478. Generate Random Point in a Circle

Medium | Geometry | Math | Rejection Sampling | Randomized | Leetcode Link

Problem Description

Given a circle defined by its `radius` and the `(x_center, y_center)` coordinates of its center, the task is to create a function `randPoint` that can generate a random point within the boundaries of this circle. A valid random point could lie anywhere from the center of the circle to the circumference, with each potential point inside the circle having an equal chance of being selected.

Intuition

The intuitive approach to randomly generating a point within a circle involves two steps - finding a random distance from the center within the range [0, radius] and finding a random angle to pair with this distance.

Here's a breakdown of the solution approach:

1. **Generate a random radius**: This should be between 0 and the radius of the circle. But we can't simply pick a uniform random number directly between these two because we're working in two dimensions. If we did that, there would be a higher concentration of points near the circumference than near the center — which wouldn't be uniformly random. To get a uniform distribution, we take the square root of a random number between 0 and the square of the radius. In a 2D space, the area of a circle increases with the square of the radius, so this method ensures a uniform distribution of points by their area.

2. **Generate a random angle**: Angles in a circle range from 0 to 2π radians, representing 0 to 360 degrees. Each point within the circle corresponds to an angle from the center. By choosing a random angle from this range, we can cover all possible directions uniformly.

3. **Calculate the coordinates**: With the random length and angle, calculate the x and y coordinates of the random point. We use the random radius (length) to determine how far from the center the point is, and the random angle (degree) to determine the direction. Using the formulas `x = centerX + length * cos(degree)` and `y = centerY + length * sin(degree)`, we can find the position of the random point in a Cartesian coordinate system where `centerX` and `centerY` are the coordinates of the center of the circle.

These steps ensure a uniformly random distribution of generated points within the circle.

Solution Approach

The implementation of the `Solution` class in Python takes advantage of the `math` module for mathematical functions such as `math.sqrt` for square root and `math.cos` and `math.sin` for cosine and sine functions. It also uses `random.uniform` to generate a floating-point number within a range. Here is a walk-through of the solution.

1. **Class Initialization**: The `__init__` method initializes the instance of the class with the `radius`, `x_center`, and `y_center`.

~~~
class Solution:
    def __init__(self, radius: float, x_center: float, y_center: float):
        self.radius = radius
        self.x_center = x_center
        self.y_center = y_center
~~~

   This is simple setup code that stores the inputs to be used in the `randPoint` method.

2. **Random Point Generation**:

   The `randPoint` method is where the random point within the circle is generated.

   a. Generate a random length from the circle's center by using the square root method described in the intuition section to ensure a uniform distribution.

~~~
```python
length = math.sqrt(random.uniform(0, self.radius**2))
```

`random.uniform(0, self.radius**2)` picks a number uniformly between `0` and the square of the `radius`, and then `math.sqrt` c
~~~

   b. Generate a random angle in radians. Any angle for a full rotation around a circle is between `0` and `2π`. This is represented by the following line of code:

~~~
```python
degree = random.uniform(0, 1) * 2 * math.pi
```

`random.uniform(0, 1)` generates a number between `0` and `1`, multiplying this by `2 * math.pi` scales it to the range of `[0,
~~~

   c. Compute the x and y coordinates based on the random `length` and `degree`. The cosine and sine functions translate the random length and direction into Cartesian coordinates:

~~~
```python
x = self.x_center + length * math.cos(degree)
y = self.y_center + length * math.sin(degree)
```

Here, `length * math.cos(degree)` finds the horizontal distance from the center, and `length * math.sin(degree)` finds the vert
~~~

   d. Return a list containing the x and y coordinates of the random point:

~~~
```python
return [x, y]
```
~~~

   This method can be called multiple times to generate different points within the circle each time it is called.

All of these steps come together to form the `randPoint` method of the `Solution` class which fulfills the problem requirement of generating a random and uniformly distributed point within a given circle.

Example Walkthrough

Let's say we're given a circle with a radius of 5 units and a center at coordinates (2, 3). We want to use the `Solution` class to generate a random point within this circle.

