Monte Carlo 3

Earlier lectures

- Monte Carlo I: integration using randomness
- Monte Carlo II: variance reduction
- Basic signal processing and sampling

Today

- Discrepancy and Quasi-Monte Carlo
- Low-discrepancy constructions
- Efficient implementation
Equi-Areal Disk Sampling

\[ \xi_i \in [0, 1)^2 \]

\[ \theta = 2\pi \xi_1 \]

\[ r = \sqrt{\xi_2} \]
Equi-Areal Disk Sampling

\[ \xi_i \in [0, 1)^2 \]

\[ \theta = 2\pi \xi_1 \]

\[ r = \sqrt{\xi_2} \]
Uniform Hemisphere Sampling

\[(\xi_1, \xi_2) \rightarrow (\sqrt{1 - \xi_1^2} \cos(2\pi \xi_2), \sqrt{1 - \xi_1^2} \sin(2\pi \xi_2), \sqrt{1 - \xi_1^2})\]

\[p(\omega) = \frac{1}{2\pi}\]
Uniform Hemisphere Sampling

\[ p(\omega) = \frac{1}{2\pi} \]
Cosine-Weighted Hemisphere

\[ \theta = 2\pi \xi_1 \]
\[ r = \sqrt{\xi_2} \]
\[ (x, y, z) = (r \sin \theta, r \cos \theta, \sqrt{1 - r^2}) \]

\[ p(\omega) = \frac{\cos \theta}{\pi} \]
Four 2D Point Sets
Point Set Evaluation: Discrepancy

\[ \Delta(x, y) = \frac{n(x, y)}{N} - xy \]

\[ A = xy \]

\( n(x, y) \) number of samples in \( A \)

\[ D_N = \max_{x,y} |\Delta(x, y)| \]
Discrepancy

Larscher-Pillichshammer  Stratified  Random
Discrepancy

Larscher-Pillichshammer  Stratified  Random
Discrepancy

Larscher-Pillichshammer 0.041
Stratified 0.081
Random 0.148
Low-Discrepancy Definition

An (infinite) sequence of $n$ samples in dimension $d$ is low discrepancy if:

$$D_n = O \left( \frac{(\log n)^d}{n} \right)$$

A (finite) set of $n$ samples in dimension $d$ is low discrepancy if:

$$D_n = O \left( \frac{(\log n)^{d-1}}{n} \right)$$
Koksma-Hlawka inequality:

\[
\left| \frac{1}{N} \sum_{i=0}^{N-1} f(X_i) - \int f(x) \, dx \right| \leq V(f) D_N
\]

\[
V(f) = \int \left| \frac{\delta f}{\delta x} \right| \, dx
\]
although error bounded as: \[ |e| \leq V(f)D_N \]

not a tight bound!

even worse, \( V(f) \) is sometimes unbounded

further, can use this inequality to show that QMC convergence is:

\[ \sim \frac{(\log N)^d}{N} \]

so, \( d \) must be small and \( N \) large to beat MC
Low Discrepancy vs. Monkey Eye Eye

Monkey Eye Cone Distribution

Low Discrepancy Point Set
Measuring Point Set Quality

Some problems with low discrepancy

- Anisotropic: rotating the points changes discrepancy
- Not shift-invariant: similarly for translation

In general, can have low discrepancy yet still have points clumped together
Low-Discrepancy Sequences
The Radical Inverse

Consider the digits of a number \( n \), expressed in base \( b \)

\[
n = \sum_{i=1}^{\infty} d_i b^{(i-1)}
\]

e.g. for \( n = 6 \) in base 2, \( n=110_2 \), and

\[
d_1 = 0, d_2 = 1, d_3 = 1, d_i = 0
\]

The radical inverse mirrors the digits around the decimal:

\[
\Phi_2(6) = 0.011_2 = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.375
\]

\[
\Phi_b(n) = \sum_{i=1}^{\infty} d_i b^{-i}
\]
1D Low Discrepancy: van der Corput

<table>
<thead>
<tr>
<th>n</th>
<th>$\Phi_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
</tr>
<tr>
<td>5</td>
<td>0.625</td>
</tr>
<tr>
<td>6</td>
<td>0.375</td>
</tr>
<tr>
<td>7</td>
<td>0.875</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### 1D Low Discrepancy: van der Corput

<table>
<thead>
<tr>
<th>n</th>
<th>$\Phi_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
</tr>
<tr>
<td>5</td>
<td>0.625</td>
</tr>
<tr>
<td>6</td>
<td>0.375</td>
</tr>
<tr>
<td>7</td>
<td>0.875</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
# 1D Low Discrepancy: van der Corput

