470. Implement Rand10() Using Rand7()

Medium Math Rejection Sampling Probability and Statistics Randomized

Problem Description

The problem provides a `rand7()` API, which produces a uniformly distributed random integer in the range from 1 to 7. The objective is to create a new function `rand10()` that generates a random integer in the range from 1 to 10 with a uniform distribution, utilizing only the `rand7()` function. It is important to achieve this without using any other random functions provided by the programming language's built-in libraries. Additionally, the function `rand10()` will be called `n` times during testing, where `n` is an internal argument used for testing purposes. It is crucial that the distribution of the numbers generated by `rand10()` is uniform, meaning each number from 1 to 10 has an equal probability of occurrence, leveraging the randomness provided by `rand7()`.

Intuition

To solve this problem, the idea is to find a way to generate a range that is a multiple of 10 using rand7(), because this would allow us to easily map the results to a 1-10 range uniformly. Since rand7() produces numbers from 1 to 7, we can simulate a larger range by treating one call to rand7() as the digit in one place of a base-7 number, and another call as the digit in another place.

Here's the reasoning:

1. Call `rand7()` to get a number `i` between 0 and 6 (inclusive) by subtracting 1 from the result.
2. Call `rand7()` again to get a number `j` between 1 and 7 (inclusive).
3. Combine `i` and `j` to generate a number `x = i * 7 + j`. The number `x` is now uniformly distributed between 1 and 49 because `i` has 7 possible outcomes and `j` also has 7 possible outcomes, which means we have `7 * 7 = 49` possibilities.

But, we need a range that is a multiple of 10 to map to the range 1 to 10. So what we do is:

- We only use the results 1 through 40 from `x`. This ensures that when `x` is within this range, each number has an equal probability of occurring because 40 is a multiple of 10.
- If `x` is greater than 40, we discard it and try again. This way we make sure every result from 1 to 10 will have an equal likelihood.
- The result needs to be in the range 1 to 10, so we take `x % 10` which gives us a range from 0 to 9, then add 1 to shift this range to 1 to 10.

The process of discarding numbers and trying again is called rejection sampling, which ensures we can get a uniform distribution in the desired range.

Solution Approach

The solution approach for implementing the `rand10()` function using the `rand7()` API involves the concept of rejection sampling, which is a technique where you generate a sample and only use it if it falls within a certain range. The aim is to produce a uniform distribution between 1 and 10 by generating a larger uniform distribution and narrowing it down.

Here is a step-by-step breakdown of the algorithm used in the given implementation:

1. Start an infinite loop to keep trying until a valid sample is produced. This loop will terminate once we get a number in the desired range.
2. Generate two independent numbers `i` and `j` by calling `rand7()`. We use `i = rand7() - 1` to get a number from 0 to 6, and `j` is just the output from `rand7()`, which ranges from 1 to 7.

   This step effectively simulates rolling two 7-sided dice, one to determine the tens' place (with possible results 0 to 6 corresponding to 00, 07, 14, ..., 42) and one to determine the ones' place (with possible results 1 to 7). You could imagine it as creating a two-digit base-7 number (`i` being the first digit and `j` the second digit).
3. Compute `x = i * 7 + j`, which gives us a uniform distribution in the range of 1 to 49 because there are 7 possible states the `i` can take on, and for each state of `i`, there are 7 possible states of `j`. Therefore, the total possible outcomes are `7 * 7 = 49`.
4. Use rejection sampling to discard values of `x` greater than 40. This rejection is necessary because we want to be able to evenly distribute outcomes in the range of 1 to 10, and we cannot do that with 49 outcomes since 49 is not divisible by 10. The closest number less than 49 that is divisible by 10 is 40, so we limit `x` to this range.
5. If `x` is less than or equal to 40, we take the modulo of `x` with 10, which gives a result ranging from 0 to 9. By adding 1 to this result, we shift the range to 1 to 10, the desired outcome for `rand10()`.
6. Return the final result which is now guaranteed to be uniformly distributed between 1 and 10.

No additional data structures are needed for this solution; only variables to store the two numbers generated by `rand7()` and to calculate `x`.

In summary, the algorithm ensures a uniform distribution for the `rand10()` function by creating a larger uniform distribution using `rand7()`, and narrowing it down using rejection sampling and modulo operation to fit within the required range.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we want to generate a random number from 1 to 10 using the `rand7()` function. We'll show one potential sequence of events:

1. We start our process and enter an infinite loop where we will keep generating numbers until we get a result less than or equal to 40. Let's say we call `rand7()` and it returns 5. According to our algorithm, we need to subtract 1 to convert this to a 0-based range, so we now have `i = 4`.
2. We call `rand7()` again for our second number and it returns 3. We do not modify this number, so `j = 3`.
3. Next, we compute `x` using the formula `x = i * 7 + j`. Substituting in our values, we get `x = 4 * 7 + 3 = 31`. Since the result, 31, falls in the range of 1 to 40, we can use it.
4. We use rejection sampling to check if the value of `x` (31) is less than or equal to 40. In this case, it is true, so there is no need to discard this value and retry.
5. The next step is to translate our `x` range of 1 to 40 to 1 to 10, so we take `x % 10`. For our example, this is `31 % 10`, which equals 1.
6. The final result of 1 is not in the range of 1 to 10, so we add 1 to shift the range: `1 + 1` equals 2.
7. We have our final result for this iteration: 2. This number is a valid output of our `rand10()` function and is uniformly distributed across the range from 1 to 10. If `rand10()` were to be called multiple times, each number from 1 to 10 would have approximately a 1 in 10 chance of being produced, satisfying the conditions of the problem.

