GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Seminar Algorithms for Big Data

Fast Random Integer Generation in an Interval

Based on a paper of the same title by Daniel Lemire

Lukas Geis

Supervised by Dr. Manuel Penschuck

29th February 2024 · Algorithm Engineering (Prof. Dr. Ulrich Meyer)

What is our goal?

What is our goal?

We want to *efficiently* draw a *uniform* random integer in an interval.

What is our goal?

We want to *efficiently* draw a *uniform* random integer in an interval.

Where do we need this?

What is our goal?

We want to *efficiently* draw a *uniform* random integer in an interval.

Where do we need this?

- Shuffling

TBD

What is our goal?

We want to *efficiently* draw a *uniform* random integer in an interval.

Where do we need this?

- Shuffling
- Complex Graph Generators

TBD TBD

What is our goal?

We want to *efficiently* draw a *uniform* random integer in an interval.

Where do we need this?

- Shuffling
- Complex Graph Generators
- Sampling

TBD TBD TBD

Table of Contents

1. **Preliminaries**
   - Formal Definition
   - Operations
   - The Naive Approach
2. **Unbiased Algorithms**
   - The OpenBSD Algorithm
   - The Java Algorithm
   - The Fast-Dice-Roller Algorithm
   - The Bitmask Algorithm
3. **Lemire's Algorithm**
   - Multiply-And-Shift
   - The Algorithm
4. **Conclusion**

1 Preliminaries

Formal Definition

Formal Definition

Setting:

Formal Definition

Setting:

- **Input:** upper bound of interval $n \in \mathbb{N}$

Formal Definition

Setting:

- **Input:** upper bound of interval $n \in \mathbb{N}$
- **Output:** uniform random integer in interval $[0, n)$

Formal Definition

Setting:

- **Input:** upper bound of interval $n \in \mathbb{N}$
- **Output:** uniform random integer in interval $[0, n)$

But what if we want a random integer in $[a, b)$ for $a, b \in \mathbb{N}$, $0 < a < b$ instead?

Formal Definition

Setting:

- **Input:** upper bound of interval $n \in \mathbb{N}$
- **Output:** uniform random integer in interval $[0, n)$

But what if we want a random integer in $[a, b)$ for $a, b \in \mathbb{N}$, $0 < a < b$ instead?

We can map this to our setting by subtracting a!

Formal Definition

Setting:

- **Input:** upper bound of interval $n \in \mathbb{N}$
- **Output:** uniform random integer in interval $[0, n)$

But what if we want a random integer in $[a, b)$ for $a, b \in \mathbb{N}$, $0 < a < b$ instead?

We can map this to our setting by subtracting a!

- Set $n = b - a$ and draw a uniform random integer $x \in [0, n)$

Formal Definition

Setting:

- **Input:** upper bound of interval $n \in \mathbb{N}$
- **Output:** uniform random integer in interval $[0, n)$

But what if we want a random integer in $[a, b)$ for $a, b \in \mathbb{N}$, $0 < a < b$ instead?

We can map this to our setting by subtracting a!

- Set $n = b - a$ and draw a uniform random integer $x \in [0, n)$
- Return $x + a$

Operations

Operations

Definition (Common Operations)

Operations

Definition (Common Operations)

- Integer-Division: $x \div y \quad := \left\lfloor {}^{x}\!/_{y} \right\rfloor$

Operations

Definition (Common Operations)

- Integer-Division: $x \div y \quad := \lfloor x/y \rfloor$
- Remainder-Operation: $x \bmod y := x - (x \div y)y$

Operations

Definition (Common Operations)

- Integer-Division: $x \div y := \lfloor x/y \rfloor$
- Remainder-Operation: $x \bmod y := x - (x \div y)y$
- Bit-RightShift: $x \gg W := x \div 2^W$

Operations

Definition (Common Operations)

- Integer-Division: $x \div y := \lfloor x/y \rfloor$
- Remainder-Operation: $x \bmod y := x - (x \div y)y$
- Bit-RightShift: $x \gg W := x \div 2^W$
- Bit-LeftShift: $x \ll W := x \cdot 2^W$

Operations

Definition (Common Operations)

- Integer-Division: $x \div y \quad := \lfloor x/y \rfloor$
- Remainder-Operation: $x \bmod y \quad := x - (x \div y)y$
- Bit-RightShift: $x \gg W \quad := x \div 2^W$
- Bit-LeftShift: $x \ll W \quad := x \cdot 2^W$
- Bitwise-And: $x \ \& \ y$

Operations

Definition (Common Operations)

