

## 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 Bitmask Algorithm

## 3 Lemire's Algorithm

- Multiply-And-Shift
- The Algorithm

## 4 Conclusion



# 1

## Preliminaries





# 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  $x + a$



## Definition (Common Operations)



## Definition (Common Operations)

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



## Definition (Common Operations)

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

## 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:  $x \gg W \quad := x \div 2^W$

## 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:  $x \ll W \quad := x \cdot 2^W$

## 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:  $x \ll W := x \cdot 2^W$
- Bitwise-AND:  $x \& y$



## 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:  $x \ll W := x \cdot 2^W$
- Bitwise-AND:  $x \& y \rightarrow x \bmod 2^W := x \& (2^W - 1)$

## 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:  $x \ll W := x \cdot 2^W$
- Bitwise-AND:  $x \& y \rightarrow x \bmod 2^W := x \& (2^W - 1)$

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

`rand() mod  $n$`



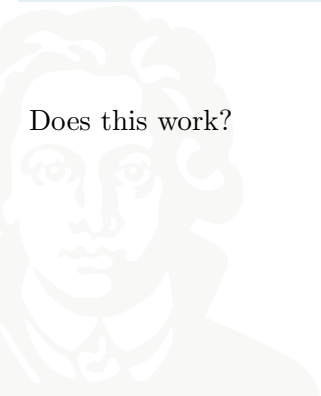
# The Naive Approach

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

`rand() mod  $n$`

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

`rand() mod  $n$`

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

`rand() mod  $n$`

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

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

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

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

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

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

$2^W$  values

# The Naive Approach

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

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

$2^W$  values

We have a **leftover** interval that introduces bias.



# The Naive Approach

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

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

$2^W$  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

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

$2^W$  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

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

$2^W$  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

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

$2^W$  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!

# 2

## 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:

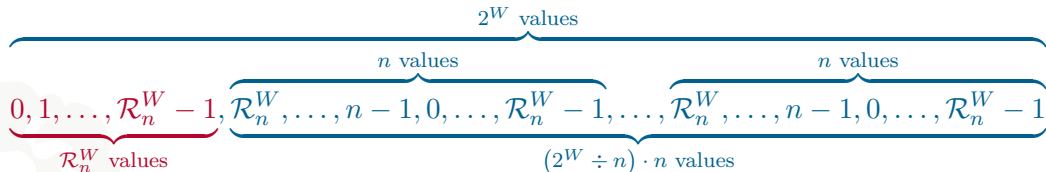
$$\underbrace{0, 1, \dots, \mathcal{R}_n^W - 1}_{\mathcal{R}_n^W \text{ values}}, \underbrace{\mathcal{R}_n^W, \dots, n-1, 0, \dots, \mathcal{R}_n^W - 1}_{n \text{ values}}, \underbrace{\mathcal{R}_n^W, \dots, n-1, 0, \dots, \mathcal{R}_n^W - 1}_{n \text{ values}}$$

$2^W$  values
   
 $(2^W \div n) \cdot n$  values



# The OpenBSD Algorithm

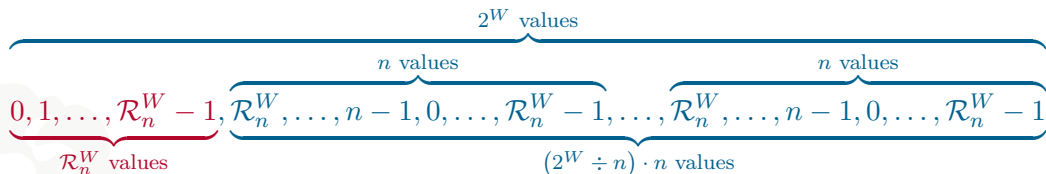
- We shift the **rejection interval** to the left:



