600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

**Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.*

Outline

- Integers
  - Modular arithmetic
  - Bases
  - Representation
  - Bit-wise operators

Arithmetic

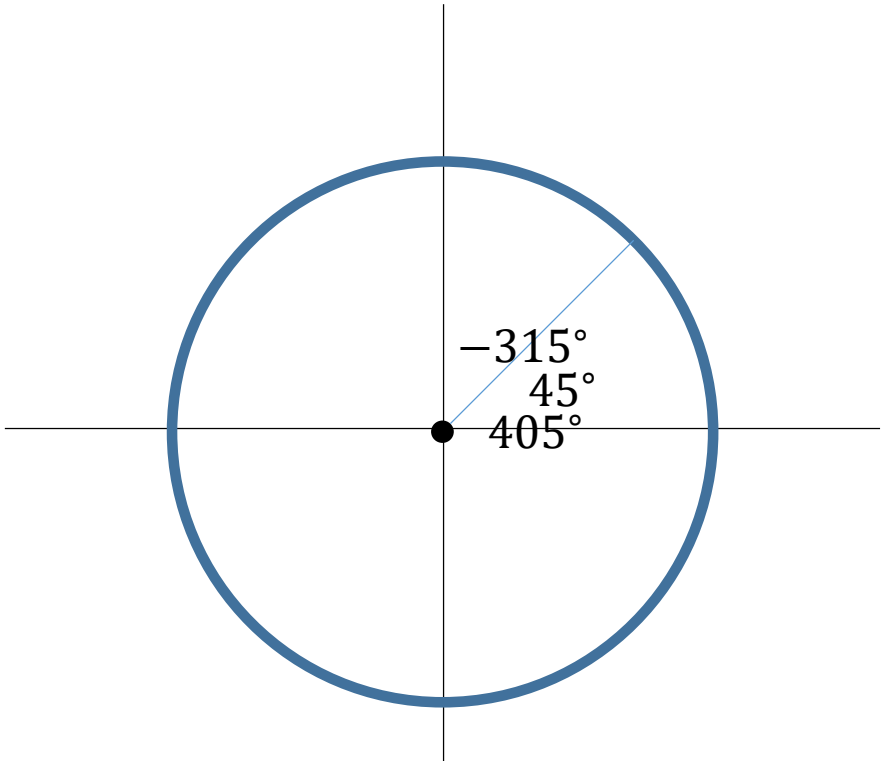
- The integers is a set of numbers
- It has an addition operator, $+$, that takes a pair of integers and returns an integer
  - It contains a zero element, 0, with the property that adding zero to any integer gives back that integer:
    $$a + 0 = a$$
  - Every integer a has an inverse $-a$ such that the sum of the two is zero:
    $$a + (-a) = a - a = 0$$

Modular arithmetic

- Given a positive integer, M, we say that two integers a and b <u>are equivalent modulo M</u>, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

  - Degrees in a circle (mod $360°$)
  - Hours on a clock (mod 12)



Modular arithmetic

- Given a positive integer, M, we say that two integers a and b <u>are equivalent modulo M</u>, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can represent integers mod M using values in the range $[0, M)$
  - While an integer is bigger than or equal to M, repeatedly subtract M
  - While an integer is less than zero, repeatedly add M

Modular arithmetic

- Given a positive integer, M, we say that two integers a and b <u>are equivalent modulo M</u>, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can represent integers mod M using values in the range $[0, M)$
- Or, we can represent integers mod M using the range $[-10, M - 10)$
- Or, we can represent integers mod M using the range $\left[-\frac{M}{2}, \frac{M}{2}\right)$

*The distinction between the integers mod M and their representation continues to confuse students in my graduate courses

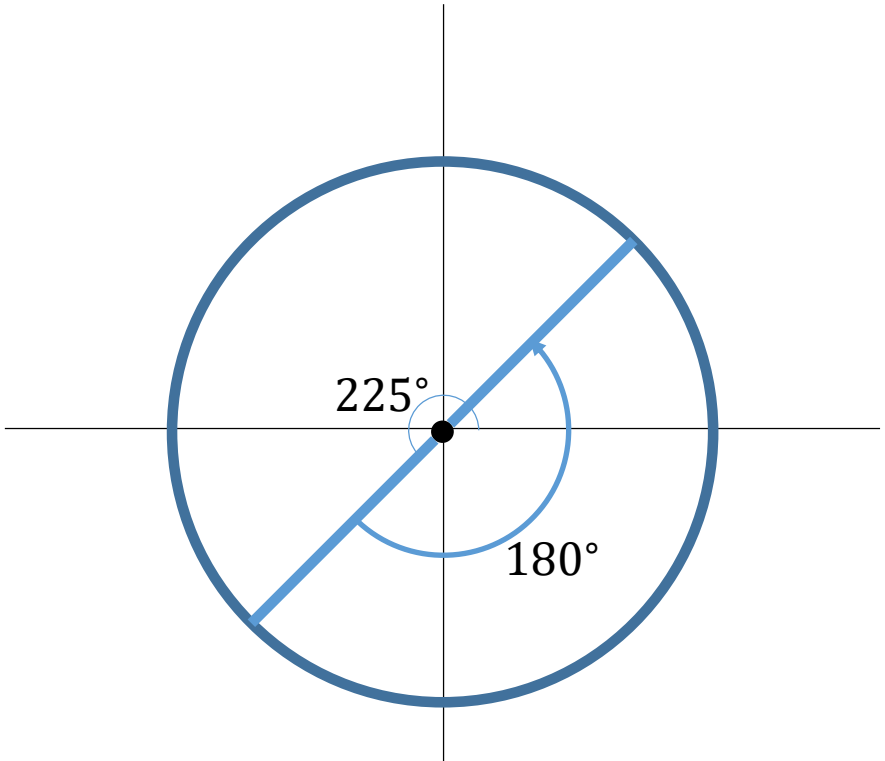
Modular arithmetic

- Given a positive integer, M, we say that two integers a and b <u>are equivalent modulo M</u>, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M:

$$225^\circ + 180^\circ = 405^\circ$$



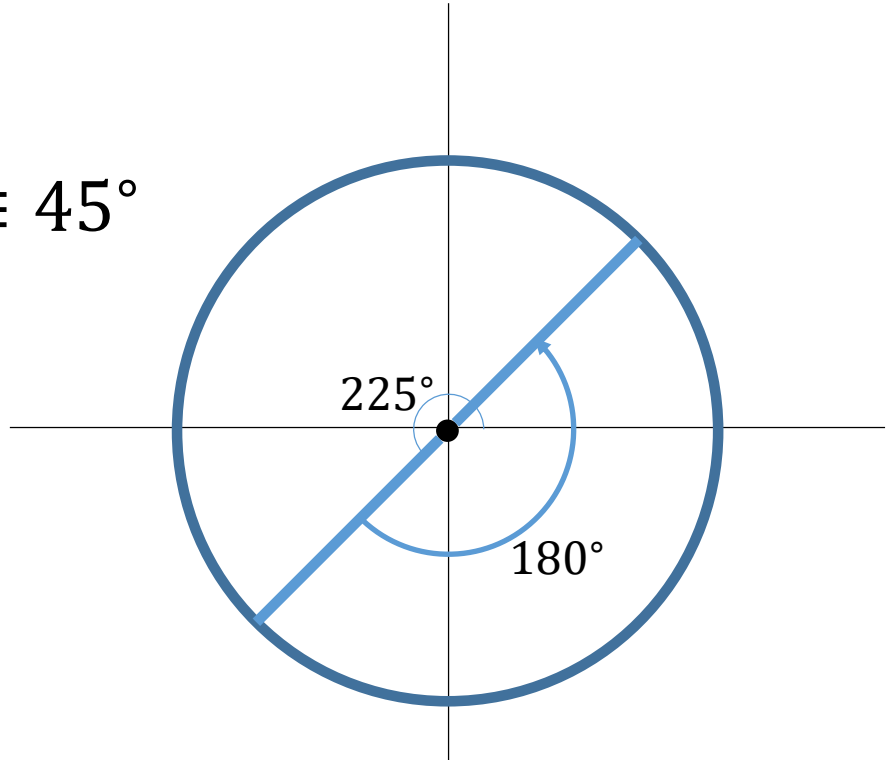
Modular arithmetic

- Given a positive integer, M, we say that two integers a and b <u>are equivalent modulo M</u>, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M:

