264. Ugly Number II

Medium | Hash Table | Math | Dynamic Programming | Heap (Priority Queue)

Problem Description

An ugly number is defined as a positive integer that only has prime factors of 2, 3, and 5. In other words, any positive integer that is divisible by any prime factor other than 2, 3, or 5 is not an ugly number. The sequence of ugly numbers starts with 1 since it has no prime factors. The task is to find and return the `n`th ugly number in the sequence of all ugly numbers.

Intuition

The straightforward way to find the `n`th ugly number would be to loop through all the integers, factorize each, and check if their factors are limited to 2, 3, and 5. However, this method would be highly inefficient as it requires factorization of each number up to the nth ugly number.

An efficient approach exploits the fact that every new ugly number must be built by multiplying a smaller ugly number by 2, 3, or 5. The idea is to maintain a list of ugly numbers found so far. For generating the next ugly number, the algorithm selects the smallest number that can be obtained by multiplying any ugly number by 2, 3, or 5. This ensures that we are building upon the sequence of ugly numbers we have already generated and not missing any in between.

The given code uses Dynamic Programming to keep track of the ugly numbers generated so far. It initializes an array `dp` where `dp[i]` will represent the `i+1`th ugly number. It uses three pointers (`p2`, `p3`, `p5`) to track the position for the next multiplication with 2, 3, and 5 respectively.

To find the next ugly number in the sequence, the algorithm will do the following:

- Multiply the ugly numbers at the respective pointers `p2`, `p3`, and `p5` by 2, 3, and 5 to get `next2`, `next3`, and `next5`.
- The minimum value of these three is the next ugly number.
- Each pointer is incremented if and only if the minimum value is equal to the product of the respective ugly number and the prime number corresponding to that pointer. This step simultaneously allows multiple increments if there are ties (i.e., two or three of the products have the same minimum value), ensuring that the same ugly number is not generated more than once.

Continuing in this manner fills the `dp` array with the first `n` ugly numbers. The `nth` ugly number, or `dp[n-1]` since the array index is 0-based, is then returned.

Solution Approach

The solution uses dynamic programming as its main algorithmic pattern to build up a series of ugly numbers, ensuring that each new ugly number is the smallest possible one that follows the previous ugly numbers.

Let's look at the major steps of the implementation in detail:

1. **Initialization**: The solution starts by initializing a list `dp` containing `n` elements, all set to 1. The first element of the list, `dp[0]`, is set to 1 because 1 is the first ugly number by definition. It also sets up three pointers `p2`, `p3`, and `p5`, each starting at 0. These pointers will keep track of which ugly number should next be multiplied by 2, 3, and 5, respectively.

2. **Building the Ugly Numbers List**: The algorithm goes into a loop starting from `i = 1` (since 1 is already in the list) up to `n-1`. Within the loop:

   - It calculates `next2`, `next3`, and `next5` by multiplying `dp[p2]` by 2, `dp[p3]` by 3, and `dp[p5]` by 5, respectively.
   - It then finds the minimum of these three values. This minimum value is the next ugly number, which is then added to the list at `dp[i]`.
3. **Pointer Incrementation**: For each pointer (`p2`, `p3`, and `p5`), if the value they pointed to times its respective factor (2, 3, or 5) is equal to the current minimum (the new ugly number just calculated), that pointer is incremented. This means that pointer is now pointing at the next candidate that needs to be multiplied by its respective prime to compete for a place in the ugly number sequence.

4. **Returning the Result**: After filling up the list with `n` ugly numbers, the algorithm returns the `n`th ugly number, which is `dp[n - 1]`.

The use of dynamic programming in this problem helps to drastically reduce the time complexity. Instead of recalculating each possibility for each new number, the algorithm builds on previously calculated values. Pointers ensure that the sequence progresses in the correct order, and no ugly numbers are missed.

Here is a visualization of how the algorithm processes the sequence for `n = 10`:

```
Ugly Numbers [1]:      p2 --> 2,     p3 --> 3,    p5 --> 5
Add min to list [1, 2]: p2 --> 4,     p3 --> 3,    p5 --> 5
Add min to list [1, 2, 3]: p2 --> 4,     p3 --> 6,    p5 --> 5
...
```

As the algorithm finds each new ugly number, it updates the list and the pointers appropriately until it reaches the `n`th element.