<table>
<thead>
<tr>
<th>n</th>
<th>$\Phi_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
</tr>
<tr>
<td>5</td>
<td>0.625</td>
</tr>
<tr>
<td>6</td>
<td>0.375</td>
</tr>
<tr>
<td>7</td>
<td>0.875</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### 1D Low Discrepancy: van der Corput

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \Phi_2(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
</tr>
<tr>
<td>5</td>
<td>0.625</td>
</tr>
<tr>
<td>6</td>
<td>0.375</td>
</tr>
<tr>
<td>7</td>
<td>0.875</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

![Diagram showing points on a number line]
## 1D Low Discrepancy: van der Corput

<table>
<thead>
<tr>
<th>n</th>
<th>$\Phi_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
</tr>
<tr>
<td>5</td>
<td>0.625</td>
</tr>
<tr>
<td>6</td>
<td>0.375</td>
</tr>
<tr>
<td>7</td>
<td>0.875</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The table and diagrams illustrate the van der Corput sequence for $\Phi_2(n)$. Each value represents the corresponding point in the sequence. The diagrams show the progression of points as $n$ increases.
Efficient Base 2 Radical Inverse

Recall the definition of the digits of a number in base $b$:

$$n = \sum_{i=1}^{\infty} d_i b^{(i-1)}$$

Thanks to computer binary representation, these digits are easily extracted:

```c
int DigitBase2(uint32_t n, int i) {
    return (n & (1 << (i-1))) ? 1 : 0;
}
```

In arbitrary bases, we’re not so lucky:

```c
int DigitBaseB(uint32_t n, int i, int b) {
    n /= ipow(b, i-1);
    return n % b;
}
```
Efficient Base 2 Radical Inverse

Assume a fixed number of bits (say 32):

\[ \Phi_b(n) = \sum_{i=1}^{32} d_i b^{-i} \]

We have the sum:

\[ d_1 2^{-1} + d_2 2^{-2} + \cdots + d_{32} 2^{-32} \]

Pull out a factor of \( 2^{-32} \):

\[ 2^{-32} \left( d_1 2^{31} + d_2 2^{30} + \cdots + d_{32} \right) \]

Can also express in terms of bit shifts:

\[ 2^{-32} \left( (d_1 \ll 31) + (d_2 \ll 30) + \cdots + d_{32} \right) \]
Efficient Base 2 Radical Inverse

\[ 2^{-32}((d_1 << 31) + (d_2 << 30) + \cdots + d_{32}) \]

We have the digits already in the bits of \( n \)

\[ n = \sum_{i=1}^{\infty} d_i b^{(i-1)} \]

32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

So

- Reverse the bits
- Multiply by \( 2^{-32} \)
Reversing Bits in Parallel

```c
uint32_t ReverseBits(uint32_t n) {

    n = (n << 16) | (n >> 16);

    n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);

    n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);

    n = ((n & 0x33333333) << 2) | ((n & 0xc0c0c0c0) >> 2);

    n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
    return n;
}
```
uint32_t ReverseBits(uint32_t n) {
    n = (n << 16) | (n >> 16);

    n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);

    n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);

    n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);

    n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
    return n;
}
Reversing Bits in Parallel

```c
uint32_t ReverseBits(uint32_t n) {

    n = (n << 16) | (n >> 16);  
    n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);  
    n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);  
    n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);  
    n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);  
    return n;
}
```
uint32_t ReverseBits(uint32_t n) {
    n = (n << 16) | (n >> 16);
    n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);
    n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);
    n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);
    n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
    return n;
}
Reversing Bits in Parallel

```c
uint32_t ReverseBits(uint32_t n) {
    n = (n << 16) | (n >> 16);
    n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);
    n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);
    n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);
    n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
    return n;
}
```
Reversing Bits in Parallel

```c
uint32_t ReverseBits(uint32_t n) {
    n = (n << 16) | (n >> 16);
    n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);
    n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);
    n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);
    n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
    return n;
}
```
float RadicalInverse2(uint32_t v) {
    v = ReverseBits(v);
    const float Inv2To32 = 1.f / (1ull << 32);
    return v * Inv2To32;
}

uint32_t ReverseBits(uint32_t n) {
    n = (n << 16) | (n >> 16);
    n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);
    n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);
    n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);
    n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
    return n;
}
The Halton Sequence

Low discrepancy sequence

- Arbitrary number of dimensions
- Arbitrary number of points

\[ (\Phi_{b_1}(n), \Phi_{b_2}(n), \Phi_{b_3}(n), \ldots) \]

where the bases for each of the dimensions are relatively prime
The Halton Sequence