$$225^\circ + 180^\circ = 405^\circ \equiv 45^\circ$$



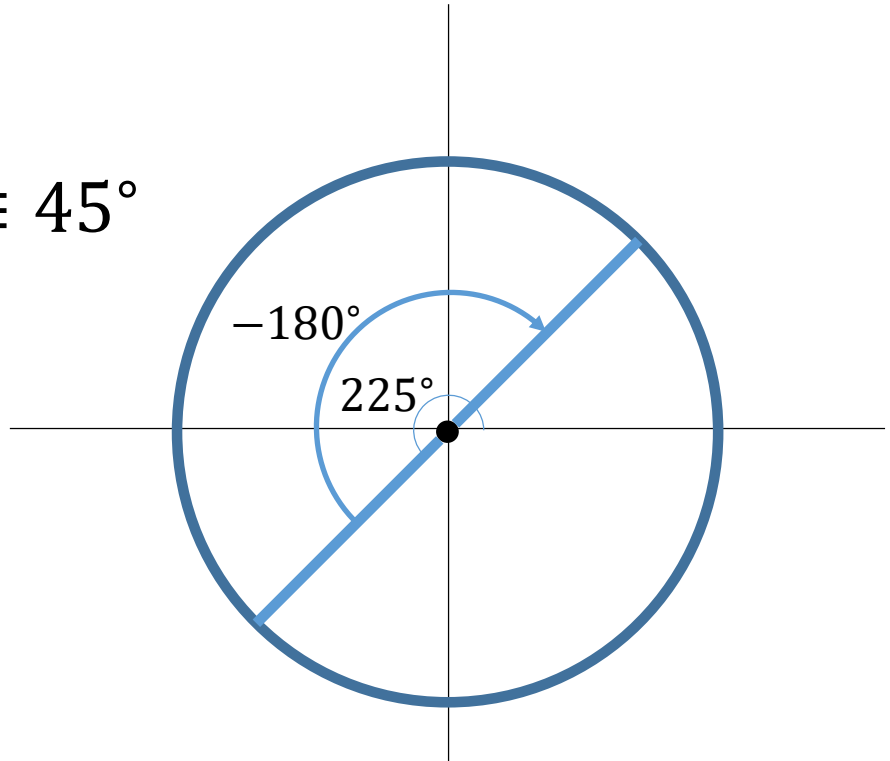
Modular arithmetic

- Given a positive integer, M, we say that two integers a and b <u>are equivalent modulo M</u>, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M:

$$225^\circ - 180^\circ \equiv 225^\circ + 180^\circ = 405^\circ \equiv 45^\circ$$



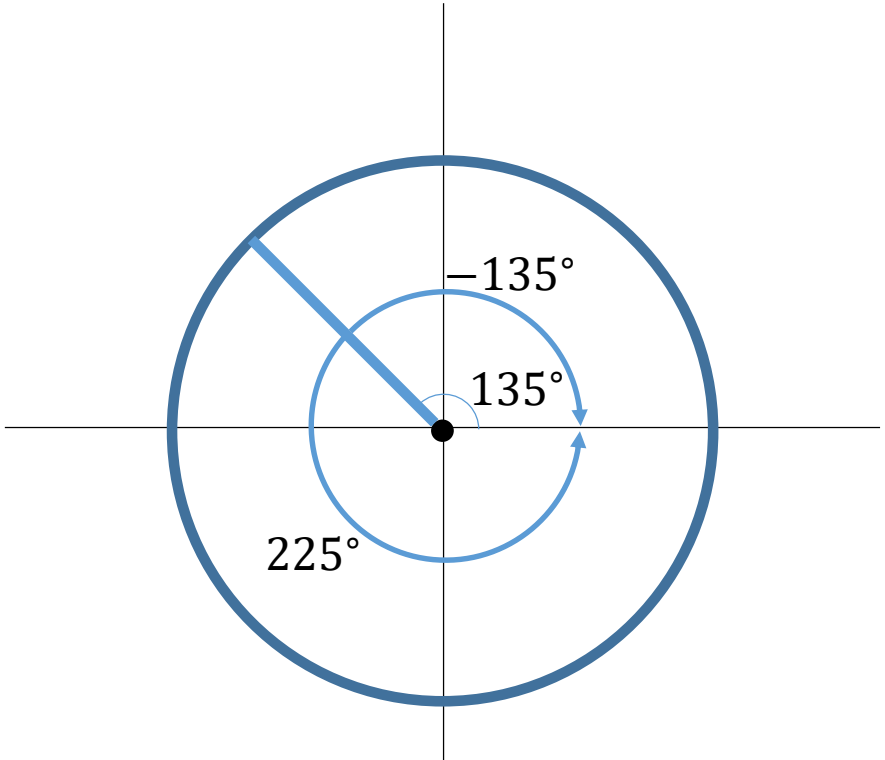
Modular arithmetic

- Given a positive integer, M, we say that two integers a and b <u>are equivalent modulo M</u>, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M

- For any integer a, the negative of a modulo M can be represented by $M - a$



Modular arithmetic

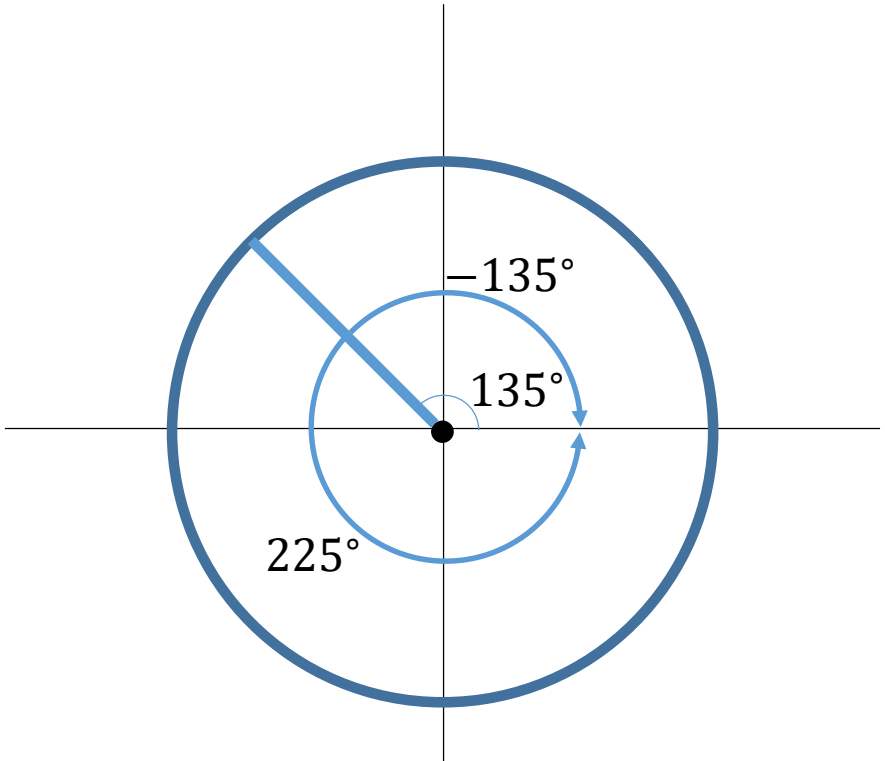
- Given a positive integer, M, we say that two integers a and b <u>are equivalent modulo M</u>, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M