First, we initialize the `Solution` class with our circle's parameters:

```
my_circle = Solution(5, 2, 3)
```

To generate a random point within the circle defined by `my_circle`, we'll walk through the `randPoint` method inside the `Solution` class.

1. **Generate a random radius (length)**:

```
length = math.sqrt(random.uniform(0, my_circle.radius**2))
```

Imagine the `random.uniform` function returns `16` after being called with parameters `0` and `25` (since `5 squared is 25`). So `math.sqrt(16)` is calculated, resulting in `4` units. This is our random radius, which is uniformly distributed within the circle.

2. **Generate a random angle (degree)**:

```
degree = random.uniform(0, 1) * 2 * math.pi
```

Here, `random.uniform` returns approximately `0.5`, and when this is multiplied by `2 * math.pi` (approximately `6.283`), we get roughly `3.142`, which is equivalent to 180 degrees in radians. So, our random angle is essentially pointing to the left (west) direction if we imagine the center of the circle corresponding to a compass.

3. **Calculate the coordinates (x, y)**:

```
x = my_circle.x_center + length * math.cos(degree)
y = my_circle.y_center + length * math.sin(degree)
```

With `length = 4` and `degree ≈ 3.142`, the computations will approximate to:

- `math.cos(3.142)` ≈ `-1` and `math.sin(3.142)` ≈ `0`.
- So, `x ≈ my_circle.x_center + 4 * (-1)` → `x ≈ 2 - 4` → `x ≈ -2`.
- `y ≈ my_circle.y_center + 4 * 0` → `y ≈ 3`.

Thus, the random point's coordinates are approximately `(-2, 3)`.

4. **Return the coordinates**:

```
return [x, y]
```

The final result will be a coordinate list `[-2, 3]`, which is a random point generated inside our circle of radius 5 with a center at (2, 3).

This example shows how each call to `randPoint()` would generate a different random point, uniformly distributed within the circle defined by the radius and center provided to the `Solution` class during initialization.

Python Solution

```python
import math
import random
from typing import List

class Solution:
    def __init__(self, radius: float, x_center: float, y_center: float):
        # Initialize the Solution object with the center coordinates and radius of the circle.
        self.radius = radius
        self.x_center = x_center
        self.y_center = y_center

    def rand_point(self) -> List[float]:
        # Generate a random length within the range [0, radius] with uniform distribution.
        # The length is sqrt(R^2 * U) where R is the radius and U is a uniform random number [0,1)
        length = math.sqrt(random.uniform(0, self.radius**2))

        # Generate a random angle between 0 and 2π for uniform distribution along the circumference.
        degree = random.uniform(0, 1) * 2 * math.pi

        # Calculate the x and y coordinates using the length and angle, relative to the center.
        x = self.x_center + length * math.cos(degree)  # Horizontal offset from center.
        y = self.y_center + length * math.sin(degree)  # Vertical offset from center.

        # Return the random point as a list of its x and y coordinates.
        return [x, y]

# Note: The method name is preserved as 'rand_point' in compliance with the instruction not to modify method names.
# However, typically in Python, method names would also follow the snake_case convention.
```

Java Solution

```java
import java.util.Random;

public class Solution {
    private double radius;
    private double xCenter;
    private double yCenter;
    private Random random;

    public Solution(double radius, double xCenter, double yCenter) {
        // Initialize the Solution object with the center coordinates and radius of the circle.
        this.radius = radius;
        this.xCenter = xCenter;
        this.yCenter = yCenter;
        this.random = new Random(); // Initialize the random instance for generating random numbers.
    }

    public double[] randPoint() {
        // Generate a random length within the range [0, radius] with uniform distribution.
        // The length is sqrt(radius^2 * U) where U is a uniform random number in [0, 1).
        double length = Math.sqrt(this.radius * this.radius * random.nextDouble());

        // Generate a random angle between 0 and 2π for uniform distribution along the circumference.
        double angle = random.nextDouble() * 2 * Math.PI;

        // Calculate the x and y coordinates using the length and angle, relative to the center.
        double x = this.xCenter + length * Math.cos(angle); // Horizontal offset from the center.
        double y = this.yCenter + length * Math.sin(angle); // Vertical offset from the center.

        // Return the random point as an array of its x and y coordinates.
        return new double[]{x, y};
    }
}
```