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

# The OpenBSD Algorithm

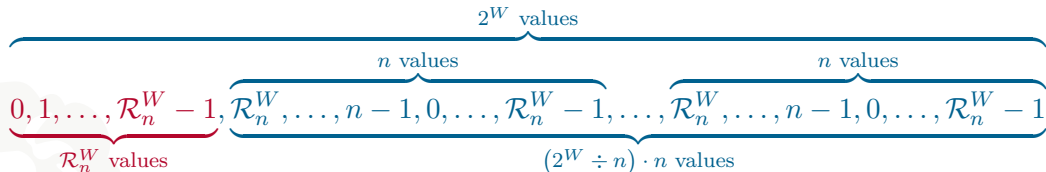
- We shift the **rejection interval** to the left:



- 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:

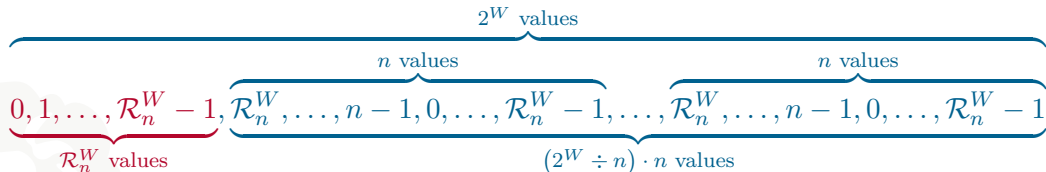


- Generate a uniform random number  $x \in [0, 2^W)$  until  $x \geq \mathcal{R}_n^W$
- Return  $x \bmod n$

## Efficiency

# The OpenBSD Algorithm

- We shift the **rejection interval** to the left:



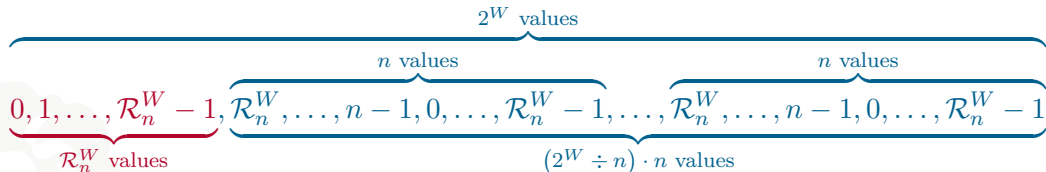
- Generate a uniform random number  $x \in [0, 2^W)$  until  $x \geq \mathcal{R}_n^W$
- Return  $x \bmod n$

## Efficiency

We require 2 integer division operations:

# The OpenBSD Algorithm

- We shift the **rejection interval** to the left:



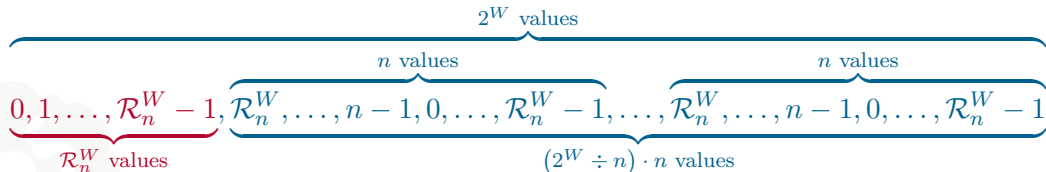
- Generate a uniform random number  $x \in [0, 2^W)$  until  $x \geq \mathcal{R}_n^W$
- Return  $x \bmod n$

## Efficiency

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

# The OpenBSD Algorithm

- We shift the **rejection interval** to the left:



- 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

- Consider  $x - (x \bmod n)$  for  $x \in [0, 2^W)$ :





## The Java Algorithm

- Consider  $x - (x \bmod n)$  for  $x \in [0, 2^W)$ :