- For any integer a, the negative of a modulo M can be represented by $M - a$:

$$a + (M - a) = (a - a) + M$$



Modular arithmetic

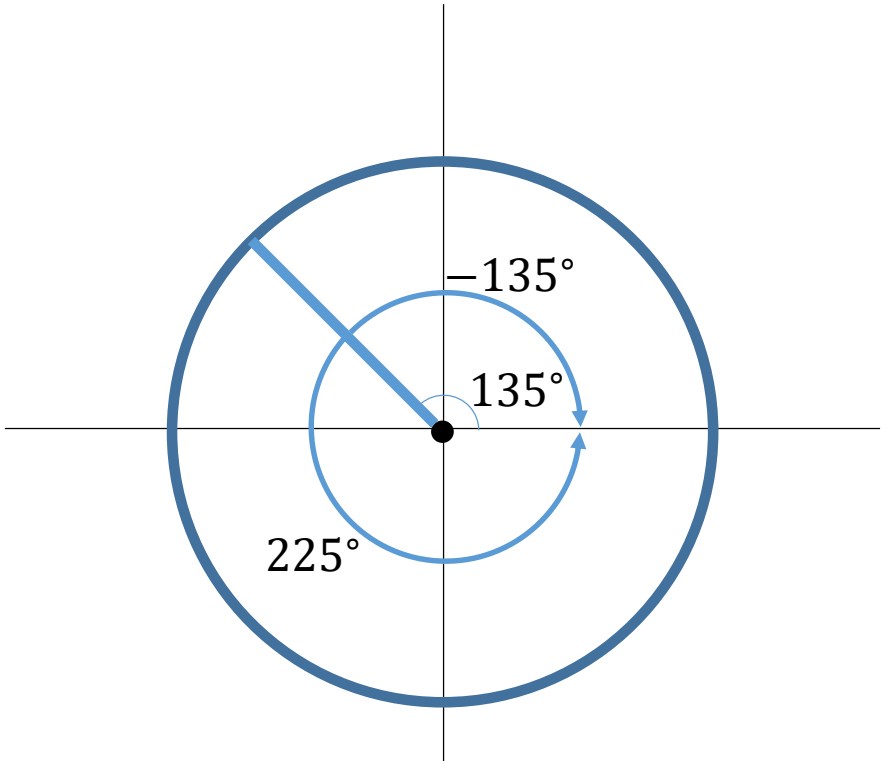
- Given a positive integer, M, we say that two integers a and b <u>are equivalent modulo M</u>, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M

- For any integer a, the negative of a modulo M can be represented by $M - a$:

$$a + (M - a) = (a - a) + M = M$$



Modular arithmetic

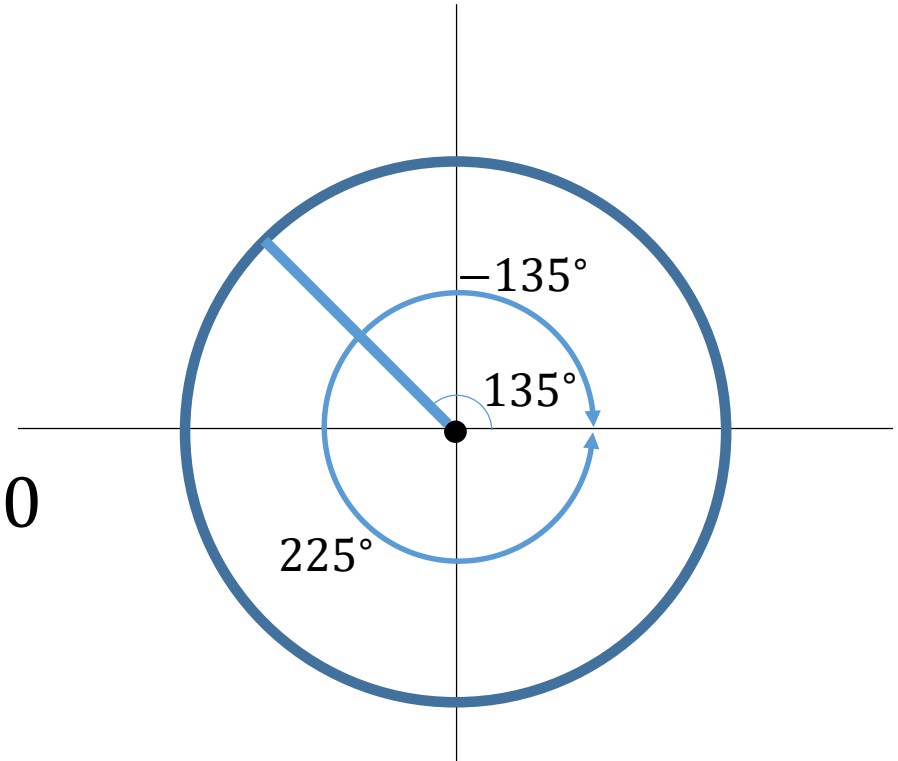
- Given a positive integer, M, we say that two integers a and b <u>are equivalent modulo M</u>, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M

- For any integer a, the negative of a modulo M can be represented by $M - a$:

$$a + (M - a) = (a - a) + M = M \equiv 0$$



Bases (decimal)

- When we write out an integer in decimal notation, we are representing it as a sum of “one”s, “ten”s, “hundred”s, etc.

$$\begin{aligned} 365 &= 3 \times 100 + 6 \times 10 + 5 \times 1 \\ &= 3 \times 10^2 + 6 \times 10^1 + 5 \times 10^0 \end{aligned}$$

- This unique because each digit is in the range 0 to 9, written $[0,10)$
- Since 10^1, 10^2, ... are divisible by 10, we can check if a number is divisible by 10 by checking if the coefficient in the one’s place is zero
  - Since 10^1, 10^2, ... are divisible by 2, we can check if a number is divisible by 2 by checking if the coefficient in the one’s place is divisible by 2
  - Since 10^1, 10^2, ... are divisible by 5, we can ...

Bases (decimal)

- We add two numbers by adding the digits from smallest to largest
  - If the sum of digits falls outside the range $[0,10)$ we <u>carry</u>

$$\begin{array}{rccc} & 3 & 6 & 5 \\ + & 6 & 7 & 3 \\ \hline \end{array}$$

Bases (decimal)

- We add two numbers by adding the digits from smallest to largest
  - If the sum of digits falls outside the range $[0,10)$ we <u>carry</u>

$$\begin{array}{rccc} & 3 & 6 & 5 \\ + & 6 & 7 & 3 \\ \hline & & & 8 \end{array}$$

Bases (decimal)

- We add two numbers by adding the digits from smallest to largest
  - If the sum of digits falls outside the range $[0,10)$ we <u>carry</u>

$$
\begin{array}{rccc}
 & 1 & & \\
 & 3 & 6 & 5 \\
+ & 6 & 7 & 3 \\
\hline
 & & 3 & 8
\end{array}
$$

Bases (decimal)

- We add two numbers by adding the digits from smallest to largest
  - If the sum of digits falls outside the range $[0,10)$ we <u>carry</u>

$$
\begin{array}{cccc}
1 & 1 & & \\
 & 3 & 6 & 5 \\
+ & 6 & 7 & 3 \\
\hline
 & 0 & 3 & 8
\end{array}
$$

Bases (decimal)

- We add two numbers by adding the digits from smallest to largest
  - If the sum of digits falls outside the range $[0,10)$ we <u>carry</u>

$$
\begin{array}{cccc}
1 & 1 & & \\
 & 3 & 6 & 5 \\
+ & 6 & 7 & 3 \\
\hline
1 & 0 & 3 & 8
\end{array}
$$

Bases (decimal)

Q: If we use three digits, how many numbers can we represent?

A: $1000 = 10^3$ (including zero)

<u>Note</u>:

- The sum of two numbers represented using three digits may require four digits to store:

$$\begin{array}{rrrr} & 3 & 6 & 5 \\ + & 6 & 7 & 3 \\ \hline 1 & 0 & 3 & 8 \end{array}$$

Bases (decimal)

Q: If we use three digits, how many numbers can we represent?

