263. Ugly Number

Easy Math

Problem Description

The task is to determine if a given positive integer is an "ugly number" or not. An ugly number is defined as a number whose only prime factors are 2, 3, or 5. This means that if the number can be divided by any other prime number, it is not considered an ugly number. Examples of ugly numbers include 1, 2, 3, 4, 5, 6, 8, 9, and 10. Non-examples are 7, 11, 14, and so on. The input to the problem is a single integer `n`, and the expected output is a boolean (`true` or `false`) indicating whether `n` is an ugly number.

Intuition

To solve this problem, our approach is to iteratively divide the input number `n` by each of the prime factors 2, 3, and 5 as long as possible. This is based on the definition that an ugly number has no prime factors other than 2, 3, or 5.

Here's the thinking process behind the solution:

- Start by checking if the number `n` is less than 1. If it is, return `false` because all ugly numbers are positive integers.
- For each of the prime factors (2, 3, 5), we use a `while` loop to divide `n` by the prime factor as long as the remainder (`n % x`) is zero (this means that `n` is divisible by `x` without any remainder).
- We keep dividing `n` by the prime factor until it is no longer divisible by that factor.
- If at the end of this process, `n` has been reduced to 1, then all of `n`'s prime factors must have been among 2, 3, or 5, and thus `n` is an ugly number (return `true`).
- If `n` is not 1 after division by all three primes, it means `n` had other prime factors, and therefore, it's not an ugly number (return `false`).

The intuition behind the solution is that dividing `n` by these primes should eventually yield 1 if `n` is indeed an ugly number because all its factors would have been 'stripped' off, leaving only 1.

Solution Approach

The implementation of the provided solution is straightforward. It does not rely on any advanced algorithms, complex data structures, or specific design patterns. Instead, it uses a simple mathematical approach to break down the problem.

Here is a step-by-step walkthrough of the code provided in the reference:

1. First, it checks if the input number `n` is less than 1. This is because an ugly number must be positive. If `n` is 0 or negative, the function immediately returns `false`.

```
if n < 1:
    return False
```

2. Then, the solution initializes a loop that iterates over a list containing the prime factors [2, 3, 5]. These are the only primes that are allowed to be factors of an ugly number.

```
for x in [2, 3, 5]:
```

3. Inside the loop, there's a `while` loop that checks whether the current prime factor `x` divides the number `n` evenly. If so, it means `n` contains the prime factor `x`, and we divide `n` by `x`, updating `n` to this quotient.

```
while n % x == 0:
    n //= x
```

   This is equivalent to removing the factor `x` from `n`, and the `while` loop ensures we remove all occurrences of this factor.

4. This process is repeated for each prime factor in the list. If `n` is an ugly number, by the end of this process, all the primes 2, 3, and 5 will have been divided out, leaving `n` equal to 1.

5. Finally, the function checks if the resulting `n` is 1. If it is, it means that `n`'s only prime factors were 2, 3, and 5, and so it returns `true`. Otherwise, it had other prime factors, and it returns `false`.

```
return n == 1
```

In terms of data structures, only a simple list containing the three allowed prime factors is used. The use of `%` (modulus) and `//=` (floor division) operators are crucial for the arithmetic manipulations necessary to determine if `n` is an ugly number.

The simplicity and beauty of this solution lie in its direct transformation of the definition of an ugly number into code. By repeatedly dividing by the allowed factors, we effectively strip `n` down to its essence. If it remains standing as 1, it confirms its identity as an ugly number.

Example Walkthrough

Let's say we want to determine if 14 is an ugly number using the solution approach described above.

We'll follow the steps outlined:

1. Check if the input number `n` is less than 1. In our case, `n = 14`, which is greater than 1, so we proceed.
2. We iterate over the list `[2, 3, 5]`—which represents the prime factors of an ugly number—in a loop.
3. We start with the first element in the list, which is 2. We enter a `while` loop and check if `n % 2 == 0`.
   - For `n = 14`, `n % 2 == 0` is `true`, so we divide `n` by 2, which gives us `n = 7`.
4. We exit the `while` loop for the factor 2 because `n % 2 == 0` is no longer true and proceed to the next prime factor, which is 3.
5. We check if `n % 3 == 0`. For `n = 7`, this is `false`, so we do not enter the `while` loop for the factor 3. We move to the next factor, which is 5.
6. We check if `n % 5 == 0`. Again, for `n = 7`, this is `false`, and we do not enter the `while` loop for the factor 5 either.
7. After dividing by 2, and not being able to divide by 3 or 5, our value of `n` is still `7`. We reach the end of our algorithm and check if `n == 1`.
   - Since `n` is not equal to 1, in this case, we conclude that 14 is not an ugly number, and the function returns `false`.

