Discretization

Phys 750 Lecture 2

Number representations

- The obvious strategies ...
  - simple enumeration:
  - labelling: e.g., Roman numerals
- For computation, we need a systematic number representation in which basic arithmetic operations are mechanistic

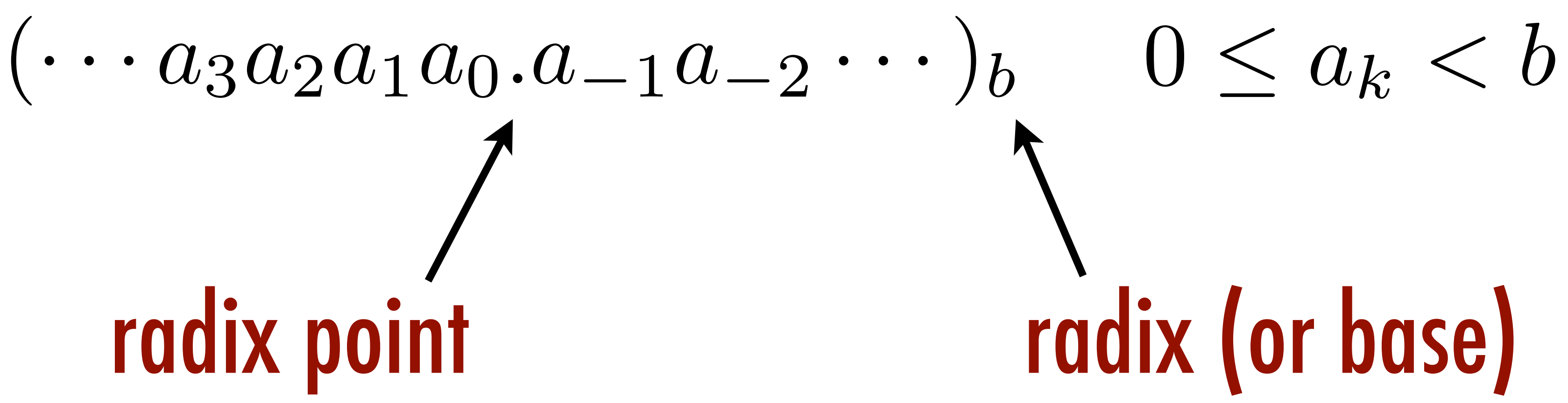
/ // /// //// ~~////~~

1	I
2	II
3	III
4	IV
5	V
6	VI
7	VII
8	VIII
9	IX

10	X
20	XX
30	XXX
40	XL
50	L
100	C
500	D
1000	M
1998	MCMXCVIII

Positional number systems

- positional notation



$$(\cdots a_3 a_2 a_1 a_0 . a_{-1} a_{-2} \cdots)_b \qquad 0 \leq a_k < b$$

radix point

radix (or base)

- conventional number system

$$b = 10$$

$$a_k \in \{0, 1, 2, \ldots, 9\}$$

Positional number systems

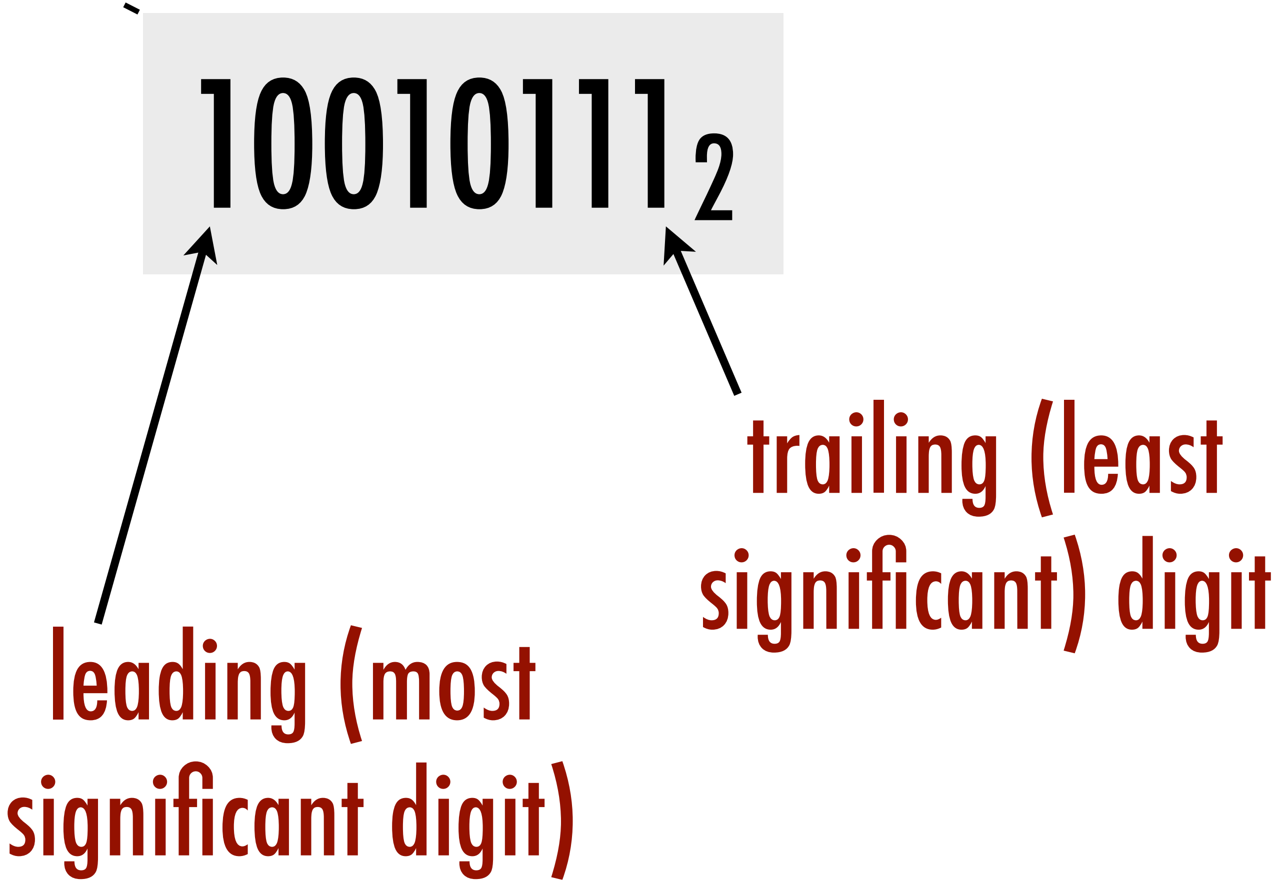
- base 2: 10010111_2
- base 8: 1735_8
- base 10: 0, 234, 1983
- base 16: 3F7A
- base 60: $23^\circ\ 44'\ 12''$

Positional number systems

- base 2: 10010111_2 ← binary
- base 8: 1735_8 ← octal (octonal)
- base 10: 0, 234, 1983 ← decimal
- base 16: 3F7A ← hexadecimal (sexadecimal)
- base 60: $23^{\circ}\ 44'\ 12''$ ← sexagesimal

Positional number systems

- base 2: 10010111_2
- base 8: 1735_8
- base 10: 0, 234, 1983
- base 16: 3F7A
- base 60: $23^\circ\ 44'\ 12''$



10010111_2

trailing (least significant) digit

leading (most significant digit)

Positional number systems

- base 2: 10010111_2
- base 8: 1735_8
- base 10: 0, 234, 1983
- base 16: 3F7A
- base 60: 23º 44′ 12′′

$$
\begin{aligned}
&= 2^7 + 2^4 + 2^2 + 2^1 + 2^0 \\
&= 128 + 32 + 4 + 2 + 1 \\
&= 167
\end{aligned}
$$