A: $1000 = 10^3$ (including zero)

<u>Note</u>:

- The sum of two numbers represented using three digits may require four digits to store:

$$\begin{array}{rccc} & 3 & 6 & 5 \\ + & 6 & 7 & 3 \\ \hline & 0 & 3 & 8 \end{array}$$

- If we only use three digits, we lose the leading digit to <u>overflow</u>
- This is the same as the number mod 10^3

Bases (general)

- We can use other bases to represent numbers, writing
$$(s_3 s_2 s_1 s_0)_b = s_3 \times b^3 + s_2 \times b^2 + s_1 \times b^1 + s_0 \times b^0$$
where b is the base and s_0, s_1, s_2, s_3 are digits in the range $[0, b)$

- Since b^1, b^2, … are divisible by b, we can check if a number is divisible by b by checking if the coefficient in the one's place is zero
  - If a divides b, we can check if a divides the number by checking if the coefficient in the one's place is divisible by a

"Base eight is just like base ten really, if you're missing two fingers."
-- Tom Lehrer

Bases (general)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum falls outside the range $[0, b)$:

$$
\begin{array}{rcccc}
 & & (3 & 6 & 5)_9 \\
+ & & (6 & 7 & 3)_9 \\
\hline
 & (& & &)_9
\end{array}
$$

Bases (general)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum falls outside the range $[0, b)$:

$$\begin{array}{cccc} & (3 & 6 & 5)_9 \\ + & (6 & 7 & 3)_9 \\ \hline (& & & 8)_9 \end{array}$$

Bases (general)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum falls outside the range $[0, b)$:

$$
\begin{array}{cccc}
 & 1 & & \\
 & (3 & 6 & 5)_9 \\
+ & (6 & 7 & 3)_9 \\
\hline
(& & 4 & 8)_9
\end{array}
$$

Bases (general)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum falls outside the range $[0, b)$:

$$\begin{array}{cccc} 1 & 1 & & \\ & (3 & 6 & 5)_9 \\ + & (6 & 7 & 3)_9 \\ \hline (& 1 & 4 & 8)_9 \end{array}$$

Bases (general)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum falls outside the range $[0, b)$:

$$\begin{array}{cccc} 1 & 1 & & \\ & (3 & 6 & 5)_9 \\ + & (6 & 7 & 3)_9 \\ \hline (1 & 1 & 4 & 8)_9 \end{array}$$

Bases (general)

Q: If we use three digits in base b, how many numbers can we represent?

A: b^3 (including zero)

<u>Note</u>:

- As before, the sum of two numbers represented using three digits may require four digits to store:

$$\begin{array}{rcccc} & (3 & 6 & 5)_9 \\ + & (6 & 7 & 3)_9 \\ \hline (1 & 1 & 4 & 8)_9 \end{array}$$

Bases (general)

Q: If we use three digits in base b, how many numbers can we represent?

A: b^3 (including zero)

<u>Note</u>:

- As before, the sum of two numbers represented using three digits may require four digits to store:

$$\begin{array}{rccc} & (3 & 6 & 5)_9 \\ + & (6 & 7 & 3)_9 \\ \hline & (1 & 4 & 8)_9 \end{array}$$

- If we only use three digits, we lose the leading digit to <u>overflow</u>
- This is the same as the number mod b^3

Bases (examples)

What is this value in base 10?

- $(125)_8 =$

Bases (examples)

What is this value in base 10?

- $(125)_8 = 1 \times 8^2 + 2 \times 8^1 + 5 \times 8^0$

$$= 64 + 16 + 5$$

$$= 85$$

Bases (examples)

What is this value in base 10?

- $(125)_8 = (85)_{10}$
- $(10010)_2 =$

Bases (examples)

What is this value in base 10?

- $(125)_8 = (85)_{10}$
- $(10010)_2 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$$= 16 + 2$$

$$= 18$$

Bases (examples)

What is this value in base 10?

- $(125)_8 = (85)_{10}$
- $(10010)_2 = (18)_2$

What is this value in base 4?

- $(13)_{10} =$

Bases (examples)

What is this value in base 10?

- $(125)_8 = (85)_{10}$
- $(10010)_2 = (18)_2$

What is this value in base 4?

- $(13)_{10} = 12 + 1$

$$= 3 \times 4^1 + 1 \times 4^0$$

$$= (31)_4$$

Bases (examples)

What is this value in base 10?

- $(125)_8 = (85)_{10}$
- $(10010)_2 = (18)_2$

What is this value in base 4?

- $(13)_{10} = (31)_4$
- $(1101)_2 =$

Bases (examples)

What is this value in base 10?

- $(125)_8 = (85)_{10}$
- $(10010)_2 = (18)_2$

What is this value in base 4?

- $(13)_{10} = (31)_4$
- $(1101)_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$$= 8 + 4 + 1$$

$$= (13)_{10}$$

$$= (31)_4$$

Bases (examples)

What is this value in base 10?

- $(125)_8 = (85)_{10}$
- $(10010)_2 = (18)_2$

What is this value in base 4?

- $(13)_{10} = (31)_4$
- $(1101)_2 = (31)_4$

<u>Alternatively</u>:

We can note that $4 = 2^2$

$\Rightarrow$ Two consecutive digits in base 2 encode exactly one digit in base 4.

$$\Rightarrow \quad (1101)_2 = (11)_2(01)_2 = (3)_4(1)_4 = (31)_4$$

Bases (in the wild)

- Decimal (base 10)
  - We have ten fingers
- Sexagesimal (base 60):
  - Minutes / seconds
  - Easy to tell if a number is divisible by 2, 3, 4, 5, 6 , 10, 12, 15, or 30
  - Dates back to the Babylonians

Bases (in the wild)

- Binary (base 2)
  - Numbers in a computer

- Hexadecimal a.k.a. hex (base 16)
  - Numbers in a computer ($16 = 2^4$)
    - We can easily convert binary to hex by grouping sets of four digits
    - We get a more compact representation, replacing 4 digits with 1

Bases (in the wild)

- Binary (base 2)
  - Numbers in a computer

- Hexadecimal a.k.a. hex (base 16)

  Q: How should we read the digits? $(115)_{16}$
  - $(115)_{16} = 1 \times 16^2 + 1 \times 16^1 + \quad 5 \times 16^0$
  - $(115)_{16} = \qquad\qquad 1 \times 16^1 + 15 \times 16^0$
  - $(115)_{16} = \qquad\qquad 11 \times 16^1 + \quad 5 \times 16^0$

Bases (in the wild)

- Binary (base 2)
  - Numbers in a computer

- Hexadecimal a.k.a. hex (base 16)

  Q: How should we separate digits? $(115)_{16}$

  A: Use numbers and letters:
  - $\{0,1,2,3,4,5,6,7,8,9\}$ to represent numbers in the range $[0,10)$
  - $\{a, b, c, d, e, f\}$ to represent values in the range $[10,16)$:
    - $(115)_{16} = 1 \times 16^2 + 1 \times 16^1 + \ 5 \times 16^0$
    - $(1f)_{16} \quad = \qquad 1 \times 16^1 + 15 \times 16^0$
    - $(b5)_{16} \quad = \qquad 11 \times 16^1 + \ 5 \times 16^0$

Representing integers

- On most machines, `[unsigned] int`s are represented using 4 bytes*
  - Each byte is composed of 8 bits

  ⇒ An `[unsigned] int` is represented by 32 bits
  - Each bit can be either "on" or "off"

  ⇒ An `[unsigned] int` is represented in binary using 32 digits with values 0 or 1

  ⇒ An `[unsigned] int` can have one of 2^{32} values

*"[...]" notation indicates an optional argument

Representing integers

- On most machines, **[unsigned] int**s are represented using 4 bytes
  - Each byte is composed of 8 bits

  ⇒ An **[unsigned] int** is represented by 32 bits
  - Each bit can be either "on" or "off"

  ⇒ An **[unsigned] int** is represented in binary using 32 digits with values 0 or 1

  ⇒ An **[unsigned] int** can have one of 2^{32} values

```
#include <stdio.h>
int main( void )
{
        int a = 30;
        printf( "%d\n" , a );
        return 0;
}
```

```
>> ./a.out
30
>>
```

On the machine, **a** is assigned the value:

$$a \leftarrow (\ 00000000\ 00000000\ 00000000\ 00011110\ )_2$$

$$a \leftarrow (\ 00\ 00\ 00\ 1e)_{16}$$

Representing integers

- On most machines, **[unsigned] int**s are represented using 4 bytes
  - Each byte is composed of 8 bits

  ⇒ An **[unsigned] int** is represented by 32 bits
  - Each bit can be either "on" or "off"

  ⇒ An **[unsigned] int** is represented in binary using 32 digits with values 0 or 1

  ⇒ An **[unsigned] int** can have one of 2^{32} values

```
#include <stdio.h>
int main( void )
{
        int a = 0x1e;
        printf( "%d\n" , a );
        return 0;
}
```

```
>> ./a.out
30
>>
```

On the machine, **a** is assigned the value:

$$a \leftarrow (\ 00000000\ 00000000\ 00000000\ 00011110\ )_2$$

$$a \leftarrow (\ 00\ 00\ 00\ 1e)_{16}$$

- You can assign using base 16 by preceding the number with **0x** to indicate hex

