RandomGen - Weighted Random Number Generator

Project Overview

A production-ready implementation of a random number generator with specified probabilities. It allows you to define a set of possible values and their respective probabilities of occurrence, then efficiently generate random numbers that follow this distribution.

Features

- **High Performance**: O(log n) lookup algorithm using binary search on cumulative probabilities
- **Thread-Safe**: Safe for concurrent use from multiple threads
- **Deterministic Testing**: Support for custom seeding to enable reproducible test results
- **Well-Tested**: Comprehensive test suite with statistical validation
- **Production-Ready**: Robust error handling, documentation, and CI/CD pipeline
- **Cross-Platform**: Works on Windows, macOS, and Linux

Implementation Details

Core Algorithm

The RandomGen class uses the following approach to generate weighted random numbers:

1. **Initialization**:
    - Validate input arrays (sizes match, probabilities sum to 1, etc.)
    - Precompute cumulative probabilities using `std::partial_sum`
    - Set up random number generator with mutex protection
2. **Random Number Generation**:
    - Generate a uniform random value between 0 and 1
    - Use binary search (`std::lower_bound`) to find the appropriate index in the cumulative probability array
    - Return the corresponding value from the randomNums array
3. **Custom Seeding**:
    - Set a specific seed for deterministic output
    - Seed is shared across all instances of RandomGen and all threads
    - Perfect for reproducible testing and simulation

Time Complexity

- **Construction**: O(n) where n is the number of possible values
- **nextNum()**: O(log n) due to binary search on the cumulative probabilities
- **Memory Usage**: O(n) to store the input arrays and cumulative probabilities

Thread Safety

RandomGen is thread-safe by design:

- Uses mutex protection for the shared random number generator
- All member variables are initialized once and become immutable after construction
- Custom seeding is synchronized across all threads

Getting Started

Option 1: Using Precompiled Binaries

Precompiled binaries for Windows, macOS, and Linux are available on the Releases page.

1. Download the appropriate zip file for your platform
2. Extract the contents
3. Run the example or test executable:

```
# Run example
./RandomGenExample

# Run tests
./RandomGenTests
```

Option 2: Using Docker

If you have Docker installed, you can run RandomGen without any other dependencies:

```
# Pull the image
docker pull ghcr.io/ayanchev01/randomgen:latest

# Run the example
docker run --rm ghcr.io/ayanchev01/randomgen:latest example

# Run the tests
docker run --rm ghcr.io/ayanchev01/randomgen:latest tests

# Run both
docker run --rm ghcr.io/ayanchev01/randomgen:latest all
```

Option 3: Building From Source

Prerequisites

- C++20 compatible compiler (GCC 10+, Clang 10+, MSVC 2019+)
- CMake 3.15 or higher
- Git (optional, for retrieving GoogleTest)

Build Instructions

Clone the repository:

```
git clone https://github.com/AYanchev01/RandomGen.git
cd RandomGen
```

Build the project:

```
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
cmake --build . --config Release
```

Run the example:

```
./bin/Release/RandomGenExample
```

Run the tests:

```
./bin/Release/RandomGenTests
```

API Documentation

RandomGen Class

```
class RandomGen {
public:
    /**
     * @brief Construct a new RandomGen object
     *
     * @param randomNums Values that may be returned by nextNum()
     * @param probabilities Probability of occurrence for each value
     * @throw std::invalid_argument if inputs are invalid
     */
    RandomGen(const std::vector<int>& randomNums, const std::vector<double>&
probabilities);

    /**
     * @brief Returns one of the randomNums based on their probabilities
     *
     * When called multiple times over a long period, it returns numbers
     * roughly with the initialized probabilities.
```

```
     *
     * This method is thread-safe and can be called concurrently from multiple
threads.
     *
     * @return int A randomly selected number
     */
    int nextNum() noexcept;

    /**
     * @brief Sets a custom seed for the random number generator
     *
     * This method affects all instances of RandomGen across all threads.
     * Useful for deterministic testing and reproducible simulations.
     *
     * @param seed The seed value to use
     */
    static void setSeed(unsigned int seed);
};
```

Usage Example

```
#include "RandomGen.h"
#include <iostream>
#include <vector>

int main() {
    // Define possible values and their probabilities
    std::vector<int> randomNums = {-1, 0, 1, 2, 3};
    std::vector<double> probabilities = {0.01, 0.3, 0.58, 0.1, 0.01};

    // Optional: Set a custom seed for deterministic output
    RandomGen::setSeed(12345);

    // Create random generator
    RandomGen randomGen(randomNums, probabilities);

    // Generate and print random numbers
    for (int i = 0; i < 10; ++i) {
        std::cout << randomGen.nextNum() << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Testing

The project includes a comprehensive test suite that validates:

1. **Input Validation**:

    - Handling of invalid probability distributions (not summing to 1)
    - Handling of negative probabilities
    - Handling of mismatched array lengths
    - Handling of empty arrays
    - Handling of edge cases (probabilities very close to 1)

2. **Functionality**:

    - Single value with probability 1
    - Uniform distribution behavior
    - Distribution matching specified probabilities
    - Statistical validation using confidence intervals

3. **Thread Safety**:

    - Concurrent access from multiple threads

4. **Deterministic Testing**:

    - Custom seeding for reproducible results
    - Seed propagation across threads
    - Verification of specific sequences with known seeds

Continuous Integration/Continuous Delivery

The project includes a full CI/CD pipeline implemented with GitHub Actions:

Continuous Integration

- **Build Matrix**: Builds and tests on Windows, macOS, and Linux with both Debug and Release configurations
- **Code Quality**: Static analysis with cppcheck and clang-tidy
- **Code Coverage**: Measures test coverage and uploads to Codecov

Continuous Delivery

- **Cross-platform Builds**: Automated builds for Windows, macOS (universal binary for Intel and Apple Silicon), and Linux
- **Release Automation**: Creates GitHub releases with precompiled binaries
- **Docker Images**: Builds and pushes Docker images to GitHub Container Registry