One caution: 2D projections of higher bases may not be great

- The overall pattern in remains low-discrepancy over all dimensions, though

$\Phi_{23}(n), \Phi_{29}(n)$
The Hammersley Point Set

If the number of points $N$ is known in advance, set one dimension to $n/N$

$$(n/N, \Phi_{b_1}(n), \Phi_{b_2}(n), \ldots)$$

Slightly lower discrepancy than Halton
Efficient Radical Inverse, base $b \neq 2$

Integer division by a constant can be done with multiplies and shifts

\[
q = \left\lfloor \frac{2^{32} + 2 \frac{n}{3}}{2^{32}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{2n}{3 \times 2^{32}} \right\rfloor
\]

\[
= \left\lfloor \frac{n}{3} \right\rfloor \text{ if } n < 2^{32}
\]

```c
unsigned int div3(unsigned int v) {
    const int64_t magic = 0x55555556;
    return (magic * v) >> 32;
}
```
Integer division by a constant can be done with multiplies and shifts

```c
int div(int v, int d) {
    return v / d;
}
// ...
div(v, 3);
```

```c
unsigned int div3(unsigned int v) {
    const int64_t magic = 0x555555556;
    return (magic * v) >> 32;
}
```

```c
# ...
iidivl %ecx
# ...
```

```c
# ...
imulq $1431655766, %rax, %rax; # imm = 0x55555556
shrq$32, %rax
# ...
```

~7x faster
Generator Matrices

Given a base $b$ and a matrix $C$, define:

$$c(n) = (b^{-1}, b^{-2}, \ldots, b^{-m}) C \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_m \end{pmatrix}$$

- where $d_i$ are the base-$b$ digits of $n$
- and arithmetic is done over the ring $\mathbb{Z}_b$
- For our purposes, just do everything “mod $b$”

This generates a set of $b^m$ points

 Appropriately-chosen $C$ matrices generate various low-discrepancy point sets
Generator Matrices

We’ll focus only on $b=2$, which allows particularly efficient implementation

$$c(n) = (2^{-1}, 2^{-2}, \ldots, 2^{-m}) C \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_m \end{pmatrix}$$
Sobol’ Point Sets

Sobol’ first showed how to find generator matrices for LD point sets in base 2

- Can scale low-discrepancy samples in 1000s of dimensions
32 2D Sobol’ Points
Elementary Intervals (1x64)
Elementary Intervals (2x32)
Elementary Intervals (4x8)
Elementary Intervals (8x4)
Elementary Intervals (32x1)
Uniform Random Samples, n=16
MSE 0.00227214
Stratified Samples, n=16
MSE 0.00208727
Low Discrepancy Samples, $n=16$
MSE 0.00201494
Uniform Random Samples, n=16
MSE 0.000421827
Stratified Samples, n=16
MSE 0.000253177
Low Discrepancy Samples, n=16
MSE 0.000164011
In addition to satisfying general stratification properties, power-of-two length subsequences are well-distributed with respect to each other.
(0,2)-sequences

In addition to satisfying general stratification properties, power-of-two length subsequences are well-distributed with respect to each other.
(0,2)-sequences

In addition to satisfying general stratification properties, power-of-two length subsequences are well-distributed with respect to each other.
(0,2)-sequences

In addition to satisfying general stratification properties, power-of-two length subsequences are well-distributed with respect to each other.
Pixel * Light Sampling
Pixel * Light Sampling
Pixel * Light Sampling
Maximized Minimum Distance

Grünschloß and Keller: exhaustive search over generator matrices

Still stratified over elementary intervals

\[ C_6 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \]
Low Discrepancy + Blue Noise

Ahmed et al. 2016
Sobol’ vs Low Discrepancy Blue Noise

Sobol’

LDBN

Power Spectra
(Live demo)
Sampling Motion Blur + Defocus

Random
MSE 3.42e-4

(0,2)-nets
MSE 2.07e-4

Halton
MSE 3.02e-4

Sobol’
MSE 1.90e-4
Efficient Generator Matrix Implementation

How do we do this multiplication efficiently?