Representing integers

- On most machines, **[unsigned] int**s are represented using 4 bytes
  - Each byte is composed of 8 bits

  ⇒ An **[unsigned] int** is represented by 32 bits
  - Each bit can be either "on" or "off"

  ⇒ An **[unsigned] int** is represented in binary using 32 digits with values 0 or 1

  ⇒ An **[unsigned] int** can have one of 2^{32} values

```
#include <stdio.h>
int main( void )
{
        int a = 30;
        printf( "%x\n" , a );
        return 0;
}
```

```
>> ./a.out
1e
>>
```

On the machine, **a** is assigned the value:

$$a \leftarrow (\ 00000000\ 00000000\ 00000000\ 00011110\ )_2$$

$$a \leftarrow (\ 00\ 00\ 00\ 1e)_{16}$$

- You can assign using base 16 by preceding the number with **0x** to indicate hex
- You can print the base 16 representation by using **%x** for formatting

Representing integers

- On most machines, [unsigned] chars are represented using 1 byte
  - ⇒ A [unsigned] char can have one of 2^8 values
- On most machines, [unsigned] long ints are represented using 8 bytes
  - A [unsigned] long int can have one of 2^{64} values

Representing integers*

⇒ An [unsigned] char can have one of $2^8 = 256$ values

⇒ [unsigned] chars are integer values mod 2^8

- <u>unsigned char</u>: We will use the range $[0,256)$ to represent integers
- <u>char</u>: We will use the range $[-128,128)$ to represent integers

Q: What's the difference?
Integers mod 2^8 are integers mod 2^8, regardless of the representation!!!

*For simplicity the following discussion will focus on chars, though it holds for other integer representations (e.g. ints and long ints)

Representing integers

- <u>unsigned char</u>: We will use the range $[0,256)$ to represent integers
- <u>char</u>: We will use the range $[-128,128)$ to represent integers

Q: What's the difference?

A1: Is $125 < 129 \bmod 256$?

Since $129 \equiv -127 \bmod 256$,
it depends on the range we use

```
#include <stdio.h>
int main( void )
{
        unsigned char c1 = 125 , c2 = 129;
        printf( "%d\n" , c1<c2 );
        return 0;
}
```

```
>> ./a.out
1
>>
```

Representing integers

- <u>unsigned char</u>: We will use the range $[0,256)$ to represent integers
- <u>char</u>: We will use the range $[-128,128)$ to represent integers

Q: What's the difference?

A1: Is $125 < 129 \bmod 256$?

Since $129 \equiv -127 \bmod 256$,
it depends on the range we use

```
#include <stdio.h>
int main( void )
{
        char c1 = 125 , c2 = 129;
        printf( "%d\n" , c1<c2 );
        return 0;
}
```

```
>> ./a.out
0
>>
```

Representing integers

- <u>unsigned char</u>: We will use the range $[0,256)$ to represent integers
- <u>char</u>: We will use the range $[-128,128)$ to represent integers

Q: What's the difference?

A2: In principal:

- Overflow with signed integers can result in undefined behavior*
- Overflow with unsigned integers assumes you wanted the modular representation

*When the standard specifies that behavior is undefined, the actual behavior can be compiler specific

Representing integers

- <u>Addition</u>:
  We add two numbers, $a + b$, by adding the digits from smallest to largest
  - We carry as necessary
  - And we cut off at 8 bits

$$
\begin{array}{r}
1\,1 \quad\;\; 11 \\
(11010011)_2 \\
+(01000110)_2 \\
\hline
(1\,00011001)_2 \\
= (00011001)_2
\end{array}
$$

Representing integers

- <u>Addition</u>:
  We add two numbers, $a + b$. by adding the digits from smallest to largest
  - We carry as necessary
  - And we cut off at 8 bits

$$
\begin{array}{r}
1\,1 \quad\;\; 11 \\
(11010011)_2 \\
+(01000110)_2 \\
\hline
(00011001)_2
\end{array}
$$

Q: What about subtraction, $a - b$?

Representing integers

- <u>Addition</u>:
  We add two numbers, $a + b$. by adding the digits from smallest to largest
  - We carry as necessary
  - And we cut off at 8 bits

$$
\begin{array}{r}
1\,1 \quad\;\; 11 \\
(11010011)_2 \\
+(01000110)_2 \\
\hline
(00011001)_2
\end{array}
$$

Q: What about subtraction, $a - b = a + (-b)$?

Equivalently, how do we define the negative of a number?

Negation

- <u>Recall</u>:
  The negative of an integer is the number we would have to add to get back zero.
- <u>Defining negative one</u>:
  - Mod 256, we have $-1 \equiv 255 = (11111111)_2$

Negation

- <u>Recall</u>:
  The negative of an integer is the number we would have to add to get back zero.
- <u>Defining negatives in general</u>:
  1. Given a binary value in 8 bits:
     $$(10011101)_2$$
  2. We can flip the bits:
     $$(01100010)_2$$
  3. Adding the two values we get $255 \equiv -1$:
     $$(11111111)_2$$
  4. Adding one to that we get 0

Negation

- <u>Recall</u>:
  The negative of an integer is the number we would have to add to get back zero.
- <u>2's complement</u>:
  To get the binary representation of the negative of a number
  1. Flip the bits
  2. Add 1

Bit-wise operations

- n << k: shifts n to the left by k positions
  - Note that the new, right-most, bits are set to 0

```
#include <stdio.h>
int main( void )
{
        char a = 5;        //  (00000101)_2
        char b = a << 2;   //  (00010100)_2
        printf( "%d\n" , b );
        return 0;
}
```

```
>> ./a.out
20
>>
```

Bit-wise operations

- n << k: shifts n to the left by k positions
- n >> k: shifts n to the right by k positions
  - Note that the new, left-most, bits are set to 0

```
#include <stdio.h>
int main( void )
{
        char a = 5;        //  (00000101)_2
        char b = a >> 2;   //  (00000001)_2
        printf( "%d\n" , b );
        return 0;
}
```

```
>> ./a.out
1
>>
```

Bit-wise operations

- n << k: shifts n to the left by k positions
- n >> k: shifts n to the right by k positions
  - Note that the new, left-most, bits are set to 0
  - And once shifted out, the bits are lost

```
#include <stdio.h>
int main( void )
{
        char a = 5;        //  (00000101)_2
        char b = a >> 2;   //  (00000001)_2
        char c = b << 2;   //  (00000100)_2
        printf( "%d\n" , b );
        return 0;
}
```

```
>> ./a.out
4
>>
```

Bit-wise operations

- n << k: shifts n to the left by k positions
- n >> k: shifts n to the right by k positions
- n & m: compute the bit-wise *and* of n and m
  - The output is 1 if and only if both bits are 1

```
#include <stdio.h>
int main( void )
{
        char a = 5;        //  (00000101)_2
        char b = 14;       //  (00001110)_2
        char c = a & b;    //  (00000100)_2
        printf( "%d\n" , a & b );
        return 0;
}
```

```
>> ./a.out
4
>>
```

Bit-wise operations

- n << k: shifts n to the left by k positions
- n >> k: shifts n to the right by k positions
- n & m: compute the bit-wise *and* of n and m
- n | m: compute the bit-wise *or* of n and m
  - The output is 1 if either (or both) bits are 1

```
#include <stdio.h>
int main( void )
{
        char a = 5;         //  (00000101)_2
        char b = 14;        //  (00001110)_2
        char c = a | b;     //  (00001111)_2
        printf( "%d\n" , a | b );
        return 0;
}
```

```
>> ./a.out
15
>>
```

Bit-wise operations

- n << k: shifts n to the left by k positions
- n >> k: shifts n to the right by k positions
- n & m: compute the bit-wise *and* of n and m
- n | m: compute the bit-wise *or* of n and m
- n ^ m: compute the bit-wise *exclusive or* of n and m

```
#include <stdio.h>
int main( void )
{
        char a = 5;         //  (00000101)_2
        char b = 14;        //  (00001110)_2
        char c = a ^ b;     //  (00001011)_2
        printf( "%d\n" , c );
        return 0;
}
```

```
>> ./a.out
11
>>
```