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

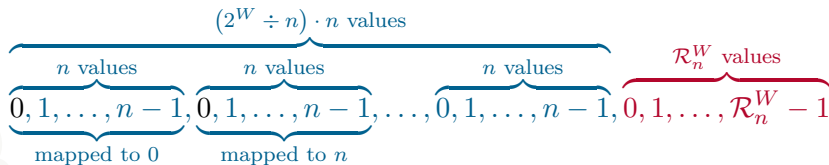
## The Java Algorithm

- Consider  $x - (x \bmod n)$  for  $x \in [0, 2^W)$ :

$$\begin{array}{c}
 \overbrace{\hspace{15em}}^{(2^W \div n) \cdot n \text{ values}} \\
 \underbrace{\overbrace{0, 1, \dots, n-1}^{n \text{ values}}, \overbrace{0, 1, \dots, n-1}^{n \text{ values}}, \dots, \overbrace{0, 1, \dots, n-1}^{n \text{ values}}}_{\text{mapped to 0}}, \overbrace{0, 1, \dots, \mathcal{R}_n^W - 1}^{\mathcal{R}_n^W \text{ values}}
 \end{array}$$

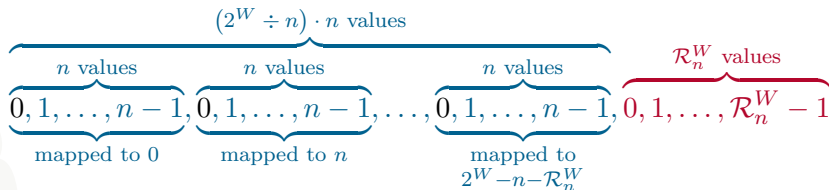
## The Java Algorithm

- Consider  $x - (x \bmod n)$  for  $x \in [0, 2^W)$ :



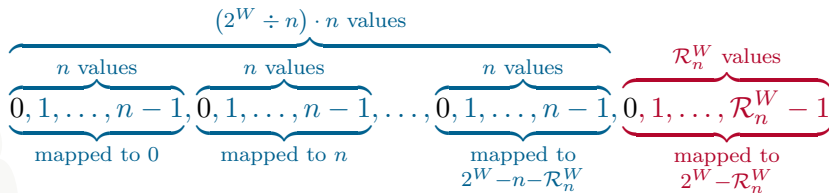
## The Java Algorithm

- Consider  $x - (x \bmod n)$  for  $x \in [0, 2^W)$ :



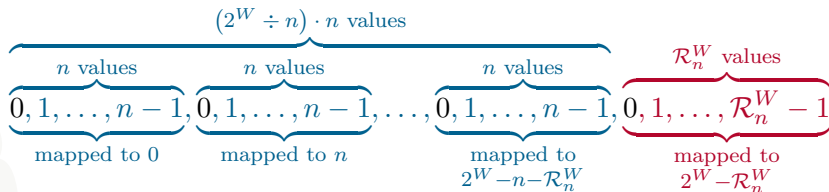
## The Java Algorithm

- Consider  $x - (x \bmod n)$  for  $x \in [0, 2^W)$ :



## The Java Algorithm

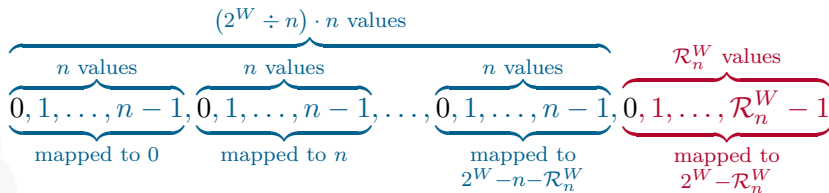
- Consider  $x - (x \bmod n)$  for  $x \in [0, 2^W)$ :



- We map every number to the next-smallest multiple of  $n$

## The Java Algorithm

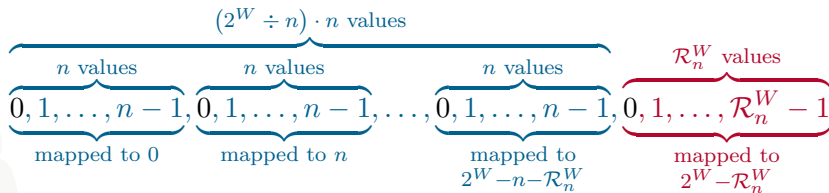
- Consider  $x - (x \bmod n)$  for  $x \in [0, 2^W)$ :



