

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)







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





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

Where do we need this?



Motivation

What is our goal?



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

Where do we need this?

Shuffling





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

Where do we need this?

- Shuffling
- Complex Graph Generators

TBD

TBD

Motivation

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











- 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











Formal Definition

GOETHE UNIVERSITÄT

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)



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?



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!



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 $x \in [0, n)$



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









Definition (Common Operations)

■ Integer-Division: $x \div y := \lfloor x/y \rfloor$



- Integer-Division: $x \div y \qquad \coloneqq |x/y|$
- Remainder-Operation: $x \mod y := x (x \div y)y$



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



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



■ Integer-Division:
$$x \div y := \lfloor x/y \rfloor$$

■ Remainder-Operation:
$$x \mod y := x - (x \div y)y$$

■ Bit-RightShift:
$$x \gg W := x \div 2^W$$

■ Bit-LeftShift:
$$x \ll W := x \cdot 2^W$$

Bitwise-AND:
$$x \& y$$



- $x \div y := |x/y|$ ■ Integer-Division:
- $x \mod y \coloneqq x (x \div y)y$ ■ Remainder-Operation:
- $x \gg W := x \div 2^W$ Bit-RightShift:
- $x \ll W := x \cdot 2^W$ Bit-LeftShift:
- $x \& y \rightarrow x \mod 2^W \coloneqq x \& (2^W 1)$ ■ Bitwise-AND:







How do we get random numbers?



How do we get random numbers?

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



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 $W \in \{32, 64\}$)



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 $W \in \{32, 64\}$)

 $rand() \mod n$



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 $W \in \{32, 64\}$)

$rand() \mod n$

Does this work?



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 $W \in \{32, 64\}$)

$rand() \mod n$

Does this work?

 \blacksquare Yes, the generated number is in [0, n).



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 $W \in \{32, 64\}$)

$rand() \mod n$

Does this work?

 \blacksquare Yes, the generated number is in [0, n).

Is this efficient?



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 $W \in \{32, 64\}$)

$rand() \mod n$

Does this work?

 \blacksquare Yes, the generated number is in [0, n).

Is this efficient?

■ No, we require one expensive integer division operation.



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 $W \in \{32, 64\}$)

$rand() \mod n$

Does this work?

 \blacksquare 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)?





Preliminaries

The Naive Approach

GOETHE UNIVERSITÄT

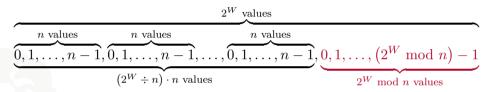
In general, applying $(x \mod n)$ to $[0, 2^W)$ yields

Preliminaries

The Naive Approach



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

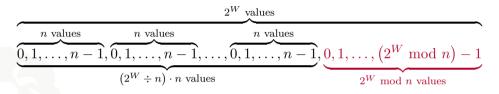


Preliminaries

The Naive Approach



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

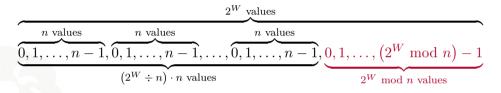


We have a leftover interval that introduces bias.

The Naive Approach



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



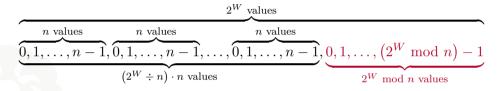
We have a leftover interval that introduces bias.

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

The Naive Approach



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



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 OpenBSD Algorithm





The OpenBSD Algorithm





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









Summary





Summary









End of Talk