Positional number systems

- base 2: 10010111_2
- base 8: 1735_8
- base 10: 0, 234, 1983
- base 16: 3F7A
- base 60: 23° 44′ 12″

$$
\begin{aligned}
&= 1 \cdot 8^3 + 7 \cdot 8^2 + 3 \cdot 8^1 + 5 \cdot 8^0 \\
&= 512 + 7 \cdot 64 + 3 \cdot 8 + 5 \\
&= 989
\end{aligned}
$$

Positional number systems

- base 2: 10010111_2
- base 8: 1735_8
- base 10: 0, 234, 1983
- base 16: 3F7A
- base 60: $23^o\ 44'\ 12''$

conventional hexadecimal digits

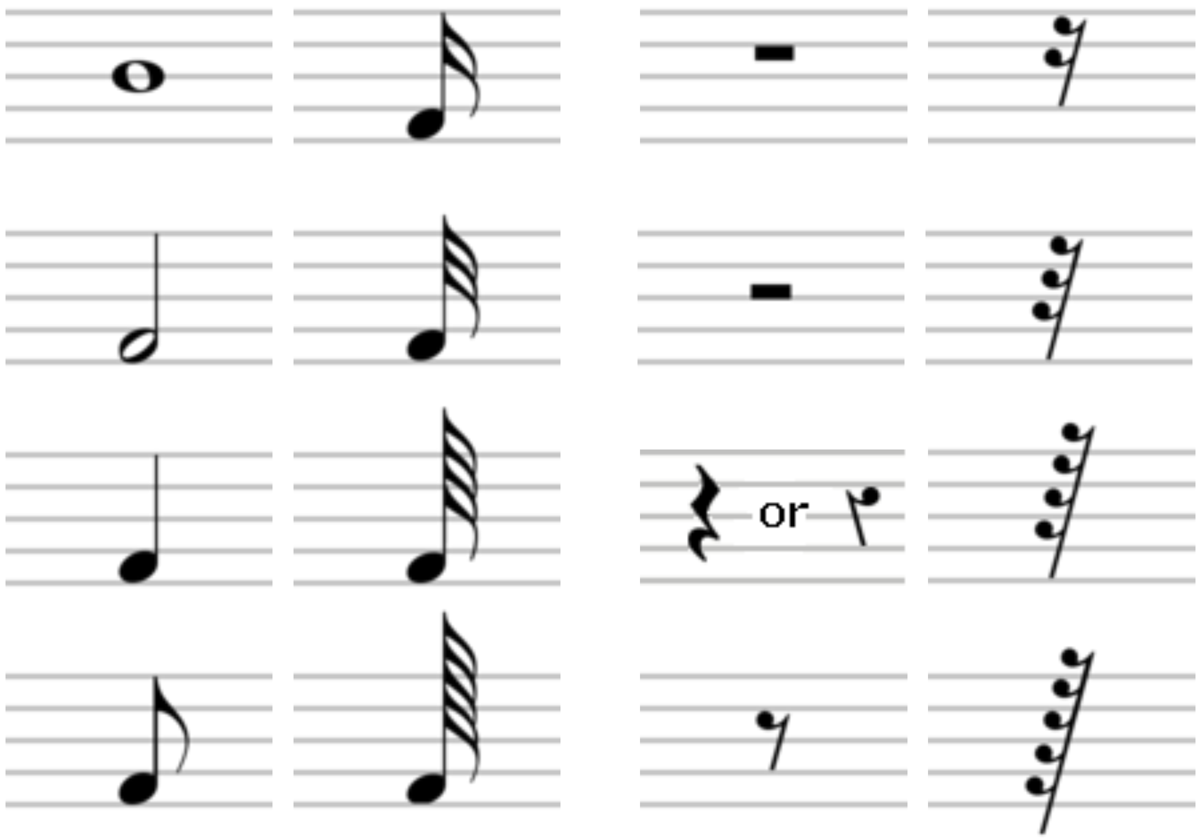
$$a_k \in \{0, \ldots, 9, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}$$

$$\begin{aligned}
&= 3 \cdot 16^3 + 15 \cdot 16^2 + 7 \cdot 16^1 + 10 \cdot 16^0 \\
&= 3 \cdot 4096 + 15 \cdot 256 + 7 \cdot 16 + 10 \\
&= 16250
\end{aligned}$$