- We map every number to the next-smallest multiple of  $n$
- Only numbers in the **leftover** interval get mapped to  $2^W - \mathcal{R}_n^W > 2^W - n$

## The Java Algorithm

- Consider  $x - (x \bmod n)$  for  $x \in [0, 2^W)$ :

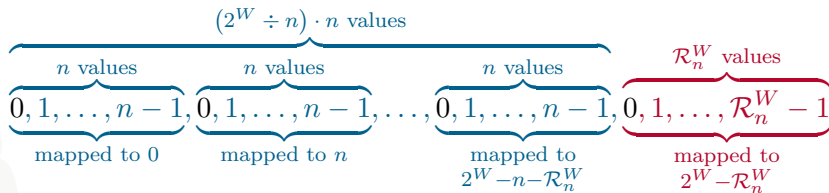


- We map every number to the next-smallest multiple of  $n$
- Only numbers in the **leftover** interval get mapped to  $2^W - \mathcal{R}_n^W > 2^W - n$
- Algorithm:



# The Java Algorithm

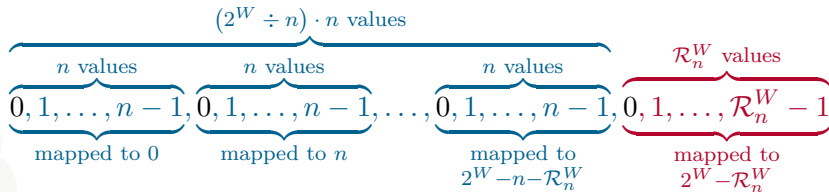
- Consider  $x - (x \bmod n)$  for  $x \in [0, 2^W)$ :



- We map every number to the next-smallest multiple of  $n$
- Only numbers in the **leftover** interval get mapped to  $2^W - \mathcal{R}_n^W > 2^W - n$
- Algorithm:
  - (1) Draw  $x \in [0, 2^W)$  and compute  $r = x \bmod n$

## The Java Algorithm

- Consider  $x - (x \bmod n)$  for  $x \in [0, 2^W)$ :



- We map every number to the next-smallest multiple of  $n$
- Only numbers in the **leftover** interval get mapped to  $2^W - \mathcal{R}_n^W > 2^W - n$
- Algorithm:
  - (1) Draw  $x \in [0, 2^W)$  and compute  $r = x \bmod n$
  - (2) Return  $r$  if  $x - r > 2^W - n$  else goto (1)

# The Java Algorithm

Algorithm:

- (1) Draw  $x \in [0, 2^W)$  and compute  $r = x \bmod n$
- (2) Return  $r$  if  $x - r > 2^W - n$  else goto (1)



# The Java Algorithm

Algorithm:

- (1) Draw  $x \in [0, 2^W)$  and compute  $r = x \bmod n$
- (2) Return  $r$  if  $x - r > 2^W - n$  else goto (1)

## Efficiency

# The Java Algorithm

Algorithm:

- (1) Draw  $x \in [0, 2^W)$  and compute  $r = x \bmod n$
- (2) Return  $r$  if  $x - r > 2^W - n$  else goto (1)

## Efficiency

- At least one integer division operation

# The Java Algorithm

Algorithm:

- (1) Draw  $x \in [0, 2^W)$  and compute  $r = x \bmod n$
- (2) Return  $r$  if  $x - r > 2^W - n$  else goto (1)

## Efficiency

- At least one integer division operation
- Number of integer divisions operations equal to number of rounds

# The Java Algorithm

Algorithm:

- (1) Draw  $x \in [0, 2^W)$  and compute  $r = x \bmod n$
- (2) Return  $r$  if  $x - r > 2^W - n$  else goto (1)

