Documentation

The problem of finding the (n)-th ugly number centers around generating a sequence of numbers where each value only has the prime factors 2, 3, and 5. Ugly numbers, by definition, are positive integers that can be expressed as a product of powers of these primes, starting from the smallest integer, 1. As we generate each number, we aim to ensure that all values in the sequence maintain the limited set of factors. Since an ugly number cannot contain other prime factors, each subsequent number in the sequence must be generated by multiplying previous ugly numbers by either 2, 3, or 5. The challenge is to keep this sequence sorted as we expand it and avoid duplication, especially as some values can be reached through different combinations of multiplications.

To tackle this problem efficiently, we utilize a min-heap data structure, which allows us to always retrieve the smallest number in the sequence, maintaining a sorted order without needing full sorting. The min-heap is initialized with the first ugly number, which is 1. From here, the algorithm repeatedly retrieves the smallest element in the heap (always the next smallest ugly number) and generates new ugly numbers by multiplying the current smallest element by 2, 3, and 5. Each new product represents a potential future ugly number. However, to prevent duplicates, we maintain a set that keeps track of all the numbers that have already been generated. This set is critical for efficiency, as it ensures that each ugly number appears only once in the heap.

The algorithm continues in a loop, removing the smallest element from the heap and pushing new ugly numbers onto the heap until the (n)-th ugly number is reached. Each iteration effectively finds the next ugly number in the sequence and expands the sequence by adding new candidates. The use of both the min-heap and set ensures that we are always working with a sorted list of unique ugly numbers, which is essential to efficiently reach the desired (n)-th position without redundant calculations. By always keeping the smallest value at the top of the heap, we avoid sorting costs and keep the addition of new numbers efficient, each push and pop operation only taking (O(log n)) time.

In terms of time complexity, this solution is highly efficient, scaling well for the problem’s constraints (up to (n = 1690)). The time complexity of (O(n log n)) arises from (n) insertions and removals from the heap, with each operation costing (O(log n)) time. The space complexity remains (O(n)) because we store the heap and set of numbers that have been processed, ensuring the solution is not overly memory-intensive. As we generate numbers, we only store as many numbers as are needed to reach the (n)-th ugly number, meaning that the algorithm is optimized for both time and space.

This method is especially useful because of its reliance on a sorted, dynamically expanding sequence without repetitive calculations. By combining heap and set data structures, the solution strikes a balance between simplicity and efficiency, allowing for a straightforward implementation that is both readable and effective. As a result, this approach provides a robust way to solve the problem within the constraints, making it suitable for applications that require fast and reliable generation of such sequences. The interplay of the min-heap and set illustrates a classic example of how carefully chosen data structures can streamline the solution of combinatorial sequence problems efficiently.