Binary systems

- Western system of musical notation

notes

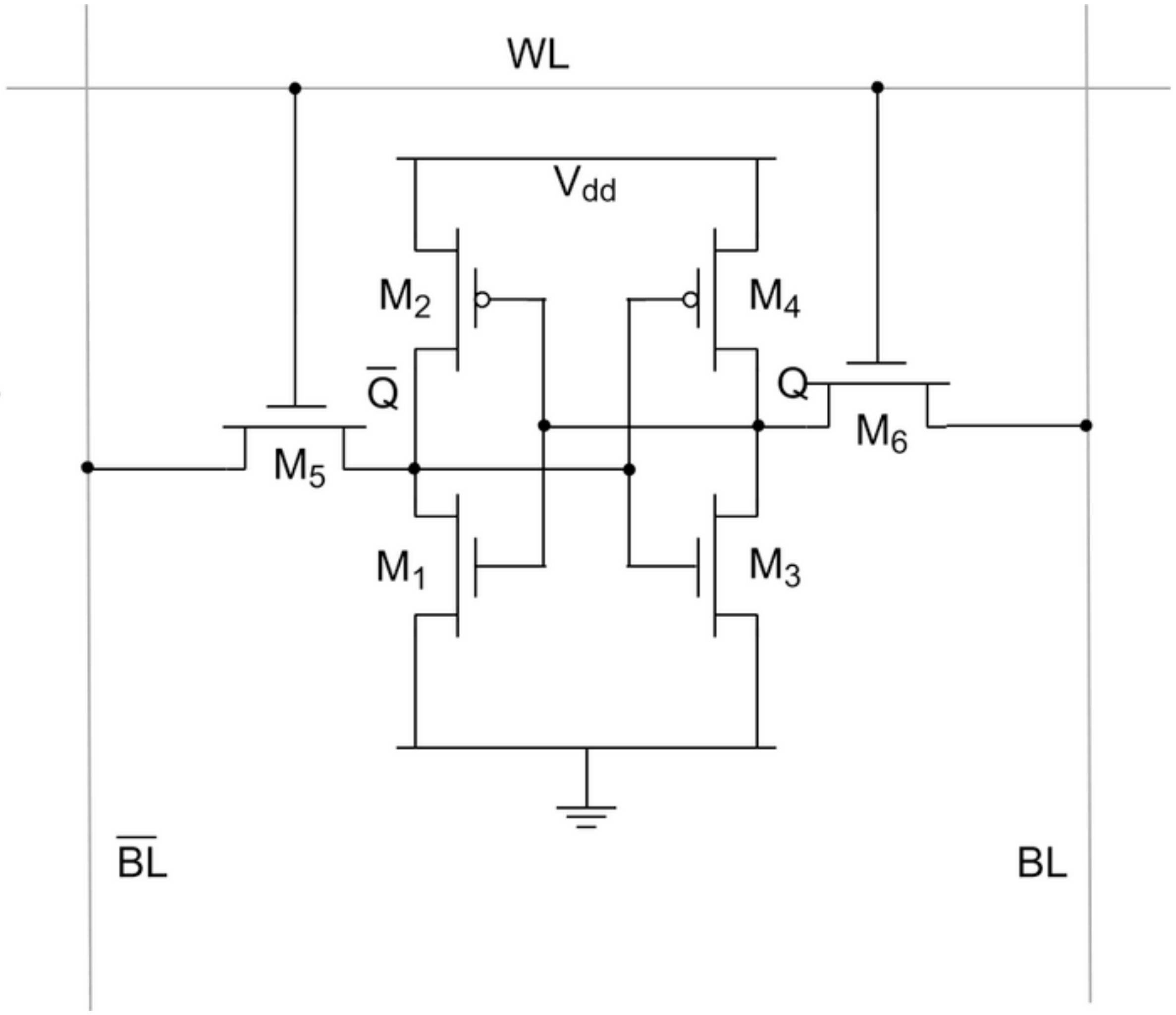
rests

Binary systems

- English system of weights and measures

2 gills = 1 chopin
2 chopins = 1 pint
2 pints = 1 quart
2 quarts = 1 pottle
2 pottles = 1 gallon
2 gallons = 1 peck
2 pecks = 1 demibushel
2 demibushels = 1 firkin
2 firkins = 1 kilderkin
2 kilderkins = 1 barrel
2 barrels = 1 hogshead
2 hogsheads = 1 pipe
2 pipes = 1 tun

Binary systems

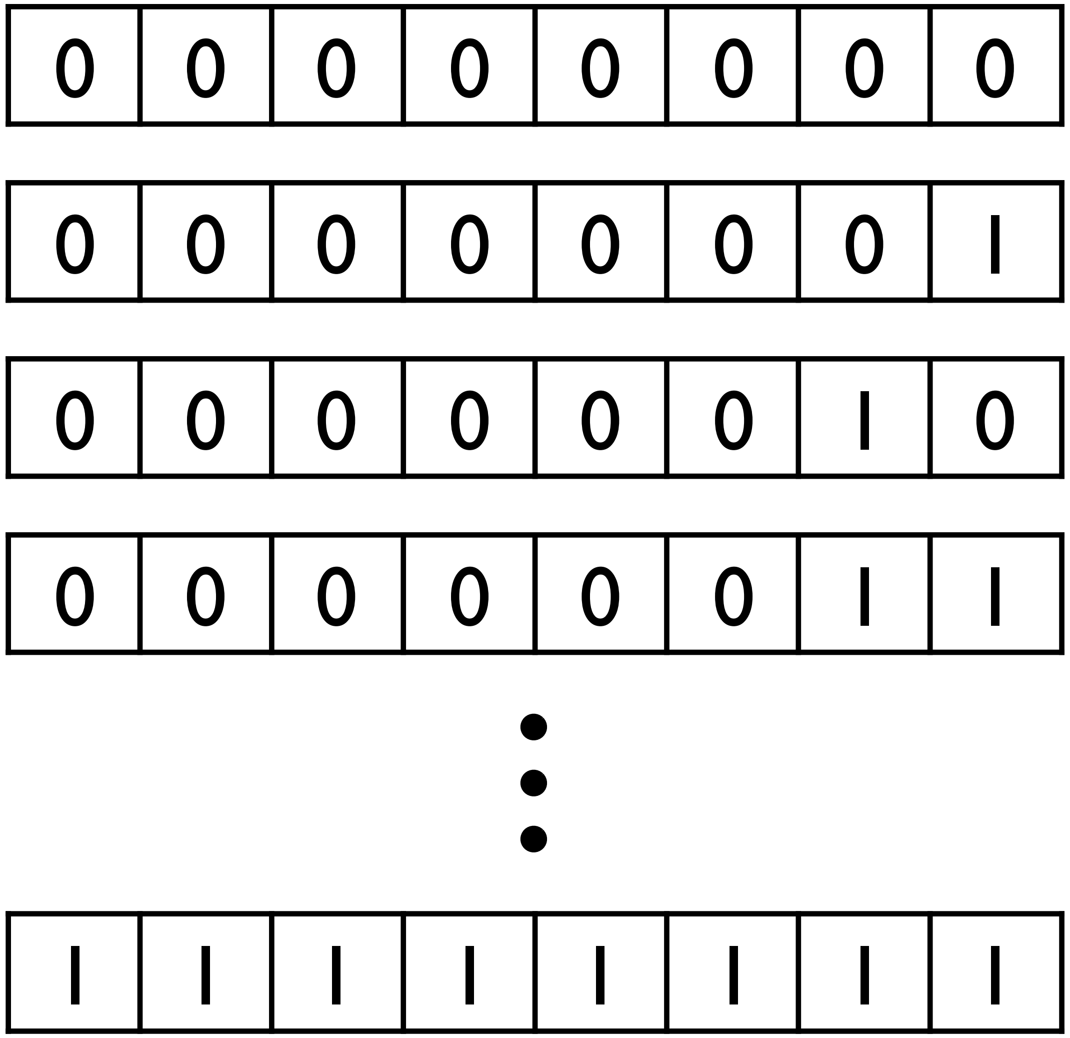
- a modern digital computer stores information in a memory cell called a "bit"
- the four-transistor arrangement has two stable internal states,



memory transistors M1,...,M4
access transistors M5,M6

Fixed-width binary

- An unsigned, 8-bit binary number can represent the natural numbers 0 – 255
- There are 2^8 unique patterns of 0 and 1