- Integer-Division: $x \div y \quad := \lfloor x/y \rfloor$
- Remainder-Operation: $x \bmod y := x - (x \div y)y$
- Bit-RightShift: $x \gg W \quad := x \div 2^W$
- Bit-LeftShift: $x \ll W \quad := x \cdot 2^W$
- Bitwise-And: $x \;\&\; y \quad \rightarrow x \bmod 2^W := x \;\&\; (2^W - 1)$

Operations

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Definition (Common Operations)

- Integer-Division: $x \div y \quad := \lfloor x/y \rfloor$
- Remainder-Operation: $x \bmod y \quad := x - (x \div y)y$
- Bit-RightShift: $x \gg W \quad := x \div 2^W$
- Bit-LeftShift: $x \ll W \quad := x \cdot 2^W$
- Bitwise-And: $x \ \& \ y \quad \rightarrow x \bmod 2^W := x \ \& \ (2^W - 1)$

Definition (Power Remainder)

For $W, n \in \mathbb{N}$, we write $\mathcal{R}_n^W$ for $2^W \bmod n$.

The Naive Approach

The Naive Approach

How do we get random numbers?

The Naive Approach

How do we get random numbers?

- Generated by Pseudo-Random-Number-Generators (PRNGs)

The Naive Approach

How do we get random numbers?

- Generated by Pseudo-Random-Number-Generators (PRNGs)
- Generated as W-bit words, i.e. unsigned integers in $[0, 2^W)$ (typically $W \in \{32, 64\}$)

The Naive Approach

How do we get random numbers?

- Generated by Pseudo-Random-Number-Generators (PRNGs)
- Generated as W-bit words, i.e. unsigned integers in $[0, 2^W)$ (typically $W \in \{32, 64\}$)

$$\texttt{rand()} \bmod n$$

The Naive Approach

How do we get random numbers?

- Generated by Pseudo-Random-Number-Generators (PRNGs)
- Generated as W-bit words, i.e. unsigned integers in $[0, 2^W)$ (typically $W \in \{32, 64\}$)

$$\texttt{rand()} \bmod n$$

Does this work?

The Naive Approach

How do we get random numbers?

- Generated by Pseudo-Random-Number-Generators (PRNGs)
- Generated as W-bit words, i.e. unsigned integers in $[0, 2^W)$ (typically $W \in \{32, 64\}$)

$$\texttt{rand()} \bmod n$$

Does this work?

- Yes, the generated number is in $[0, n)$.

The Naive Approach

How do we get random numbers?

- Generated by Pseudo-Random-Number-Generators (PRNGs)
- Generated as W-bit words, i.e. unsigned integers in $[0, 2^W)$ (typically $W \in \{32, 64\}$)

$$\texttt{rand()} \bmod n$$

Does this work?

- Yes, the generated number is in $[0, n)$.

Is this efficient?

The Naive Approach

How do we get random numbers?

- Generated by Pseudo-Random-Number-Generators (PRNGs)
- Generated as W-bit words, i.e. unsigned integers in $[0, 2^W)$ (typically $W \in \{32, 64\}$)

$$\texttt{rand()} \bmod n$$

Does this work?

- Yes, the generated number is in $[0, n)$.

Is this efficient?

- No, we require one expensive integer division operation.

The Naive Approach

How do we get random numbers?

- Generated by Pseudo-Random-Number-Generators (PRNGs)
- Generated as W-bit words, i.e. unsigned integers in $[0, 2^W)$ (typically $W \in \{32, 64\}$)

$$\texttt{rand()} \bmod n$$

Does this work?

- Yes, the generated number is in $[0, n)$.

Is this efficient?

- No, we require one expensive integer division operation.

Is the generated number uniform in $[0, n)$?

The Naive Approach

The Naive Approach

In general, applying $x \bmod n$ to $[0, 2^W)$ yields

The Naive Approach

In general, applying $x \bmod n$ to $[0, 2^W)$ yields

$$\overbrace{\underbrace{\overbrace{0,1,\dots,n-1}^{n\text{ values}},\overbrace{0,1,\dots,n-1}^{n\text{ values}},\dots,\overbrace{0,1,\dots,n-1}^{n\text{ values}}}_{(2^W \div n)\cdot n\text{ values}},\underbrace{0,1,\dots,\mathcal{R}_n^W-1}_{\mathcal{R}_n^W\text{ values}}}^{2^W\text{ values}}$$

The Naive Approach

In general, applying $x \bmod n$ to $[0, 2^W)$ yields

$$\overbrace{\underbrace{\overbrace{0, 1, \ldots, n-1}^{n \text{ values}}, \overbrace{0, 1, \ldots, n-1}^{n \text{ values}}, \ldots, \overbrace{0, 1, \ldots, n-1}^{n \text{ values}}}_{(2^W \div n) \cdot n \text{ values}}, \underbrace{0, 1, \ldots, \mathcal{R}_n^W - 1}_{\mathcal{R}_n^W \text{ values}}}^{2^W \text{ values}}$$

We have a leftover interval that introduces bias.