Example Walkthrough

Let's walk through an example to illustrate the solution approach for finding the 7th ugly number using the efficient dynamic programming method described. We will start with `n = 7`.

1. **Initialization**: We begin by initializing our `dp` array with a size of `n`. Initially, `dp = [1, _, _, _, _, _, _]` where `1` is the first ugly number and _ represents uninitialized values. The pointers `p2`, `p3`, and `p5` are all set to 0, which points to `dp[0]`.

2. **Building the Ugly Numbers List**: We will iterate and calculate `next2`, `next3`, `next5`, pick the minimum, and then store it in `dp[i]`. So let's illustrate the iterations:

   - First Iteration (`i = 1`):

     - `next2 = 2 * dp[p2] = 2 * dp[0] = 2`
     - `next3 = 3 * dp[p3] = 3 * dp[0] = 3`
     - `next5 = 5 * dp[p5] = 5 * dp[0] = 5`
     - `min(next2, next3, next5) = 2`
     - `dp[1] = 2`
     - `p2` is incremented because `next2` was used.
   - Second Iteration (`i = 2`):

     - `next2 = 2 * dp[p2] = 2 * dp[1] = 4`
     - `next3 = 3 * dp[p3] = 3 * dp[0] = 3`
     - `next5 = 5 * dp[p5] = 5 * dp[0] = 5`
     - `min(next2, next3, next5) = 3`
     - `dp[2] = 3`
     - `p3` is incremented because `next3` was used.

   And here's the sequence of the array and pointer updates for each step until `dp[6]` is filled (as `dp` is 0-indexed, `dp[6]` will give the 7th ugly number):

   ```
   dp[1] = 2     p2 --> 1,  p3 --> 0,  p5 --> 0
   dp[2] = 3     p2 --> 1,  p3 --> 1,  p5 --> 0
   dp[3] = 4     p2 --> 2,  p3 --> 1,  p5 --> 0
   dp[4] = 5     p2 --> 2,  p3 --> 1,  p5 --> 1
   dp[5] = 6     p2 --> 3,  p3 --> 2,  p5 --> 1
   dp[6] = 8     p2 --> 4,  p3 --> 2,  p5 --> 1
   ```

3. **Pointer Incrementation**: At each iteration, the pointer for which the next ugly number was found (either `p2`, `p3`, or `p5`) is incremented. If there is a tie (i.e., two or three `next` values are the same), all tied pointers are incremented, ensuring unique entries and correct sequence progression.

4. **Returning the Result**: After looping until `i = 6` (which is when we have our 7th ugly number), the array looks like `[1, 2, 3, 4, 5, 6, 8]`. The loop stops and we have generated our `n` ugly numbers. The 7th ugly number is `dp[6]` which is `8`.

The algorithm efficiently calculates the sequence of ugly numbers, keeping track of the next potential candidates for ugly numbers using pointers and only iterates `n` times. Thus, for `n = 7`, the 7th ugly number is 8.

Python Solution

```python
class Solution:
    def nth_ugly_number(self, n: int) -> int:
        # Initialize a dynamic programming array with 1 as the first ugly number
        ugly_numbers = [1] * n

        # Initialize pointers for multiples of 2, 3, and 5
        index_2, index_3, index_5 = 0, 0, 0

        # Generate ugly numbers up to the nth one
        for i in range(1, n):
            # Calculate the next multiples for 2, 3, and 5
            next_2 = ugly_numbers[index_2] * 2
            next_3 = ugly_numbers[index_3] * 3
            next_5 = ugly_numbers[index_5] * 5

            # Select the minimum of these multiples to be the next ugly number
            ugly_numbers[i] = min(next_2, next_3, next_5)

            # Increment the corresponding indices when their multiples are used
            if ugly_numbers[i] == next_2:
                index_2 += 1
            if ugly_numbers[i] == next_3:
                index_3 += 1
            if ugly_numbers[i] == next_5:
                index_5 += 1

        # Return the nth ugly number
        return ugly_numbers[n - 1]