Bit-wise operations

- n << k: shifts n to the left by k positions
- n >> k: shifts n to the right by k positions
- n & m: compute the bit-wise *and* of n and m
- n | m: compute the bit-wise *or* of n and m
- n ^ m: compute the bit-wise *exclusive or* of n and m
- ~n: flip the bits of n

```
#include <stdio.h>
int main( void )
{
        char a = 5;         //  (00000101)_2
        char b = ~a;        //  (11111010)_2
        char c = b + 1;     //  (11111011)_2
        printf( "%d\n" , c );
        return 0;
}
```

```
>> ./a.out
-5
>>
```

Bit-wise operations

- There are also variants of these that evaluate-and-set
  - n <<= k
  - n >>= k
  - n &= m
  - n |= m
  - n ^= m

```
#include <stdio.h>
int main( void )
{
        char a = 5;          //  (00000101)_2
        a <<= 3;             //  (00101000)_2
        printf( "%d\n" , a );
        return 0;
}
```

```
>> ./a.out
40
>>
```

Bit-wise operations

- Masking
  - We can determine if a bit is on or off using << and &

```
#include <stdio.h>
int main( void )
{
        char a = 5;          //  (00000101)_2
        char mask = 1<<2;//  (00000100)_2
        char c = a & mask;
        printf( "%d %d\n" , c , c!=
        return 0;
}
```

```
>> ./a.out
4 1
>>
```

Bit-wise operations

- Masking
  - We can determine if a bit is on or off using << and &
  - Or we can use >> and &

```
#include <stdio.h>
int main( void )
{
        char a = 5;         //  (00000101)_2
        char b = a>>2;      //  (00000001)_2
        char c = b & 1;
        printf( "%d %d\n" , c , c!=
        return 0;
}
```

```
>> ./a.out
1 1
>>
```

Bit-wise operations

- Masking
  - We can determine if a bit is on or off using << and &
  - Or we can use >> and &
  - <u>Note</u>:

    Integers in $[0,128)$ all have a binary representation of the form:

    $$(0\ *******)$$

    Integers in $[128,256) \equiv [-128,0)$ all have a binary representation of the form:

    $$(1\ *******)$$

Bit-wise operations

- Masking
  - We can determine if a bit is on or off using << and &
  - Or we can use >> and &
  - We can determine the sign by testing the highest (a.k.a. most significant) bit

```
#include <stdio.h>
int main( void )
{
        char a = 5;          // (00000101)_2
        char b = -4;         // (11111100)_2
        char mask = 1<<7;// (10000000)_2
        printf( "%d %d\n" , ( a & mask )!=0 , ( b & mask )!=0 );
        return 0;
}
```

```
>> ./a.out
0 1
>>
```

Bit-wise operations

- Masking
  - We can determine if a bit is on or off using << and &
  - Or we can use >> and &
  - We can determine the sign by testing the highest (a.k.a. most significant) bit*

```
#include <stdio.h>
int main( void )
{
        char a = 5;          // (00000101)_2
        char b = -4;         // (11111100)_2
        char mask = 1<<7;// (10000000)_2
        printf( "%d %d\n" , ( a & mask )!=0 , ( b & mask )!=0 );
        return 0;
}
```

```
>> ./a.out
```

*This assumes that the most significant bit is on the left.
It's true for most (big-endian) machines but should not be assumed.

Bit-wise operations

- Bit interleaving
  - Combining the bit-wise operations, we can interleave the bits (low-order) bits of two integers:

$$(00010101)_2$$
$$(00001001)_2$$

$$\Downarrow$$

$$(\qquad\qquad)_2$$

```
#include <stdio.h>

char interleave( char a , char b )
{
        …
}
int main( void )
{
        char a = 31;        //  (00010101)_2
        char b = 9;         //  (00001001)_2
        char c = interleave( a , b );
        printf( "%d\n" , c );
        return 0;
}
```

Bit-wise operations

- Bit interleaving
  - Combining the bit-wise operations, we can interleave the bits (low-order) bits of two integers:

$$(0001010\mathbf{1})_2$$
$$(00001001)_2$$

$$\Downarrow$$

$$(\qquad\qquad 1)_2$$

```
#include <stdio.h>

char interleave( char a , char b )
{
        …
}
int main( void )
{
        char a = 31;        //  (00010101)_2
        char b = 9;         //  (00001001)_2
        char c = interleave( a , b );
        printf( "%d\n" , c );
        return 0;
}
```

Bit-wise operations

- Bit interleaving
  - Combining the bit-wise operations, we can interleave the bits (low-order) bits of two integers:

$$(00010101)_2$$
$$(00001001)_2$$

$$\Downarrow$$

$$(\qquad\qquad 11)_2$$

```
#include <stdio.h>

char interleave( char a , char b )
{
        …
}
int main( void )
{
        char a = 31;        //  (00010101)_2
        char b = 9;         //  (00001001)_2
        char c = interleave( a , b );
        printf( "%d\n" , c );
        return 0;
}
```

Bit-wise operations

- Bit interleaving
  - Combining the bit-wise operations, we can interleave the bits (low-order) bits of two integers:

$$(00010101)_2$$
$$(00001001)_2$$

$$\Downarrow$$

$$(\qquad\quad 011)_2$$

```
#include <stdio.h>

char interleave( char a , char b )
{
        …
}
int main( void )
{
        char a = 31;        //  (00010101)_2
        char b = 9;         //  (00001001)_2
        char c = interleave( a , b );
        printf( "%d\n" , c );
        return 0;
}
```

Bit-wise operations

- Bit interleaving
  - Combining the bit-wise operations, we can interleave the bits (low-order) bits of two integers:

$$(00010101)_2$$
$$(00001\mathbf{0}01)_2$$

$$\Downarrow$$

$$(\qquad 0011)_2$$

```
#include <stdio.h>

char interleave( char a , char b )
{
        ...
}
int main( void )
{
        char a = 31;        //  (00010101)_2
        char b = 9;         //  (00001001)_2
        char c = interleave( a , b );
        printf( "%d\n" , c );
        return 0;
}
```

Bit-wise operations

- Bit interleaving
  - Combining the bit-wise operations, we can interleave the bits (low-order) bits of two integers:

$$(00010\mathbf{1}01)_2$$
$$(00001001)_2$$

$$\Downarrow$$

$$(\quad 10011)_2$$

```
#include <stdio.h>

char interleave( char a , char b )
{
        …
}
int main( void )
{
        char a = 31;        //  (00010101)_2
        char b = 9;         //  (00001001)_2
        char c = interleave( a , b );
        printf( "%d\n" , c );
        return 0;
}
```

Bit-wise operations

- Bit interleaving
  - Combining the bit-wise operations, we can interleave the bits (low-order) bits of two integers:

$$(00010101)_2$$
$$(00001\mathbf{0}01)_2$$

$$\Downarrow$$

$$(\quad 010011)_2$$

```
#include <stdio.h>

char interleave( char a , char b )
{
        …
}
int main( void )
{
        char a = 31;        //  (00010101)_2
        char b = 9;         //  (00001001)_2
        char c = interleave( a , b );
        printf( "%d\n" , c );
        return 0;
}
```

Bit-wise operations

- Bit interleaving
  - Combining the bit-wise operations, we can interleave the bits (low-order) bits of two integers:

$$(0001\mathbf{0}101)_2$$
$$(00001001)_2$$

$$\Downarrow$$

$$(\ \ 0010011)_2$$

```
#include <stdio.h>

char interleave( char a , char b )
{
        …
}
int main( void )
{
        char a = 31;        //  (00010101)_2
        char b = 9;         //  (00001001)_2
        char c = interleave( a , b );
        printf( "%d\n" , c );
        return 0;
}
```

Bit-wise operations

- Bit interleaving
  - Combining the bit-wise operations, we can interleave the bits (low-order) bits of two integers:

$$(00010101)_2$$
$$(0000\mathbf{1}001)_2$$

$$\Downarrow$$

$$(10010011)_2$$

```
#include <stdio.h>

char interleave( char a , char b )
{
        …
}
int main( void )
{
        char a = 31;        //  (00010101)_2
        char b = 9;         //  (00001001)_2
        char c = interleave( a , b );
        printf( "%d\n" , c );
        return 0;
}
```

Bit-wise operations

- Bit interleaving
  - Combining the bit-wise operations, we can interleave the bits (low-order) bits of two integers:

$$(00010101)_2$$
$$(00001001)_2$$

$$(10010011)_2$$

  - Note that the high 4 bits are ignored

```
#include <stdio.h>

char interleave( char a , char b )
{
        ...
}
int main( void )
{
        char a = 31;        //  (00010101)_2
        char b = 9;         //  (00001001)_2
        char c = interleave( a , b );
        printf( "%d\n" , c );
        return 0;
}
```

```
>> ./a.out
-119
>>
```

Bit-wise operations

- Bit interleaving

Q: Why do we care?