Now let's go through with an ugly number, such as 6:

1. We check if `n` is less than 1. For `n = 6`, this is not true, so we proceed.
2. We start the loop over `[2, 3, 5]`.
3. We check if `n % 2 == 0`. For `n = 6`, this is true, so we divide `n` by 2. Now `n = 3`.
4. We check if `n % 2 == 0` again. Since `n = 3`, it is not true, so we move to the next factor, 3.
5. We check if `n % 3 == 0`. Since `n = 3`, this is true, so we divide `n` by 3, and now `n = 1`.
6. We move to the next factor, 5, but since `n` is already `1`, there is no need to proceed further.
7. We check if `n == 1`. Since it is, we return `true`, determining that 6 is indeed an ugly number according to our algorithm.

Solution Implementation

Python

```python
class Solution:
    def isUgly(self, number: int) -> bool:
        # The number must be positive to be considered ugly
        if number < 1:
            return False

        # Define the prime factors specific to ugly numbers
        prime_factors = [2, 3, 5]

        # Divide the number by each of the prime factors as long as it's divisible
        for factor in prime_factors:
            while number % factor == 0:
                number //= factor

        # If the resulting number is 1, it's an ugly number
        return number == 1
```

Java

```java
class Solution {

    /**
     * Determines if the given number is an ugly number.
     * Ugly numbers are positive numbers whose prime factors only include 2, 3, and 5.
     *
     * @param num The number to check for ugliness.
     * @return true if num is an ugly number, otherwise false.
     */
    public boolean isUgly(int num) {
        // A non-positive number cannot be an ugly number
        if (num < 1) {
            return false;
        }

        // Divide num by 2 as long as it is divisible by 2
        while (num % 2 == 0) {
            num /= 2;
        }

        // Divide num by 3 as long as it is divisible by 3
        while (num % 3 == 0) {
            num /= 3;
        }

        // Divide num by 5 as long as it is divisible by 5
        while (num % 5 == 0) {
            num /= 5;
        }

        // If num has been reduced to 1, then it only contains the prime factors 2, 3, and/or 5
        return num == 1;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to check if a number is ugly
    bool isUgly(int number) {
        // An ugly number must be positive
        if (number < 1) return false;

        // Divide the number by 2 as long as it is even
        while (number % 2 == 0) {
            number /= 2;
        }

        // Divide the number by 3 as long as it is divisible by 3
        while (number % 3 == 0) {
            number /= 3;
        }

        // Divide the number by 5 as long as it is divisible by 5
        while (number % 5 == 0) {
            number /= 5;
        }

        // If the resulting number is 1,
        // it is an ugly number
        return number == 1;
    }
};
```

TypeScript

```typescript
/**
 * Determine if a number is "ugly".
 * Ugly numbers are positive numbers whose prime factors only include 2, 3, 5.
 *
 * @param {number} num - The number to check.
 * @returns {boolean} - `true` if the number is ugly; otherwise, `false`.
 */
const isUgly = (num: number): boolean => {
    // If the number is less than 1, it's not an ugly number
    if (num < 1) return false;

    // Divide the number by 2 as long as it's divisible by 2
    while (num % 2 === 0) {
        num /= 2;
    }

    // Divide the number by 3 as long as it's divisible by 3
    while (num % 3 === 0) {
        num /= 3;
    }

    // Divide the number by 5 as long as it's divisible by 5
    while (num % 5 === 0) {
        num /= 5;
    }

    // If the remaining number is 1, then it's an ugly number
    // otherwise, it's not.
    return num === 1;
};

// Example usage:
// const result = isUgly(6); // Should return `true`
// const result = isUgly(14); // Should return `false`


class Solution:
    def isUgly(self, number: int) -> bool:
        # The number must be positive to be considered ugly
        if number < 1:
            return False

        # Define the prime factors specific to ugly numbers
        prime_factors = [2, 3, 5]

        # Divide the number by each of the prime factors as long as it's divisible
        for factor in prime_factors:
            while number % factor == 0:
                number //= factor

        # If the resulting number is 1, it's an ugly number
        return number == 1
```

Time and Space Complexity

The time complexity of the code is `O(log n)`. This is because the input number `n` is continuously divided by 2, 3, or 5 in the worst case, which would only happen log n times before it reduces to 1 or a number that is not divisible by 2, 3, or 5. In practice, as soon as `n` is no longer divisible by these primes, the loop ends.

The space complexity of the code is `O(1)`. This is because the function uses a constant amount of space; only a fixed number of variables are introduced, and no additional structures are dependent on the input size.