■ Consider e.g. m=32 for regular 32-bit integers...

\[
\begin{pmatrix}
  c_{1,1} & c_{1,2} & \cdots & c_{1,m} \\
  c_{2,1} & \ddots & & \vdots \\
  \vdots & & \ddots & \vdots \\
  c_{m,1} & c_{m,2} & \cdots & c_{m,m}
\end{pmatrix}
\begin{pmatrix}
  d_1 \\
  d_2 \\
  \vdots \\
  d_n
\end{pmatrix}
\]

\[
= d_1 \begin{pmatrix}
  c_{1,1} \\
  c_{2,1} \\
  \vdots \\
  c_{m,1}
\end{pmatrix} + d_2 \begin{pmatrix}
  c_{1,2} \\
  c_{2,2} \\
  \vdots \\
  c_{m,2}
\end{pmatrix} + \cdots + d_m \begin{pmatrix}
  c_{1,m} \\
  c_{2,m} \\
  \vdots \\
  c_{m,m}
\end{pmatrix}
\]
Efficient Generator Matrix Implementation

Recall that we’re doing all of this arithmetic mod 2

- All values are either 0 or 1...

\[
d_1 \begin{pmatrix}
  c_{1,1} \\
  c_{2,1} \\
  \vdots \\
  c_{m,1}
\end{pmatrix} + d_2 \begin{pmatrix}
  c_{1,2} \\
  c_{2,2} \\
  \vdots \\
  c_{m,2}
\end{pmatrix} + \cdots d_m \begin{pmatrix}
  c_{1,m} \\
  c_{2,m} \\
  \vdots \\
  c_{m,m}
\end{pmatrix}
\]

- What logical ops are + and *, mod 2, equivalent to?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Efficient Generator Matrix Implementation

Add (XOR) columns if corresponding digit is one.

```c
// C[] holds columns of the generator matrix
float GenMatA(uint32_t n, const uint32_t C[32]) {
    uint32_t bits = 0;
    for (int i = 0; i < 32; ++i) {
        if (n & (1 << i) != 0)
            bits ^= C[i];  // 32 ADDs, mod 2
    }
    const float Inv2To32 = 1.f / (1ull << 32);
    return ReverseBits(bits) * Inv2To32;
}
```
**Efficient Generator Matrix Implementation**

**Better: stop when n=0**

```c
float GenMatB(uint32_t n, const uint32_t C[32]) {
    uint32_t bits = 0, i = 0;
    while (n != 0) {
        if (n & 1)
            bits ^= C[i];
        n >>= 1;
        ++i;
    }
    const float Inv2To32 = 1.f / (1ull << 32);
    return ReverseBits(bits) * Inv2To32;
}
```
Efficient Generator Matrix Implementation

Avoid the bit reverse by reversing the columns of the matrix

```c
float GenMatB(uint32_t n, const uint32_t C[32]) {
    uint32_t bits = 0, i = 31;
    while (n != 0) {
        if (n & 1)
            bits ^= C[i];
        n >>= 1;
        --i;
    }
    const float Inv2To32 = 1.f / (1ull << 32);
    return bits * Inv2To32;
}
```
Even Faster Evaluation with Grey Codes

Grey codes: permutation of integers within blocks of size $2^n$ such that adjacent values only differ in a single bit

Very simple to compute:

```c
int GreyCode(int v) {
    return v ^ (v >> 1);
}
```

<table>
<thead>
<tr>
<th>n</th>
<th>binary</th>
<th>Grey code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Incremental Evaluation with Grey Codes

If \( g() \) is the Grey code function, and we have \( c(g(n-1)) \), what is \( c(g(n)) \)?

- We know that \( g(n-1) \) and \( g(n) \) differ in a single bit (call it \( b \))

\[
c(g(n)) = c(g(n - 1)) + \begin{pmatrix} c_{b,1} \\ c_{b,2} \\ \vdots \\ c_{b,m} \end{pmatrix}
\]

Or

\[
c(g(n)) = c(g(n - 1)) - \begin{pmatrix} c_{b,1} \\ c_{b,2} \\ \vdots \\ c_{b,m} \end{pmatrix}
\]
Good News In Base 2

Both + and - are equivalent to XOR in $\mathbb{Z}_2$

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>-</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$$c(g(n)) = c(g(n - 1)) \pm \begin{pmatrix} C_{b,1} \\ C_{b,2} \\ \vdots \\ C_{b,m} \end{pmatrix}$$

```plaintext
next = prev ^ C[changedBit]
```
### Even Faster Evaluation with Grey Codes

#### Which Bit Changed?

<table>
<thead>
<tr>
<th>n</th>
<th>binary</th>
<th>Grey code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- # of trailing 0s in the binary representation of n

---
Even Faster Evaluation with Grey Codes

Final implementation is super efficient

```c
uint32_t CIncremental(uint32_t n,
    uint32_t prev, const uint32_t C[32]) {
    int changedBit = CountTrailingZeros(n);
    return prev ^ C[31-changedBit];
}
```

```assembly
bsfl  %edi, %eax
xorl  $31, %eax
xorl  (%rdx,%rax,4), %esi
```

Most ISAs have an instruction to count trailing zeros