Bit-wise operations

- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0)	(1,0)	(2,0)	(3,0)

Bit-wise operations

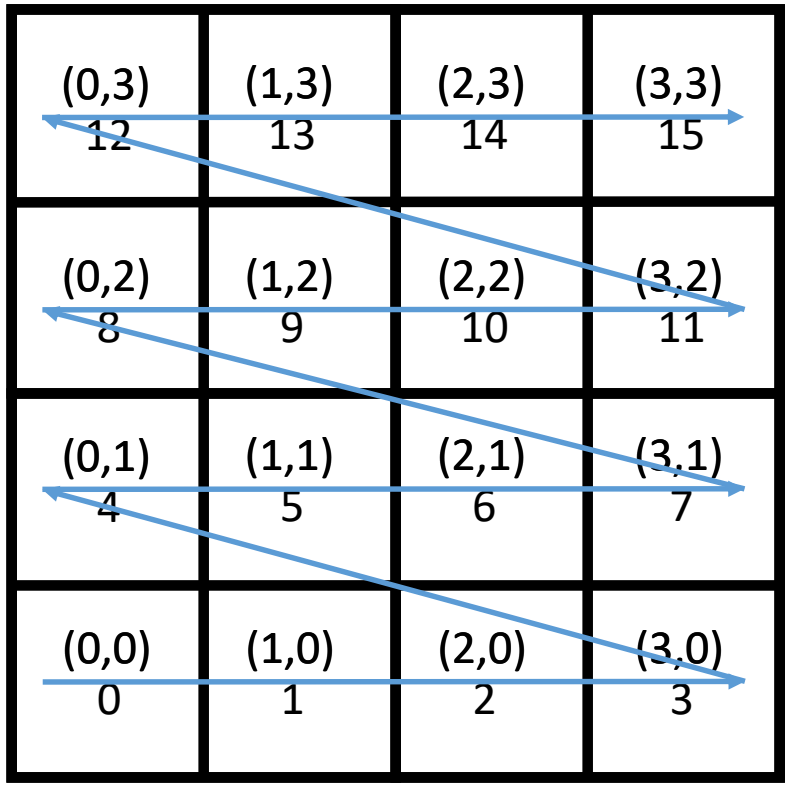
- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order

```
...
char index( char x , char y )
{
        return (y*4) + x;
}
...
```



Bit-wise operations

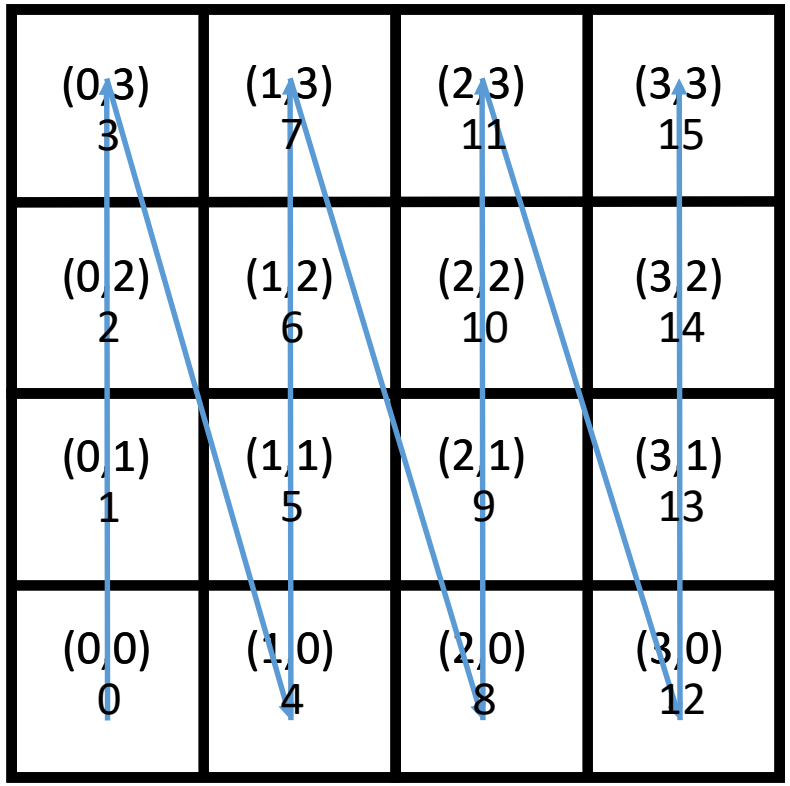
- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order
- We can index the cells in column-major order

```
...
char index( char x , char y )
{
        return y + (4*x);
}
...
```



Bit-wise operations

- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order
- We can index the cells in column-major order
- We can index the cells in interleaved, z-curve, order

```
...
char index( char x , char y )
{
        return interleave( x , y );
}
...
```

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0)	(1,0)	(2,0)	(3,0)

Bit-wise operations

$(00000000)_2$
$(00000000)_2$

$\Downarrow$

$(00000000)_2$

- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order
- We can index the cells in column-major order
- We can index the cells in interleaved, z-curve, order

```
...
char index( char x , char y )
{
        return interleave( x , y );
}
...
```

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0) 0	(1,0)	(2,0)	(3,0)

Bit-wise operations

$(00000001)_2$
$(00000000)_2$

$\Downarrow$

$(00000001)_2$

- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order
- We can index the cells in column-major order
- We can index the cells in interleaved, z-curve, order

```
...
char index( char x , char y )
{
        return interleave( x , y );
}
...
```

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0) 0	(1,0) 1	(2,0)	(3,0)

Bit-wise operations

$(00000010)_2$
$(00000000)_2$

$\Downarrow$

$(00000100)_2$

- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order
- We can index the cells in column-major order
- We can index the cells in interleaved, z-curve, order

```
...
char index( char x , char y )
{
        return interleave( x , y );
}
...
```

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0) 0	(1,0) 1	(2,0) 4	(3,0)

Bit-wise operations

$(00000011)_2$
$(00000000)_2$

$\Downarrow$

$(00000101)_2$

- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order
- We can index the cells in column-major order
- We can index the cells in interleaved, z-curve, order

```
...
char index( char x , char y )
{
        return interleave( x , y );
}
...
```

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0) 0	(1,0) 1	(2,0) 4	(3,0) 5

Bit-wise operations

$(00000000)_2$
$(00000001)_2$

$\Downarrow$

$(00000010)_2$

- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order
- We can index the cells in column-major order
- We can index the cells in interleaved, z-curve, order

```
...
char index( char x , char y )
{
        return interleave( x , y );
}
...
```

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1) 2	(1,1)	(2,1)	(3,1)
(0,0) 0	(1,0) 1	(2,0) 4	(3,0) 5

Bit-wise operations

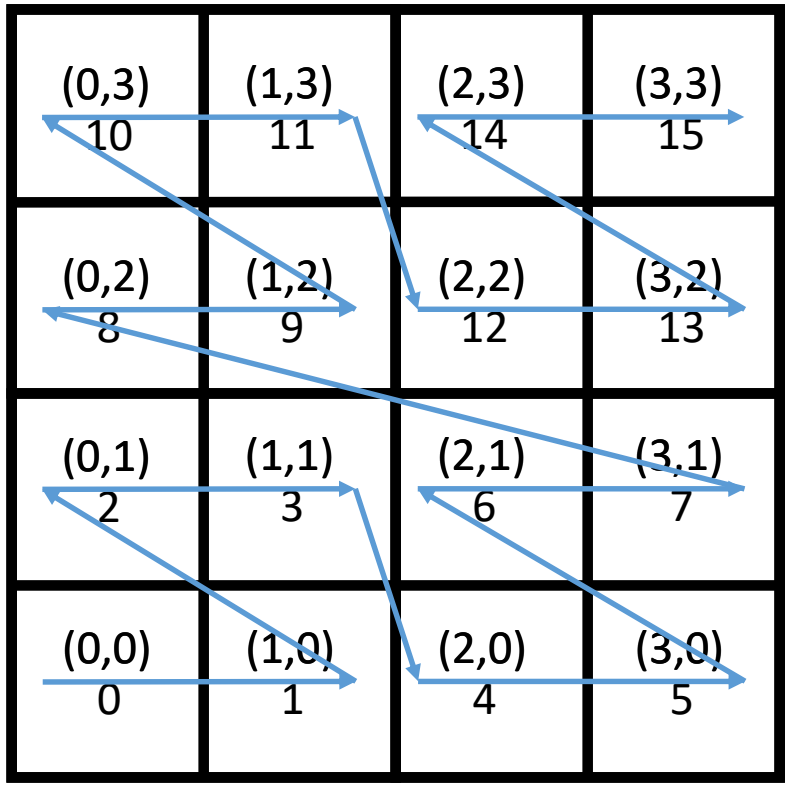
- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order
- We can index the cells in column-major order
- We can index the cells in interleaved, z-curve, order

```
...
char index( char x , char y )
{
        return interleave( x , y );
}
...
```



