heap (a priority queue that pops the smallest element) to repeatedly extract the smallest super ugly number and generate its successors by multiplying with the prime factors. This way, we ensure that numbers are always processed in increasing order, relating back to the constraint that we're required to return the `n`th super ugly number.

To further optimize and avoid duplicate calculations, when we take the smallest number from the heap, only the first prime factor that produces a new super ugly number which hasn't been added to the heap before will generate successors for this number. This is made possible by checking if the current super ugly number is divisible by a prime factor. If it is, it means we have found it using this prime factor previously, thus we can safely stop checking later prime factors for this iteration.

Additionally, there's a check to ensure we do not exceed the 32-bit integer limit during our multiplication, which aligns with the constraint mentioned in the problem description.

Solution Approach

The solution uses a **priority queue** data structure for maintaining the order of super ugly numbers. In Python, a priority queue can be implemented using a **heap**, from the `heapq` module.

Here's a step-by-step walkthrough of the implementation process:

1. Initialize a priority queue `q` with the number 1. This represents the first super ugly number.
2. Then, for producing the `n`th super ugly number, we perform a loop that runs `n` times.
3. In each iteration of the loop:
   - We pop the smallest number `x` from the heap. This is the next super ugly number in sequence.
   - For each prime factor `k` in the `primes` array:
     - We multiply `x` by `k` and check if this product would exceed the maximum 32-bit signed integer value (which is `(1 << 31) - 1` in the code). If not, we push the product onto the heap.
     - If `x` is divisible by `k`, we stop looping through the prime factors for this `x`, as we have found the smallest prime factor that can generate `x`, and we want to avoid duplicates in the heap.
   - The check for the maximum value is to prevent integer overflows, and the condition `x % k == 0` is to ensure we do not insert the same super ugly number into the heap more than once.
4. After running the loop `n` times, the `n`th super ugly number `x` is the last number popped from the heap.

The usage of the heap is crucial here, as it allows us to efficiently get the smallest number, which is our next super ugly number in each iteration. Also, the heap manages possible duplicates, which would otherwise complicate the implementation if we used another data structure such as an array or list.

Overall, this approach is efficient because it only considers necessary multiples of super ugly numbers and prime factors, and due to the nature of heap operations, the time complexity for each insertion and extraction is `O(log n)`, making the overall algorithm significantly faster than a brute-force method that might consider all possible products.

Example Walkthrough

Let’s walk through a small example to illustrate the solution approach. Consider finding the 6th super ugly number with primes = [2, 3, 5].

Step-by-step process:

1. Initialize the priority queue `q` with 1 (the first super ugly number).
2. For finding the 6th super ugly number, we need to iterate 6 times.
3. Begin the loop:
   - First iteration:
     Pop the smallest number from the heap, which is 1. Multiply it by each of the primes.
     - Multiply by 2: Get 2, push onto the heap.
     - Multiply by 3: Get 3, push onto the heap.
     - Multiply by 5: Get 5, push onto the heap. Heap after first iteration: [2, 3, 5]
   - Second iteration:
     Pop the smallest number, which is 2.
     - Multiply by 2: Get 4, push onto the heap.
     - 2 is not divisible by the next prime 3, so we continue.
     - Multiply by 3: Get 6, push onto the heap.
     - Multiply by 5: Get 10, push onto the heap. Heap after second iteration: [3, 4, 5, 6, 10]
   - Third iteration: Pop the smallest number, which is 3.
     - Multiply by 2: Get 6, which is already in the heap, so we skip pushing it.
     - 3 is divisible by 3, we can stop here. Heap after third iteration: [4, 5, 6, 10]
   - Fourth iteration: Pop the smallest number, which is 4.
     - Multiply by 2: Get 8, push onto the heap.
     - 4 is not divisible by the next prime 3, so we continue.
     - Multiply by 3: Get 12, push onto the heap.
     - Multiply by 5: Get 20, push onto the heap. Heap after fourth iteration: [5, 6, 8, 10, 12, 20]
   - Fifth iteration: Pop the smallest number, which is 5.
     - Multiply by 2: Get 10, which is already in the heap, so we skip pushing it.
     - 5 is not divisible by the next prime 3, so we continue.
     - Multiply by 3: Get 15, push onto the heap.
     - Multiply by 5: Get 25, push onto the heap. Heap after fifth iteration: [6, 8, 10, 12, 15, 20, 25]
   - Sixth iteration: Pop the smallest number, which is 6.
     - Multiply by 2: Get 12, which is already in the heap, so we skip pushing it.
     - 6 is divisible by 3, so we can stop here.

At this point, we have popped 6 numbers, which means the last popped number, 6, is our answer. The 6th super ugly number is 6.

By following this heap-based approach and ensuring we only push new products onto the heap, we can efficiently find the desired super ugly number without duplicate calculations or unnecessary multiplications.