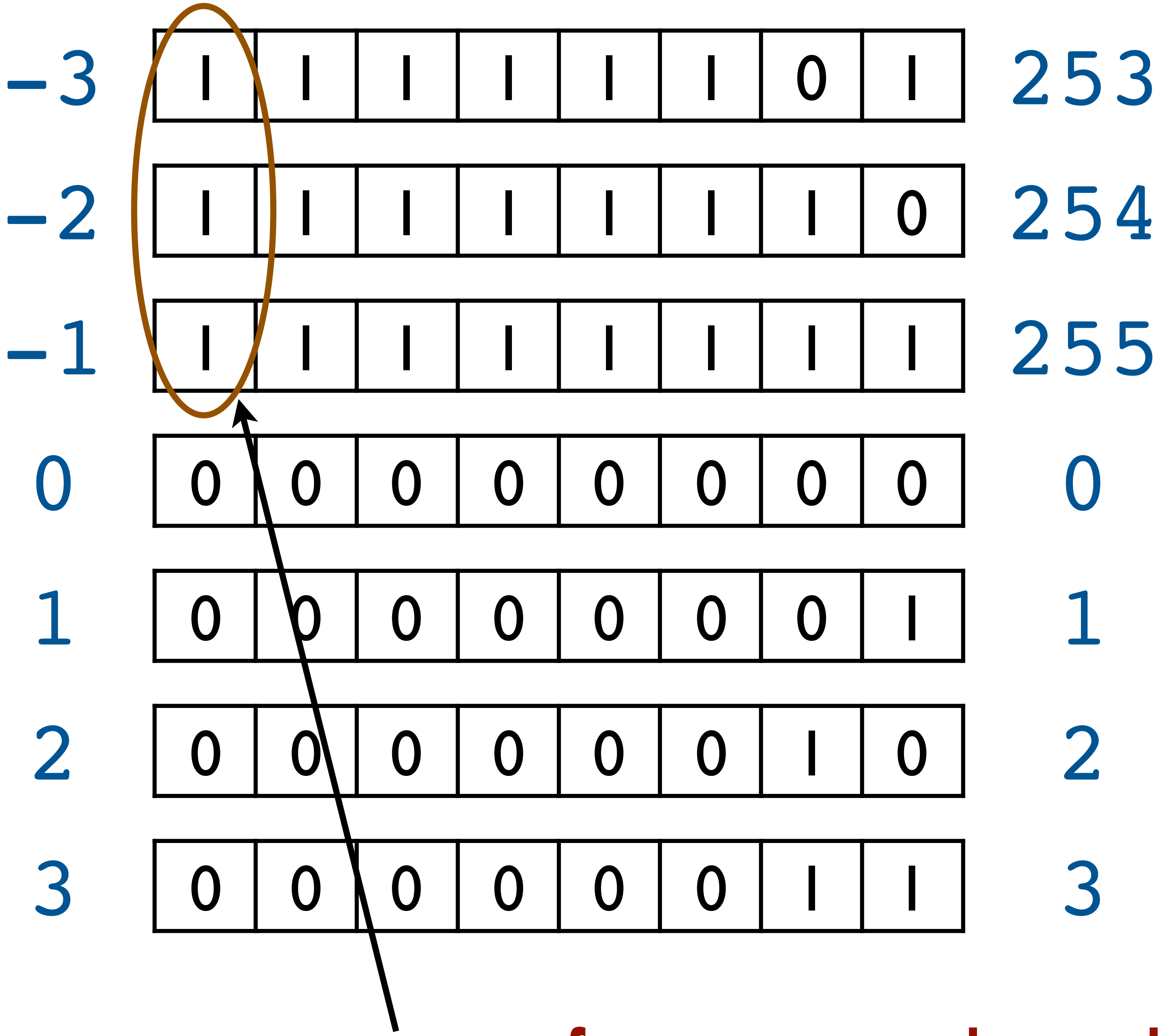


0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	1

⋮

1	1	1	1	1	1	1	1

Fixed-width binary



two's complement									conventional binary
−3	1	1	1	1	1	1	0	1	253
−2	1	1	1	1	1	1	1	0	254
−1	1	1	1	1	1	1	1	1	255
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	3

two's complement

$-128 \leq x \leq 127$

conventional binary

$0 \leq x \leq 255$

sign information resides in the high bit

Potential dangers

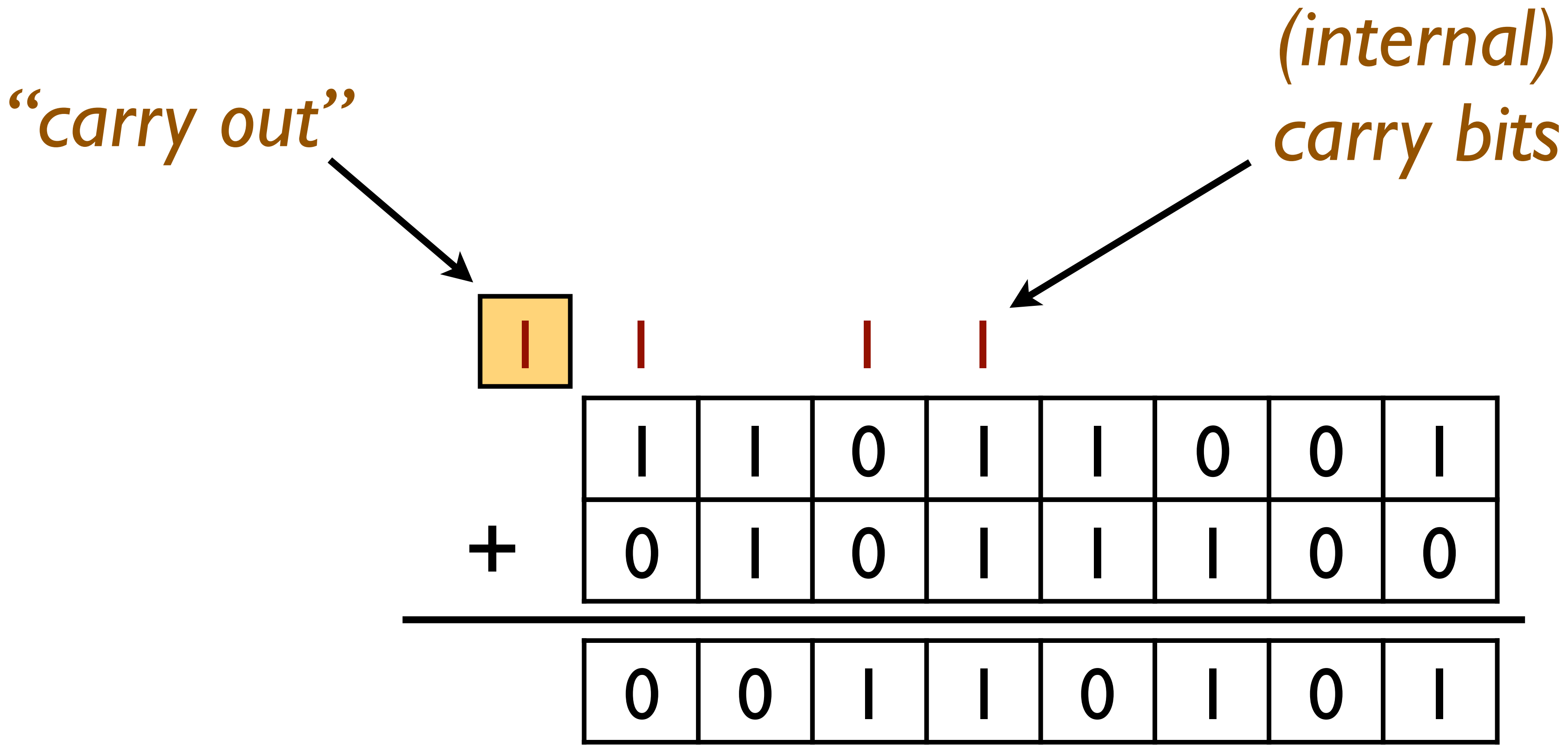
- Fixed-width binary numbers can represent only a limited range of integers
- The result of an operation (such as addition or multiplication) performed on pairs of representable integers may not be representable itself!
- This condition is called "overflow"

- Wait, does this really matter? Yes, there many famous real-life examples (YouTube: ariane 5 explosion)

Why two’s complement?

- Unique bit representation for zero
- Algebraic operations—in terms of the manipulation of the underlying bit representations—are identical for unsigned and two’s complement numbers
- Single hardware implementation; only the interpretation changes with context

Why two’s complement?



“carry out”

(internal)
carry bits

1	1		1	1				
	1	1	0	1	1	0	0	1
+	0	1	0	1	1	1	0	0
	0	0	1	1	0	1	0	1

`217+92==53`

overflow

`-39+92==53`

correct

Why two's complement?