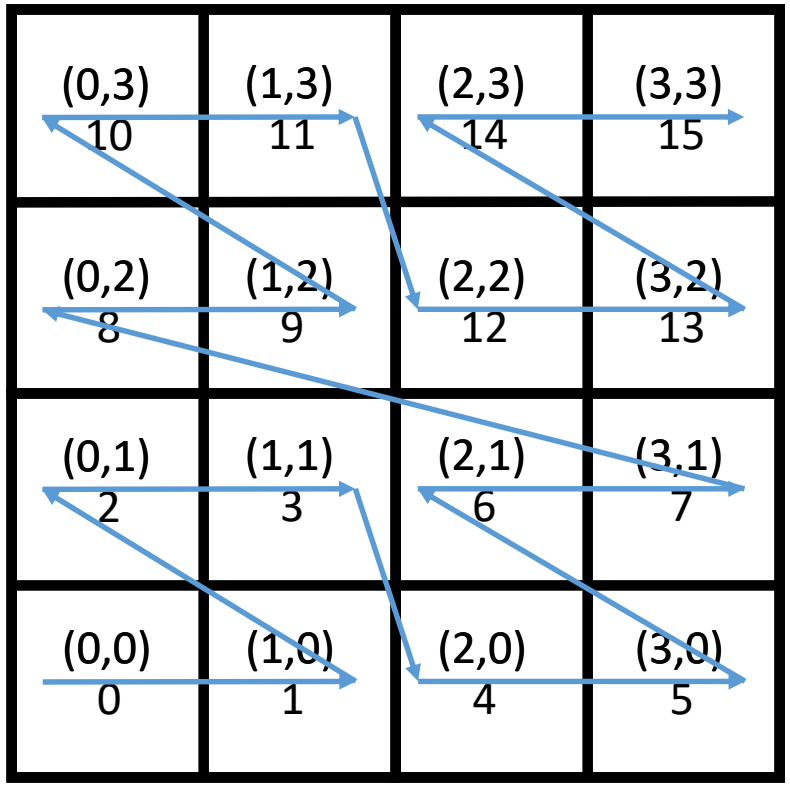
Bit-wise operations

- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order
- We can index the cells in column-major order
- We can index the cells in interleaved, z-curve, order
  - Cells within a sub-block are indexed sequentially



Bit-wise operations

- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order
- We can index the cells in column-major order
- We can index the cells in interleaved, z-curve, order
  - Cells within a sub-block are indexed sequentially
  - As are cells within a sub-block of a sub-block

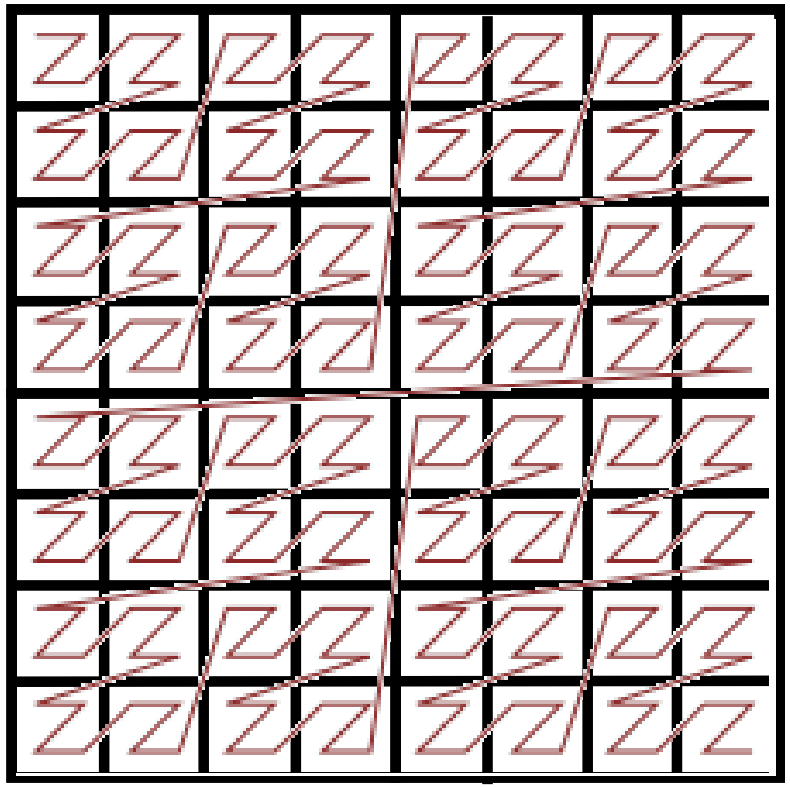


Image courtesy of https://en.wikipedia.org/wiki/Z-order_curve

Bit-wise operations

- Bit interleaving

Q: Why do we care?

A: Ordering the cells of a grid

- We can index the cells in row-major order
- We can index the cells in column-major order
- We can index the cells in interleaved, z-curve, order
  - Cells within a sub-block are indexed sequentially
  - As are cells within a sub-block of a sub-block
  - And this holds for higher dimensions

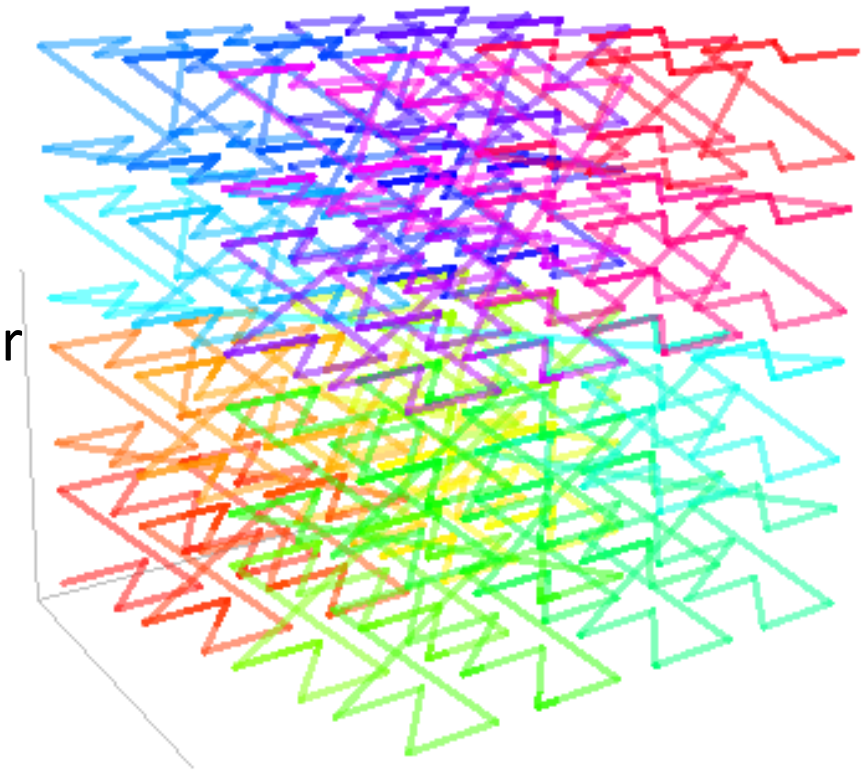


Image courtesy of https://en.wikipedia.org/wiki/Z-order_curve