The Naive Approach

In general, applying $x \bmod n$ to $[0, 2^W)$ yields

$$\overbrace{\underbrace{\overbrace{0,1,\ldots,n-1}^{n \text{ values}},\overbrace{0,1,\ldots,n-1}^{n \text{ values}},\ldots,\overbrace{0,1,\ldots,n-1}^{n \text{ values}}}_{(2^W \div n)\cdot n \text{ values}},\underbrace{0,1,\ldots,\mathcal{R}_n^W-1}_{\mathcal{R}_n^W \text{ values}}}^{2^W \text{ values}}$$

We have a leftover interval that introduces bias.

Every approach that maps every integer in $[0, 2^W)$ to a single number in $[0, n)$

The Naive Approach

In general, applying $x \bmod n$ to $[0, 2^W)$ yields

$$\overbrace{\underbrace{\overbrace{0,1,\ldots,n-1}^{n\text{ values}},\overbrace{0,1,\ldots,n-1}^{n\text{ values}},\ldots,\overbrace{0,1,\ldots,n-1}^{n\text{ values}}}_{(2^W \div n)\cdot n\text{ values}},\underbrace{0,1,\ldots,\mathcal{R}_n^W-1}_{\mathcal{R}_n^W\text{ values}}}^{2^W\text{ values}}$$

We have a leftover interval that introduces bias.

Every approach that maps every integer in $[0, 2^W)$ to a single number in $[0, n)$ does not generate uniform random integers in one step

The Naive Approach

In general, applying $x \bmod n$ to $[0, 2^W)$ yields

$$\overbrace{\underbrace{\overbrace{0,1,\dots,n-1}^{n \text{ values}},\overbrace{0,1,\dots,n-1}^{n \text{ values}},\dots,\overbrace{0,1,\dots,n-1}^{n \text{ values}}}_{(2^W \div n)\cdot n \text{ values}},\underbrace{0,1,\dots,\mathcal{R}_n^W-1}_{\mathcal{R}_n^W \text{ values}}}^{2^W \text{ values}}$$

We have a leftover interval that introduces bias.

Every approach that maps every integer in $[0, 2^W)$ to a single number in $[0, n)$ does not generate uniform random integers in one step whenever n does not divide 2^W.

The Naive Approach

In general, applying $x \bmod n$ to $[0, 2^W)$ yields

$$\overbrace{\underbrace{\overbrace{0,1,\ldots,n-1}^{n \text{ values}},\overbrace{0,1,\ldots,n-1}^{n \text{ values}},\ldots,\overbrace{0,1,\ldots,n-1}^{n \text{ values}}}_{(2^W \div n)\cdot n \text{ values}},\underbrace{0,1,\ldots,\mathcal{R}_n^W-1}_{\mathcal{R}_n^W \text{ values}}}^{2^W \text{ values}}$$

We have a leftover interval that introduces bias.

Every approach that maps every integer in $[0, 2^W)$ to a single number in $[0, n)$ does not generate uniform random integers in one step whenever n does not divide 2^W.

Idea: Use rejection sampling to achieve uniformity!

Unbiased Algorithms

The OpenBSD Algorithm

The OpenBSD Algorithm

- We shift the rejection interval to the left:

The OpenBSD Algorithm

- We shift the rejection interval to the left:

$$\overbrace{\underbrace{0, 1, \dots, \mathcal{R}_n^W - 1}_{\mathcal{R}_n^W \text{ values}}, \underbrace{\overbrace{\mathcal{R}_n^W, \dots, n-1, 0, \dots, \mathcal{R}_n^W - 1}^{n \text{ values}}, \dots, \overbrace{\mathcal{R}_n^W, \dots, n-1, 0, \dots, \mathcal{R}_n^W - 1}^{n \text{ values}}}_{(2^W \div n) \cdot n \text{ values}}}^{2^W \text{ values}}$$

The OpenBSD Algorithm

- We shift the rejection interval to the left:

$$\overbrace{\underbrace{0, 1, \ldots, \mathcal{R}_n^W - 1}_{\mathcal{R}_n^W \text{ values}}, \underbrace{\overbrace{\mathcal{R}_n^W, \ldots, n-1, 0, \ldots, \mathcal{R}_n^W - 1}^{n \text{ values}}, \ldots, \overbrace{\mathcal{R}_n^W, \ldots, n-1, 0, \ldots, \mathcal{R}_n^W - 1}^{n \text{ values}}}_{(2^W \div n) \cdot n \text{ values}}}^{2^W \text{ values}}$$