Python Solution

```python
import heapq  # Import the heapq module for heap operations

class Solution:
    def nthSuperUglyNumber(self, n: int, primes: List[int]) -> int:
        # Initialize a min-heap with the first super ugly number, which is always 1
        min_heap = [1]

        # Initialize a variable to store the current super ugly number
        current_ugly_number = 0

        # Define the maximum value based on 32-bit signed integer range
        max_value = (1 << 31) - 1

        # Iterate to find n super ugly numbers
        for _ in range(n):
            # Pop the smallest value from the min-heap
            current_ugly_number = heapq.heappop(min_heap)

            # Try to generate new super ugly numbers by multiplying the current smallest
            # with each of the provided prime factors
            for prime in primes:
                # Prevent integer overflow by ensuring the product does not exceed max_value
                if current_ugly_number <= max_value // prime:
                    heapq.heappush(min_heap, prime * current_ugly_number)

                # If the current ugly number is divisible by prime,
                # do not consider higher primes to avoid duplicates
                if current_ugly_number % prime == 0:
                    break

        # Return the nth super ugly number after the loop ends
        return current_ugly_number

```

Java Solution

```java
class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        // A min-heap to hold and automatically sort the ugly numbers
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        // Adding the first ugly number which is always 1
        minHeap.offer(1);
        // Variable to hold the current ugly number
        int currentUglyNumber = 0;

        // Generate the nth ugly number
        while (n-- > 0) {
            // Get the smallest ugly number from the min-heap
            currentUglyNumber = minHeap.poll();

            // Avoid duplicates by polling all instances of the current ugly number
            while (!minHeap.isEmpty() && minHeap.peek() == currentUglyNumber) {
                minHeap.poll();
            }

            // Multiply the current ugly number with each prime to get new ugly numbers
            for (int prime : primes) {
                // Check for overflow before adding the new ugly number
                if (prime <= Integer.MAX_VALUE / currentUglyNumber) {
                    minHeap.offer(prime * currentUglyNumber);
                }
                // If current ugly number is divisible by prime, no need to check further
                if (currentUglyNumber % prime == 0) {
                    break;
                }
            }
        }
        // Return the nth ugly number
        return currentUglyNumber;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>
#include <climits>

class Solution {
public:
    // Function to find the nth super ugly number
    int nthSuperUglyNumber(int n, std::vector<int>& primes) {
        // Priority queue to store intermediate super ugly numbers,
        // with smaller numbers having higher priority.
        std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;

        // Start with 1 as the first super ugly number.
        minHeap.push(1);

        int currentUglyNumber = 0; // Variable to store the current ugly number.

        // Iterate until we find the nth super ugly number.
        while (n--) {
            // Extract the smallest super ugly number from the heap.
            currentUglyNumber = minHeap.top();
            minHeap.pop();

            // Generate new super ugly numbers by multiplying with the given primes.
            for (int prime : primes) {
                // Prevent integer overflow by checking the multiplication condition.
                if (currentUglyNumber <= INT_MAX / prime) {
                    minHeap.push(prime * currentUglyNumber);
                }

                // If the current number is divisible by the prime, then break to
                // avoid duplicates (since all primes could evenly divide the number).
                if (currentUglyNumber % prime == 0) {
                    break;
                }
            }
        }
        // Return the nth super ugly number.
        return currentUglyNumber;
    }
};

```

Typescript Solution

```typescript
function nthSuperUglyNumber(n: number, primes: number[]): number {
    // Array to store intermediate super ugly numbers,
    // sorted so that smaller numbers come first.
    let uglyNumbers: number[] = [1];

    // Variable to store the index of the last number
    // extracted from the sorted array for each prime.
    let indices = new Array(primes.length).fill(0);

    // Variable to store the next multiples for each prime.
    let nextMultiples = [...primes];

    for (let count = 1; count < n; count++) {
        // Find the minimum super ugly number among the next multiples.
        let nextUglyNumber = Math.min(...nextMultiples);

        // Add the smallest number to the list of super ugly numbers.
        uglyNumbers.push(nextUglyNumber);

        // Update the multiples list and indices.
        for (let j = 0; j < primes.length; j++) {
            // If this prime's next multiple is the one we just added,
            // update indices and calculate its new multiple.
            if (nextMultiples[j] === nextUglyNumber) {
                indices[j]++;
                nextMultiples[j] = uglyNumbers[indices[j]] * primes[j];
            }
        }
    }

    // Return the nth super ugly number.
    return uglyNumbers[n - 1];
}

```

Time and Space Complexity

The provided code defines a function to find the `n`-th super ugly number where super ugly numbers are positive integers whose prime factors are in the given list of `primes`.

Time Complexity

The time complexity of the code is as follows:

- The `for` loop runs `n` times because we are looking for the `n`-th super ugly number.
- Inside the loop, the `heappop()` operation is performed once, which has a complexity of `O(log m)` (where `m` is the current size of the heap `q`).
- After popping the smallest element, we iterate over the list of `primes`, which has a complexity of `O(k)` per loop iteration, where `k` is the number of primes.
- In the worst case, we push one element to the heap per every prime in the list, resulting in `O(k log m)` since each `heappush()` operation has a complexity of `O(log m)`.
- The `if x % k == 0: break` will potentially reduce the number of pushes since it stops pushing once the prime divides `x` since `x` wouldn't be super ugly for some larger prime factors.
- The multiplication `k * x` is `O(1)` operation, and the comparison `x <= mx // k` is also `O(1)`.

Given that the heap potentially has all `n` elements and the number of primes is `k`, the overall time complexity is `O(n * k * log(n))`.

Space Complexity

The space complexity is as follows:

- The heap `q` can grow up to a size of `n`, holding at most all the super ugly numbers up to the `n`-th.
- No additional meaningful storage is used, so the space complexity is primarily from the heap.

Thus, the space complexity is `O(n)`.