Solution Implementation

Python

```python
class Solution:
    def rand10(self):
        """
        Generate a random integer in the range 1 to 10 using the provided rand7() function.

        :rtype: int
        """
        while True:
            # Generate two independent numbers from 1 to 7.
            # Subtract 1 from the first number to make it range from 0 to 6.
            row = rand7() - 1
            col = rand7()

            # Calculate a unique number in the range 1 to 49 (7x7 grid)
            value = row * 7 + col

            # If the number is within the first 40 numbers, use it for a uniform distribution from 1 to 10.
            if value <= 40:
                # The modulo operation ensures a uniform distribution [0, 9].
                # Adding 1 adjusts the range to [1, 10].
                return value % 10 + 1
```

Java

```java
class Solution extends SolBase {
    public int rand10() {
        // Continue the loop until a suitable number is generated
        // which can be scaled down to the range 1 to 10
        while (true) {
            // Generate a number from 0 to 6 using rand7()
            int row = rand7() - 1;
            // Generate another number from 1 to 7 using rand7()
            int col = rand7();
            // Calculate index in a 7x7 matrix
            int idx = row * 7 + col;
            // Check if the index is within the range we can use
            // Which is the first 40 numbers of the 7x7 matrix.
            // This is important to maintain the uniform distribution of rand10.
            if (idx <= 40) {
                // Use modulus to scale the result to be within 1 to 10 and return
                return idx % 10 + 1;
            }
            // If index is greater than 40, reject it and try again
        }
    }
}
```

C++

```cpp
// The rand7() API is already defined for the user.
// int rand7();
// @return a random integer in the range 1 to 7

class Solution {
public:
    int rand10() {
        while (true) {
            // Generate two random numbers using rand7
            int row = rand7() - 1;  // Subtracting 1 to get a range from 0 to 6.
            int col = rand7();      // Keeping range as 1 to 7.

            // Calculate a new index from the two random numbers to get a range from 1 to 49.
            int index = row * 7 + col;

            // Check if the index is within the range we can use to generate a random number from 1 to 10.
            if (index <= 40) {
                // Use the modulo operation to get a final result in the range from 1 to 10.
                return index % 10 + 1;
            }
            // If the index is greater than 40, discard the number and try again.
            // This is done to avoid a skewed distribution that could occur due to the reject sampling.
        }
    }
};
```

TypeScript

```typescript
/**
 * Utilizes the predefined rand7() function to generate a random number between 1 and 10.
 * @return {number} A random integer in the range 1 to 10
 */
function rand10(): number {
    while (true) {
        // Generate two independent numbers from the rand7() to increase the range
        const num1 = rand7() - 1; // Subtract 1 to make it from 0 to 6, enabling multiplication
        const num2 = rand7();

        // Combine the two numbers to get a number in the range of 1 to 49
        const combinedNum = num1 * 7 + num2;

        // Check if the generated number can be evenly distributed within 1-10
        if (combinedNum <= 40) {
            // If within the desired range, use modulo operation to get a number from 1 to 10
            return (combinedNum % 10) + 1;
        }

        // If the number is greater than 40, repeat the process
        // This ensures that each number from 1 to 10 has an equal probability of being returned
    }
}
```

```python
class Solution:
    def rand10(self):
        """
        Generate a random integer in the range 1 to 10 using the provided rand7() function.

        :rtype: int
        """
        while True:
            # Generate two independent numbers from 1 to 7.
            # Subtract 1 from the first number to make it range from 0 to 6.
            row = rand7() - 1
            col = rand7()

            # Calculate a unique number in the range 1 to 49 (7x7 grid)
            value = row * 7 + col

            # If the number is within the first 40 numbers, use it for a uniform distribution from 1 to 10.
            if value <= 40:
                # The modulo operation ensures a uniform distribution [0, 9].
                # Adding 1 adjusts the range to [1, 10].
                return value % 10 + 1
```

Time and Space Complexity

The given Python code uses a rejection sampling method to generate a uniform distribution from 1 to 10 using the `rand7()` function.

Time Complexity:

The time complexity of `rand10()` is not constant, as it depends on the number of times the while loop executes before generating a number less than or equal to 40. Since we generate a number between 1 and 49 (`7 * 7` possibilities), and only the numbers from 1 to 40 are used, the probability `p` of stopping at each iteration is `40/49`. Thus, the expected number of iterations `E` is `1/p`, which is `49/40`. Due to the constant work in each iteration, the expected time complexity is `O(1/p) = O(49/40)`, which simplifies to `O(1)` since we disregard constants in Big O notation.

However, please note that this is the expected time complexity. The worst-case time complexity is unbounded because in theory, it's possible (though extremely unlikely) that the while loop could run indefinitely if the condition `x <= 40` is never met.

Space Complexity:

The space complexity of the `rand10()` method is `O(1)` as it uses only a constant amount of additional space. Variables `i`, `j`, and `x` are used, but their space usage does not scale with the size of the input, so the space complexity is constant.