C++ Solution

```cpp
#include <cmath>
#include <vector>
#include <random>

class Solution {
private:
    double radius;
    double x_center;
    double y_center;
    std::default_random_engine generator;
    std::uniform_real_distribution<double> distribution;

public:
    Solution(double radius, double x_center, double y_center)
        : radius(radius), x_center(x_center), y_center(y_center), distribution(0.0, 1.0) {
        // Initialize the Solution object with the center coordinates and radius of the circle.
    }

    std::vector<double> rand_point() {
        // Generate a random length within the range [0, radius] with uniform distribution.
        // The length is sqrt(R^2 * U) where R is the radius and U is a uniform random number [0,1)
        double length = std::sqrt(distribution(generator) * (radius * radius));

        // Generate a random angle between 0 and 2π for uniform distribution along the circumference.
        double angle = distribution(generator) * 2 * M_PI;

        // Calculate the x and y coordinates using the length and angle, relative to the center.
        double x = x_center + length * std::cos(angle); // Horizontal offset from center.
        double y = y_center + length * std::sin(angle); // Vertical offset from center.

        // Return the random point as a vector of its x and y coordinates.
        return {x, y};
    }
};

// Usage example
int main() {
    Solution solution(10.0, 5.0, 5.0);
    std::vector<double> point = solution.rand_point();
    // Now `point` contains the random coordinates
}
```

Typescript Solution

```typescript
import * as math from "mathjs";

// Define the circle's properties globally
let radius: number;
let x_center: number;
let y_center: number;

// Function to initialize the global variables with the center coordinates and radius of the circle
function initCircle(newRadius: number, newX_center: number, newY_center: number): void {
    radius = newRadius;
    x_center = newX_center;
    y_center = newY_center;
}

// Function to generate a random point within the circle
function randPoint(): number[] {
    // Generate a random length within the range [0, radius] with uniform distribution
    // The length is the square root of (radius squared multiplied by a uniform random number [0, 1))
    const length: number = Math.sqrt(Math.random() * (radius ** 2));

    // Generate a random angle between 0 and 2π for uniform distribution along the circumference
    const angle: number = Math.random() * 2 * Math.PI;

    // Calculate the x and y coordinates using the length and angle, relative to the center
    const x: number = x_center + length * Math.cos(angle); // Horizontal offset from center
    const y: number = y_center + length * Math.sin(angle); // Vertical offset from center

    // Return the random point as an array of its x and y coordinates
    return [x, y];
}

// Note: Although we have removed the class definition and included the methods globally,
// in a typical TypeScript environment, we would still use a class or module to encapsulate these functions.
// Additionally, 'import * as math from "mathjs";' is used instead of the default 'import math',
// as TypeScript does not natively have a math module.
```

Time and Space Complexity

Time Complexity

The given code involves calculating a random point within a circle. The `randPoint` method contains a constant number of operations regardless of the input size:

- `random.uniform(0, self.radius**2)` computes a uniform random value between 0 and the square of the radius, which takes constant time O(1).
- Taking the square root with `math.sqrt()` is also a constant time operation O(1).
- `random.uniform(0, 1) * 2 * math.pi` computes a random angle, which is again constant time O(1).
- The sine and cosine functions are computed once per call to `randPoint`, both of which take constant time O(1).
- Multiplying the length and angle calculations to get the `x` and `y` coordinates are basic arithmetic operations with constant time O(1).

Since all the operations are executed a constant number of times, the time complexity of the `randPoint` method is O(1).

Space Complexity

As for space complexity:

- The `Solution` class holds three variables: `self.radius`, `self.x_center`, and `self.y_center`. This space usage does not scale with the number of times `randPoint` is called, and they are only stored once when the class instance is created.
- Temporary variables used in `randPoint` for length, degree, `x`, and `y` are re-created on each call and do not accumulate. Thus, the space required for these variables is constant.

Given that no additional space is used that scales with the size of the input or the number of operations, the space complexity of the `randPoint` method is O(1).