## Efficiency

- At least one integer division operation
- Number of integer divisions operations equal to number of rounds
- Return a number in round if  $x < 2^W - \mathcal{R}_n^W$

# The Java Algorithm

Algorithm:

- (1) Draw  $x \in [0, 2^W)$  and compute  $r = x \bmod n$
- (2) Return  $r$  if  $x - r > 2^W - n$  else goto (1)

## Efficiency

- At least one integer division operation
- Number of integer divisions operations equal to number of rounds
- Return a number in round if  $x < 2^W - \mathcal{R}_n^W$
- Happens with probability  $\frac{2^W - \mathcal{R}_n^W}{2^W} > \frac{1}{2}$



# The Java Algorithm

Algorithm:

- (1) Draw  $x \in [0, 2^W)$  and compute  $r = x \bmod n$
- (2) Return  $r$  if  $x - r > 2^W - n$  else goto (1)

## Efficiency

- At least one integer division operation
- Number of integer divisions operations equal to number of rounds
- Return a number in round if  $x < 2^W - \mathcal{R}_n^W$
- Happens with probability  $\frac{2^W - \mathcal{R}_n^W}{2^W} > \frac{1}{2}$
- Expected number of integer division operations is  $\frac{2^W}{2^W - \mathcal{R}_n^W} < 2$

# The Bitmask Algorithm



# The Bitmask Algorithm



# Lemire's Algorithm

# Multiply-And-Shift



# Multiply-And-Shift



# Lemire's Algorithm

## The Algorithm



# Lemire's Algorithm

## The Algorithm





# 4 Conclusion



## Conclusion

# Summary



	expected number of integer division oper- ations	maximal number of integer division oper- ations	Unbiased?
--	--	---	-----------



## Conclusion

# Summary

	expected number of integer division oper- ations	maximal number of integer division oper- ations	Unbiased?
Modulo Reduction	1	1	<b>X</b>



## Conclusion

# Summary

	expected number of integer division oper- ations	maximal number of integer division oper- ations	Unbiased?
Modulo Reduction	1	1	<b>X</b>
Multiply-and-Shift	0	0	<b>X</b>

	expected number of integer division oper- ations	maximal number of integer division oper- ations	Unbiased?
Modulo Reduction	1	1	✗
Multiply-and-Shift	0	0	✗
OpenBSD	2	2	✓

## Conclusion

# Summary

	expected number of integer division oper- ations	maximal number of integer division oper- ations	Unbiased?
Modulo Reduction	1	1	✗
Multiply-and-Shift	0	0	✗
OpenBSD	2	2	✓
Java	$\frac{2^W}{2^W - (2^W \bmod n)}$	$\infty$	✓

## Conclusion

# Summary

	expected number of integer division oper- ations	maximal number of integer division oper- ations	Unbiased?
Modulo Reduction	1	1	✗
Multiply-and-Shift	0	0	✗
OpenBSD	2	2	✓
Java	$\frac{2^W}{2^W - (2^W \bmod n)}$	$\infty$	✓
Bitmask	0	0	✓



## Conclusion

# Summary

	expected number of integer division oper- ations	maximal number of integer division oper- ations	Unbiased?
Modulo Reduction	1	1	✗
Multiply-and-Shift	0	0	✗
OpenBSD	2	2	✓
Java	$\frac{2^W}{2^W - (2^W \bmod n)}$	$\infty$	✓
Bitmask	0	0	✓
Lemire	$\frac{n}{2^W}$	1	✓

## Conclusion

# Summary

	expected number of integer division oper- ations	maximal number of integer division oper- ations	Unbiased?
Modulo Reduction	1	1	✗
Multiply-and-Shift	0	0	✗
OpenBSD	2	2	✓
Java	$\frac{2^W}{2^W - (2^W \bmod n)}$	$\infty$	✓
Bitmask	0	0	✓
Lemire	$\frac{n}{2^W}$	1	✓

**End of Talk**