		1	1	1	1	1	1	
	1	0	1	1	0	1	0	1
+	0	0	1	1	1	0	1	1
	1	1	1	1	0	0	0	0

`-75+59==-16`

correct

`181+92==-16`

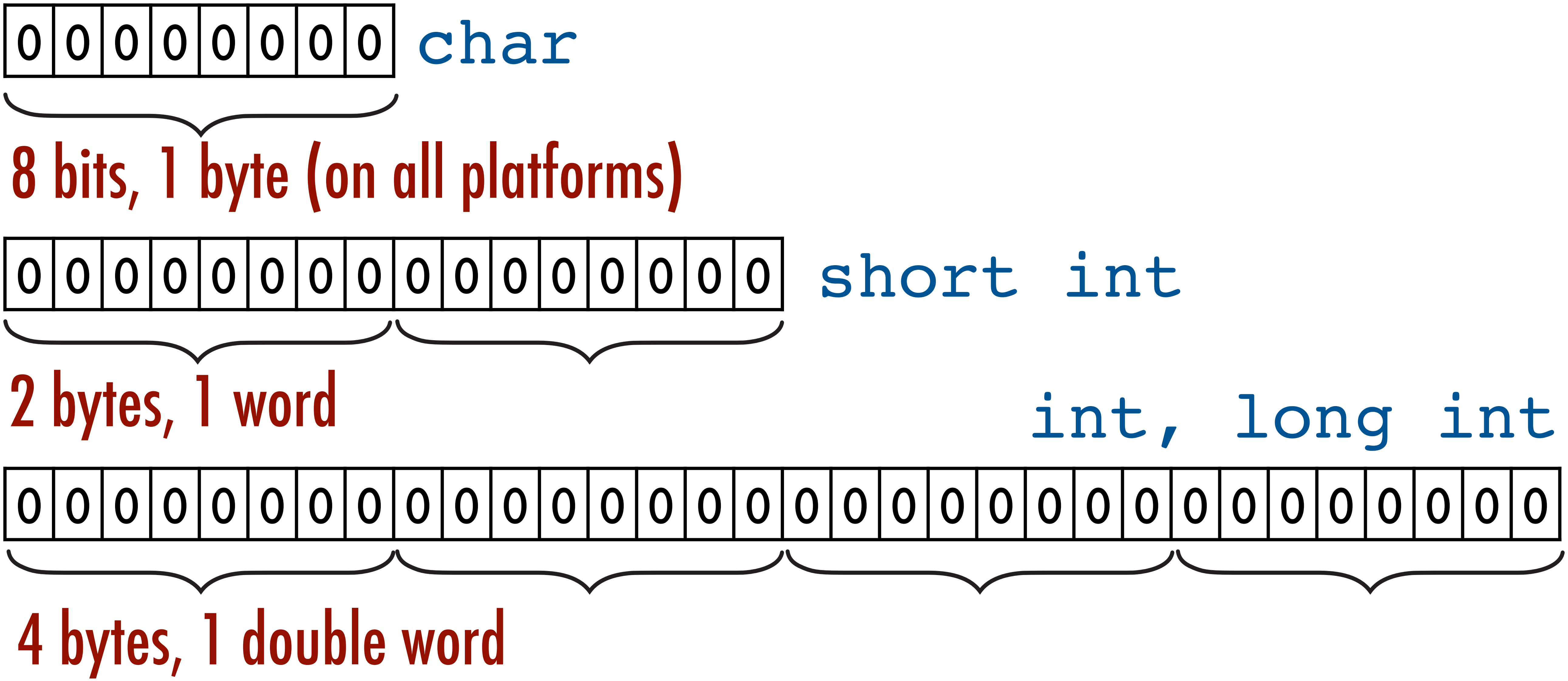
overflow

Why two’s complement?

- Straightforward to detect overflow:
  - If the sum of two positive numbers yields a negative result, the sum has overflowed
  - If the sum of two negative numbers yields a positive result, the sum has overflowed
- In two's complement, carry out does not indicate an overflow condition

C++ integer types

- Only relative sizes guaranteed; on the Intel architecture . . .



0 0 0 0 0 0 0 0 `char`

8 bits, 1 byte (on all platforms)