```

Java Solution

```java
class Solution {
    // Function to find the nth ugly number
    public int nthUglyNumber(int n) {
        // Dynamic programming array to store the ugly numbers
        int[] uglyNumbers = new int[n];
        // The first ugly number is always 1
        uglyNumbers[0] = 1;

        // Pointers for multiples of 2, 3, and 5
        int index2 = 0, index3 = 0, index5 = 0;

        // Populate the uglyNumbers array
        for (int i = 1; i < n; ++i) {
            // Next multiples of 2, 3, and 5
            int nextMultipleOf2 = uglyNumbers[index2] * 2;
            int nextMultipleOf3 = uglyNumbers[index3] * 3;
            int nextMultipleOf5 = uglyNumbers[index5] * 5;

            // Next ugly number is the minimum of the next multiples of 2, 3 and 5
            uglyNumbers[i] = Math.min(nextMultipleOf2, Math.min(nextMultipleOf3, nextMultipleOf5));

            // Increment indices depending on which multiple the current ugly number is based on
            // This ensures that for each prime factor, we only count multiples in ascending order
            if (uglyNumbers[i] == nextMultipleOf2) {
                index2++;
            }
            if (uglyNumbers[i] == nextMultipleOf3) {
                index3++;
            }
            if (uglyNumbers[i] == nextMultipleOf5) {
                index5++;
            }
        }

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

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to find the nth ugly number.
    int nthUglyNumber(int n) {
        // Create a dynamic programming table to store ugly numbers.
        std::vector<int> dp(n);

        // The first ugly number is 1.
        dp[0] = 1;

        // Initialize pointers for multiples of 2, 3, and 5.
        int pointer2 = 0, pointer3 = 0, pointer5 = 0;

        // Populate the table with the next ugly numbers.
        for (int i = 1; i < n; ++i) {
            // Find the next multiples for 2, 3 and 5.
            int next2 = dp[pointer2] * 2;
            int next3 = dp[pointer3] * 3;
            int next5 = dp[pointer5] * 5;

            // Select the minimum of these multiples as the next ugly number.
            dp[i] = std::min(next2, std::min(next3, next5));

            // Increase the respective pointer if it was used.
            if (dp[i] == next2) ++pointer2;
            if (dp[i] == next3) ++pointer3;
            if (dp[i] == next5) ++pointer5;
        }

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

```

Typescript Solution

```typescript
/**
 * Calculates the nth ugly number.
 * Ugly numbers are numbers whose only prime factors are 2, 3, or 5.
 * @param n The position of the ugly number to find.
 * @return The nth ugly number.
 */
function nthUglyNumber(n: number): number {
    // Initialize dp (dynamic programming) array with the first ugly number
    let dp: number[] = [1];
    // Initialize pointers for the multiples of 2, 3, and 5
    let pointer2 = 0, pointer3 = 0, pointer5 = 0;

    for (let i = 1; i < n; ++i) {
        // Find the next multiples of 2, 3, and 5
        const nextMultipleOf2 = dp[pointer2] * 2;
        const nextMultipleOf3 = dp[pointer3] * 3;
        const nextMultipleOf5 = dp[pointer5] * 5;

        // Determine the next ugly number as the minimum of the next multiples
        const nextUglyNumber = Math.min(nextMultipleOf2, nextMultipleOf3, nextMultipleOf5);
        dp.push(nextUglyNumber); // Append it to the dp array

        // Increment the corresponding pointer if it matches the ugly number
        if (nextUglyNumber === nextMultipleOf2) {
            pointer2++;
        }
        if (nextUglyNumber === nextMultipleOf3) {
            pointer3++;
        }
        if (nextUglyNumber === nextMultipleOf5) {
            pointer5++;
        }
    }

    // Return the nth ugly number, at index n - 1
    return dp[n - 1];
}

```

Time and Space Complexity

The given Python code implements the Ugly Number II solution to find the *n*th ugly number using dynamic programming. An ugly number is a number whose prime factors are limited to 2, 3, and 5.

Time Complexity

The time complexity of the code is `O(n)`. This is because there's a single loop that runs `n-1` times (since the loop starts at 1 and runs until `n`), and within each iteration of the loop, the operations performed are constant time operations such as multiplication and comparison.

Space Complexity

The space complexity of the code is also `O(n)`. The `dp` array is the primary space-consuming data structure, which stores the ugly numbers up to the `n`th ugly number. Since we have `n` elements in the `dp` array, the space consumed is proportional to `n`.