- Generate a uniform random number $x \in [0, 2^W)$ until $x \geq \mathcal{R}_n^W$

The OpenBSD Algorithm

- We shift the rejection interval to the left:

$$\overbrace{\underbrace{0,1,\ldots,\mathcal{R}_n^W-1}_{\mathcal{R}_n^W \text{ values}},\underbrace{\overbrace{\mathcal{R}_n^W,\ldots,n-1,0,\ldots,\mathcal{R}_n^W-1}^{n \text{ values}},\ldots,\overbrace{\mathcal{R}_n^W,\ldots,n-1,0,\ldots,\mathcal{R}_n^W-1}^{n \text{ values}}}_{(2^W \div n)\cdot n \text{ values}}}^{2^W \text{ values}}$$

- Generate a uniform random number $x \in [0, 2^W)$ until $x \geq \mathcal{R}_n^W$
- Return $x \bmod n$

The OpenBSD Algorithm

- We shift the rejection interval to the left:

$$\overbrace{\underbrace{0,1,\ldots,\mathcal{R}_n^W-1}_{\mathcal{R}_n^W \text{ values}},\underbrace{\overbrace{\mathcal{R}_n^W,\ldots,n-1,0,\ldots,\mathcal{R}_n^W-1}^{n \text{ values}},\ldots,\overbrace{\mathcal{R}_n^W,\ldots,n-1,0,\ldots,\mathcal{R}_n^W-1}^{n \text{ values}}}_{(2^W \div n)\cdot n \text{ values}}}^{2^W \text{ values}}$$

- Generate a uniform random number $x \in [0, 2^W)$ until $x \geq \mathcal{R}_n^W$
- Return $x \bmod n$

Efficiency

The OpenBSD Algorithm

- We shift the rejection interval to the left:

$$\overbrace{\underbrace{0, 1, \ldots, \mathcal{R}_n^W - 1}_{\mathcal{R}_n^W \text{ values}}, \underbrace{\overbrace{\mathcal{R}_n^W, \ldots, n-1, 0, \ldots, \mathcal{R}_n^W - 1}^{n \text{ values}}, \ldots, \overbrace{\mathcal{R}_n^W, \ldots, n-1, 0, \ldots, \mathcal{R}_n^W - 1}^{n \text{ values}}}_{(2^W \div n) \cdot n \text{ values}}}^{2^W \text{ values}}$$

- Generate a uniform random number $x \in [0, 2^W)$ until $x \geq \mathcal{R}_n^W$
- Return $x \bmod n$

Efficiency

We require 2 integer division operations:

The OpenBSD Algorithm

- We shift the rejection interval to the left:

$$\overbrace{\underbrace{0, 1, \ldots, \mathcal{R}_n^W - 1}_{\mathcal{R}_n^W \text{ values}}, \underbrace{\overbrace{\mathcal{R}_n^W, \ldots, n-1, 0, \ldots, \mathcal{R}_n^W - 1}^{n \text{ values}}, \ldots, \overbrace{\mathcal{R}_n^W, \ldots, n-1, 0, \ldots, \mathcal{R}_n^W - 1}^{n \text{ values}}}_{(2^W \div n) \cdot n \text{ values}}}^{2^W \text{ values}}$$

- Generate a uniform random number $x \in [0, 2^W)$ until $x \geq \mathcal{R}_n^W$
- Return $x \bmod n$

Efficiency

We require 2 integer division operations: one for computing $\mathcal{R}_n^W$ and

The OpenBSD Algorithm

- We shift the rejection interval to the left:

$$\overbrace{\underbrace{0, 1, \ldots, \mathcal{R}_n^W - 1}_{\mathcal{R}_n^W \text{ values}}, \underbrace{\overbrace{\mathcal{R}_n^W, \ldots, n-1, 0, \ldots, \mathcal{R}_n^W - 1}^{n \text{ values}}, \ldots, \overbrace{\mathcal{R}_n^W, \ldots, n-1, 0, \ldots, \mathcal{R}_n^W - 1}^{n \text{ values}}}_{(2^W \div n) \cdot n \text{ values}}}^{2^W \text{ values}}$$

- Generate a uniform random number $x \in [0, 2^W)$ until $x \geq \mathcal{R}_n^W$
- Return $x \bmod n$

Efficiency

We require 2 integer division operations: one for computing $\mathcal{R}_n^W$ and one for computing $x \bmod n$.

The Java Algorithm

The Java Algorithm

The Fast-Dice-Roller Algorithm

The Fast-Dice-Roller Algorithm

The Bitmask Algorithm

The Bitmask Algorithm

Lemire's Algorithm

Multiply-And-Shift

Multiply-And-Shift

The Algorithm

The Algorithm

4

Conclusion

Summary

Summary

End of Talk