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 `short int`

2 bytes, 1 word

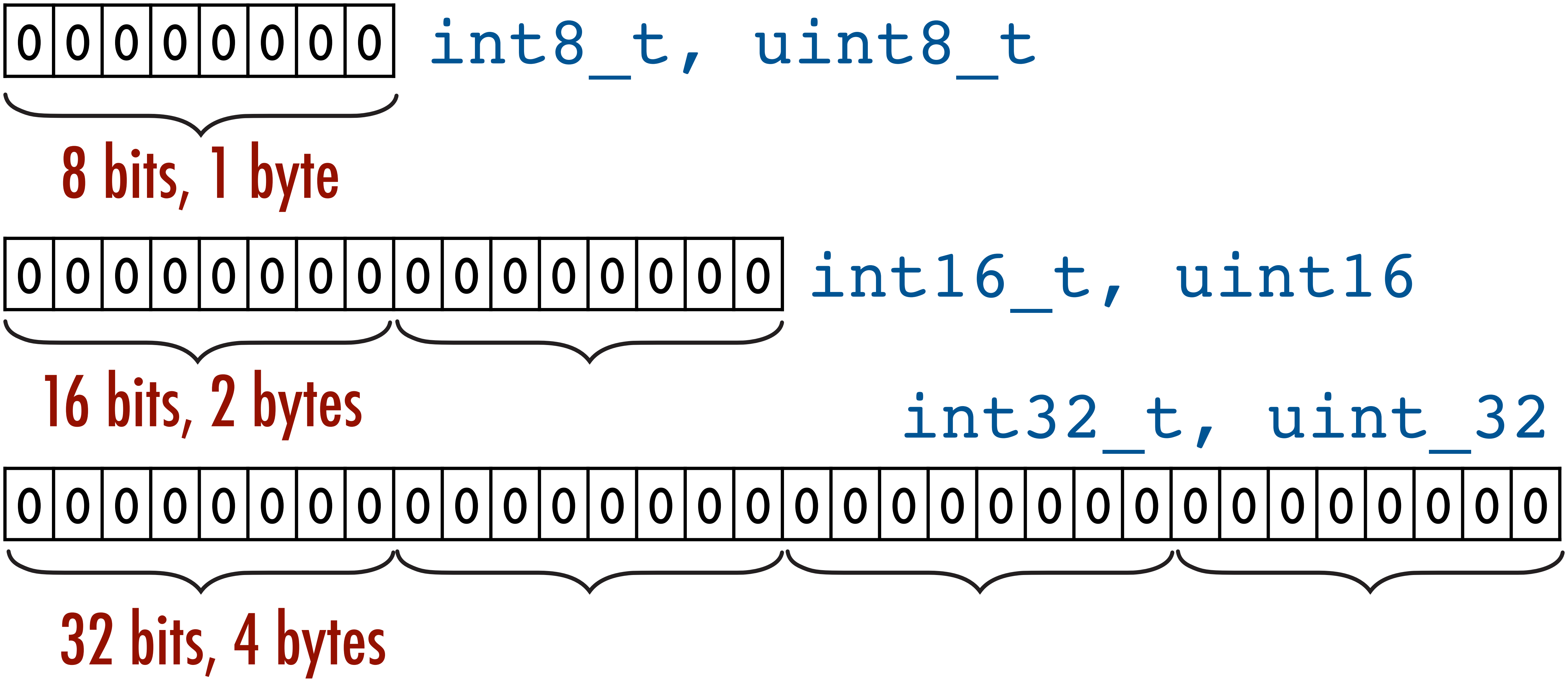
`int, long int`

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

4 bytes, 1 double word

C++ integer types

- Types of fixed width available in C++ `#include <stdint>`



0 0 0 0 0 0 0 0 `int8_t, uint8_t`

8 bits, 1 byte

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 `int16_t, uint16`

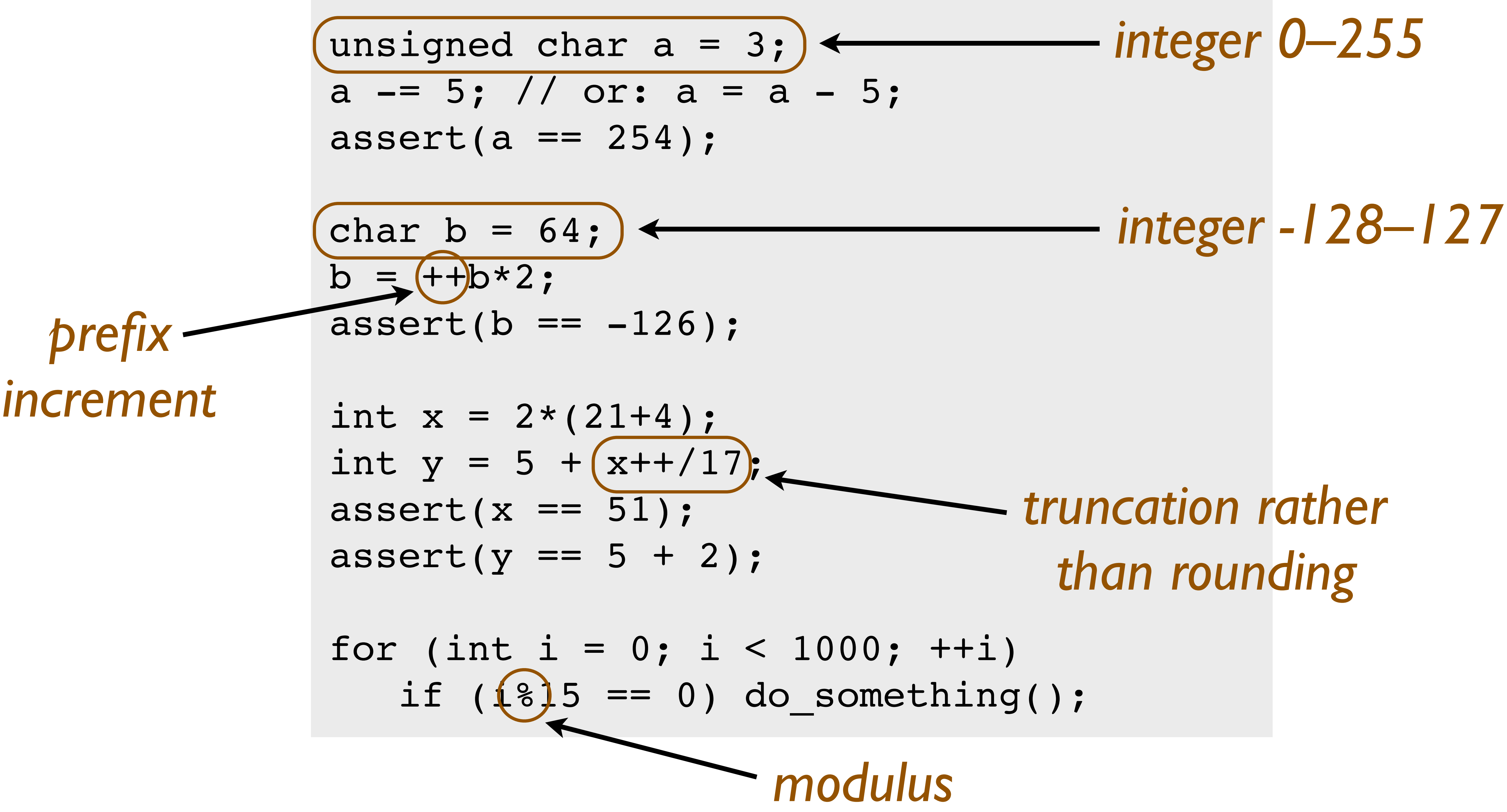
16 bits, 2 bytes

`int32_t, uint_32`

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

32 bits, 4 bytes

Arithmetic operators



```
unsigned char a = 3;
a -= 5; // or: a = a - 5;
assert(a == 254);

char b = 64;
b = ++b*2;
assert(b == -126);

int x = 2*(21+4);
int y = 5 + x++/17;
assert(x == 51);
assert(y == 5 + 2);

for (int i = 0; i < 1000; ++i)
    if (i%15 == 0) do_something();
```

integer 0–255

integer -128–127

prefix increment

truncation rather than rounding

modulus

Bitwise operators

enumerated type

```
enum directions { N = 1, E = 2, S = 4, W = 8 };
const uint8_t opt1 = 020;   //  2*8 == 16
const uint8_t opt2 = 0x20;  // 2*16 == 32

unsigned char flags = N | W;
assert( (flags & N) and (flags & W) );

flags |= S | E;
assert( flags == N | S | E | W );

flags &= ~S;
assert( flags == N | E | W );

flags ^= N | E | opt1;
assert( flags == W | opt1 );
flags ^= opt1 | opt2;
assert( !(flags & opt1) and (flags & opt2) );
```

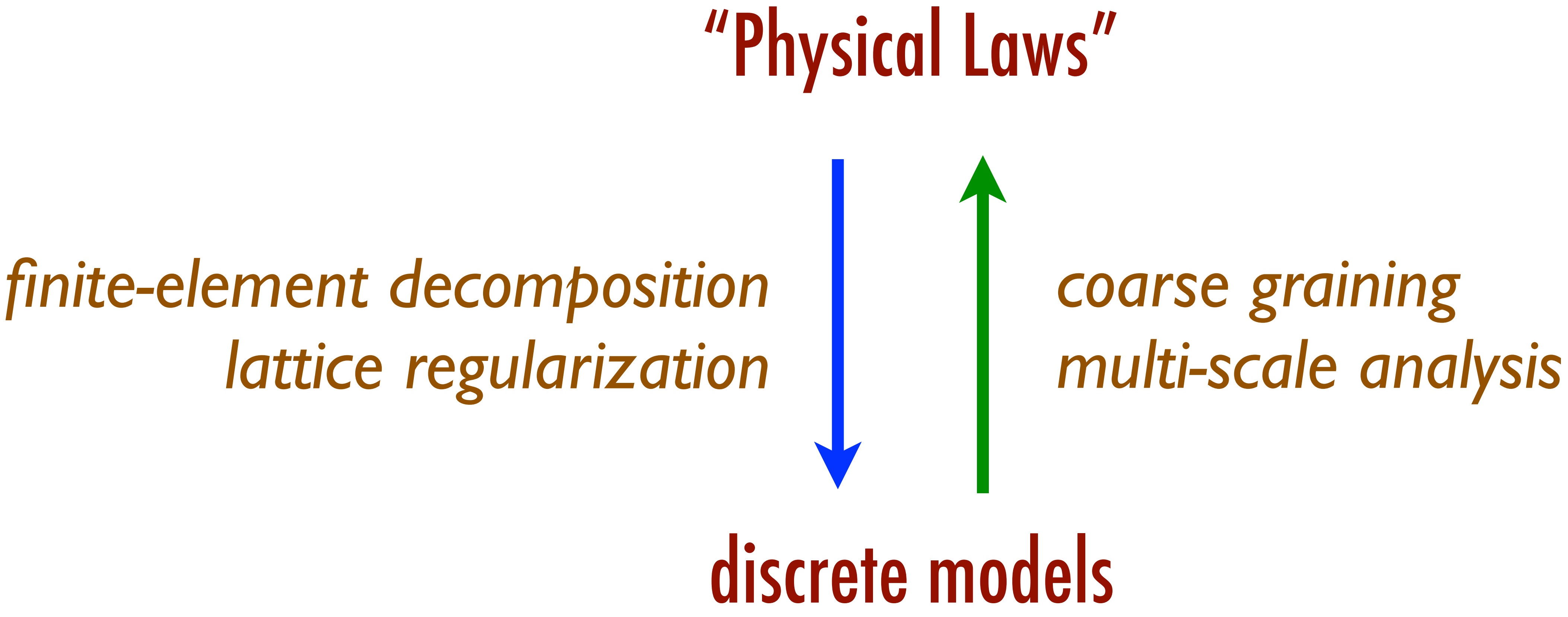
octal and hex

test bits

set,
clear,
and toggle
bits

Discretization

- Computers cannot naturally handle continuous properties
- But the granularity of a simulation is not apparent on sufficiently long length scales



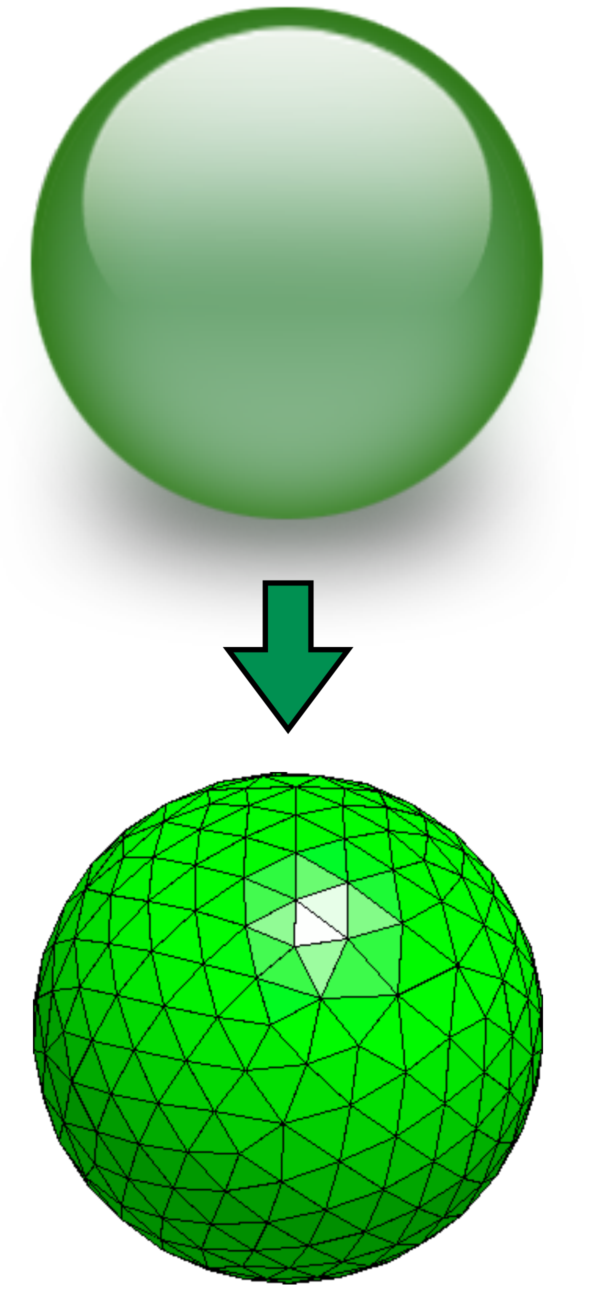
“Physical Laws”

finite-element decomposition
lattice regularization

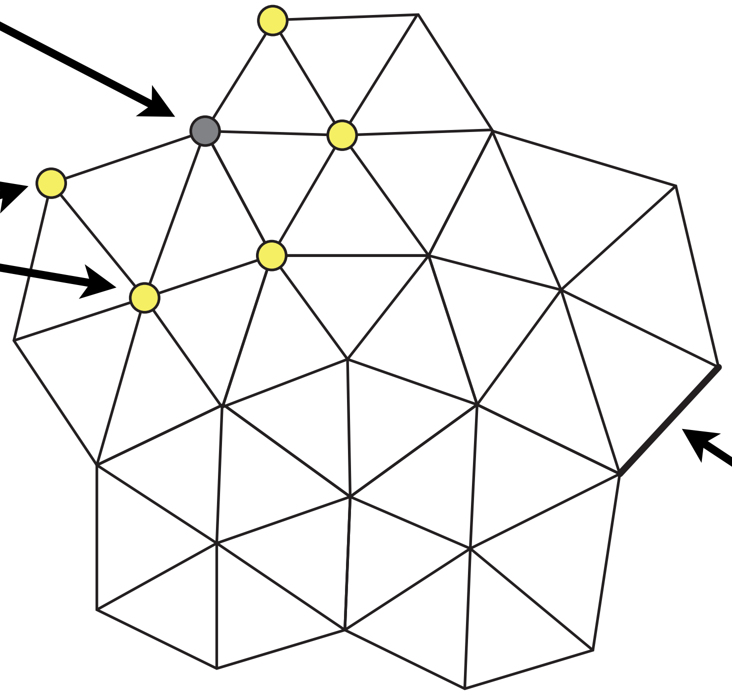
coarse graining
multi-scale analysis

discrete models

Spatial grids



- Replace continuous manifold by a mesh of points
- Topology is encoded by their connectivity



vertex = "site"

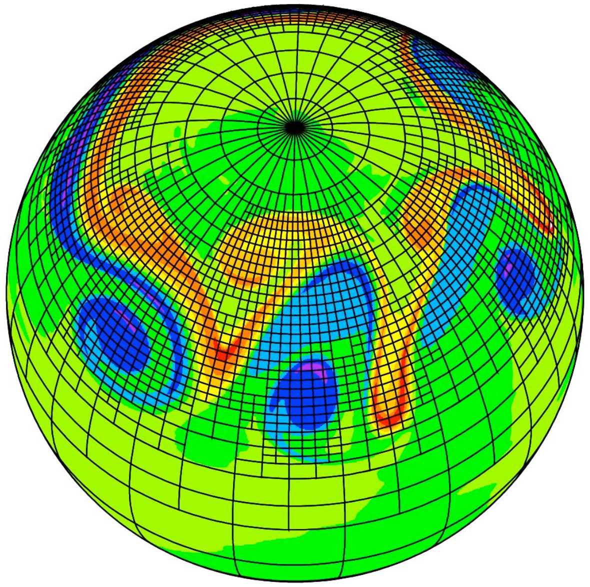
"neighbours"

graph edge = "link"
or "bond"

Adaptive mesh

- Inhomogeneous arrangement of points
- optimized so that their local density and connectivity track some key physical property
- the presence of many different length scales provides a hierarchy of resolutions
- grid may be static or dynamic; the latter sometimes offers big savings in storage and computational effort

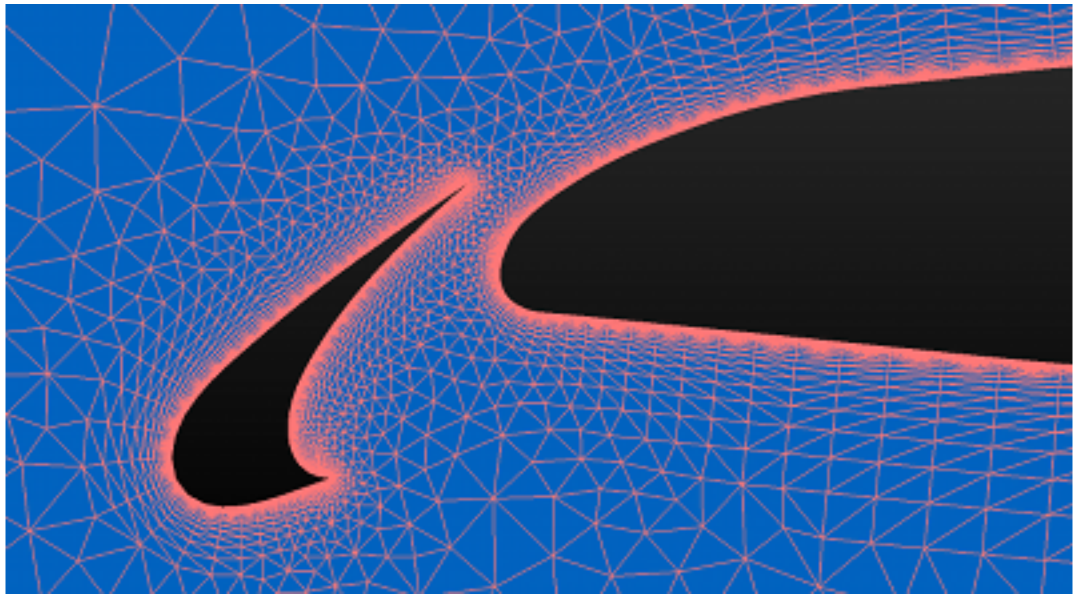
Adaptive mesh

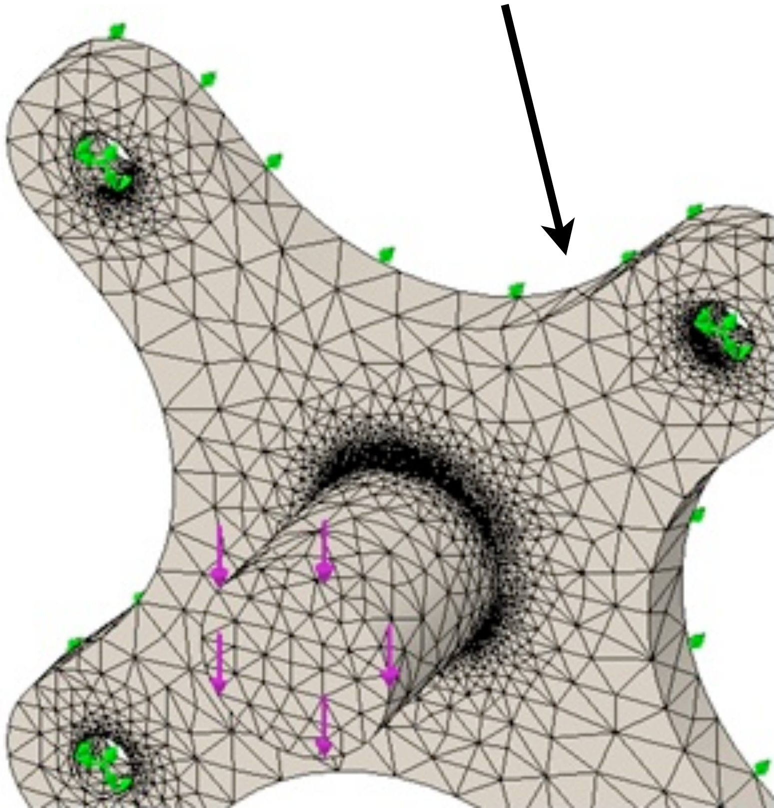


modelling of ocean currents with the grid resolution tied to the local vorticity field; recursive, squares-within-squares geometry

materials modelling with small finite elements at points of high stress

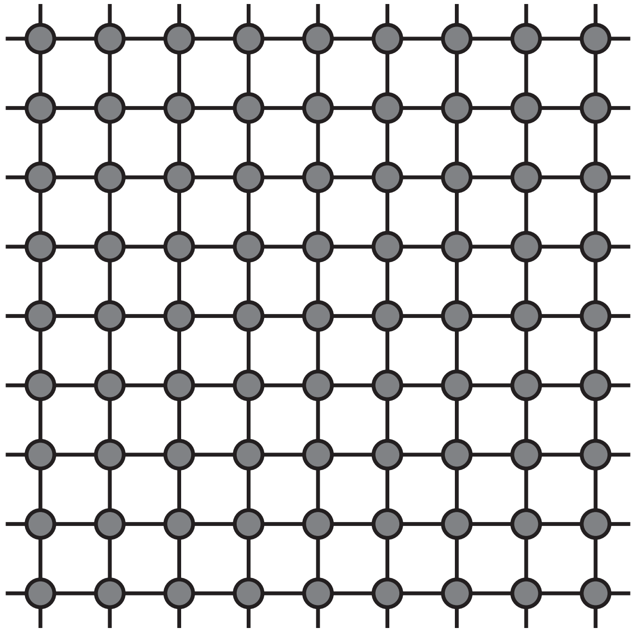
mesh of triangles whose area is inversely proportional to the speed of air flow around an aerofoil





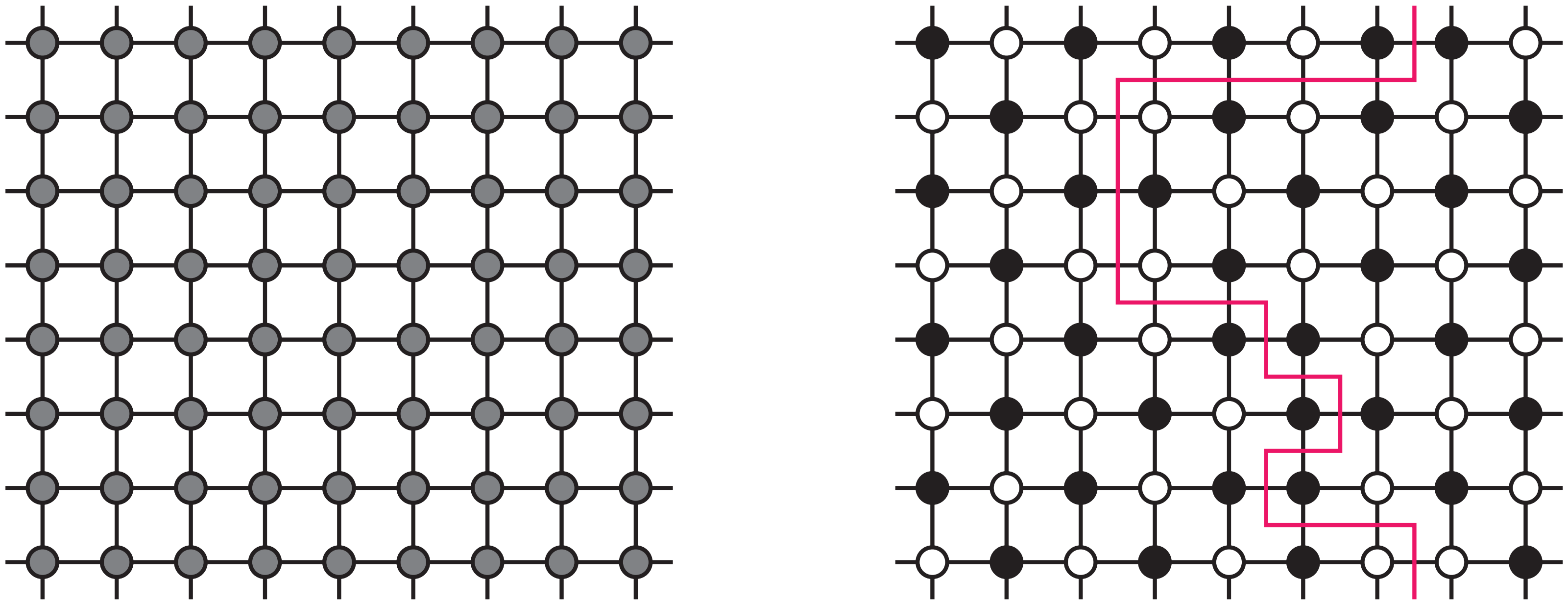
Lattices

- Uniform mesh of infinite repeating units related to an underlying Bravais lattice
- Exhibits definite space and point group symmetries (translation, rotation, reflection, ...)
- How to connect boundary sites in a finite sample?
- Can the symmetries be preserved?



Lattices

- Compatibility of boundary conditions with ordered states
- Possibility of even/odd effects



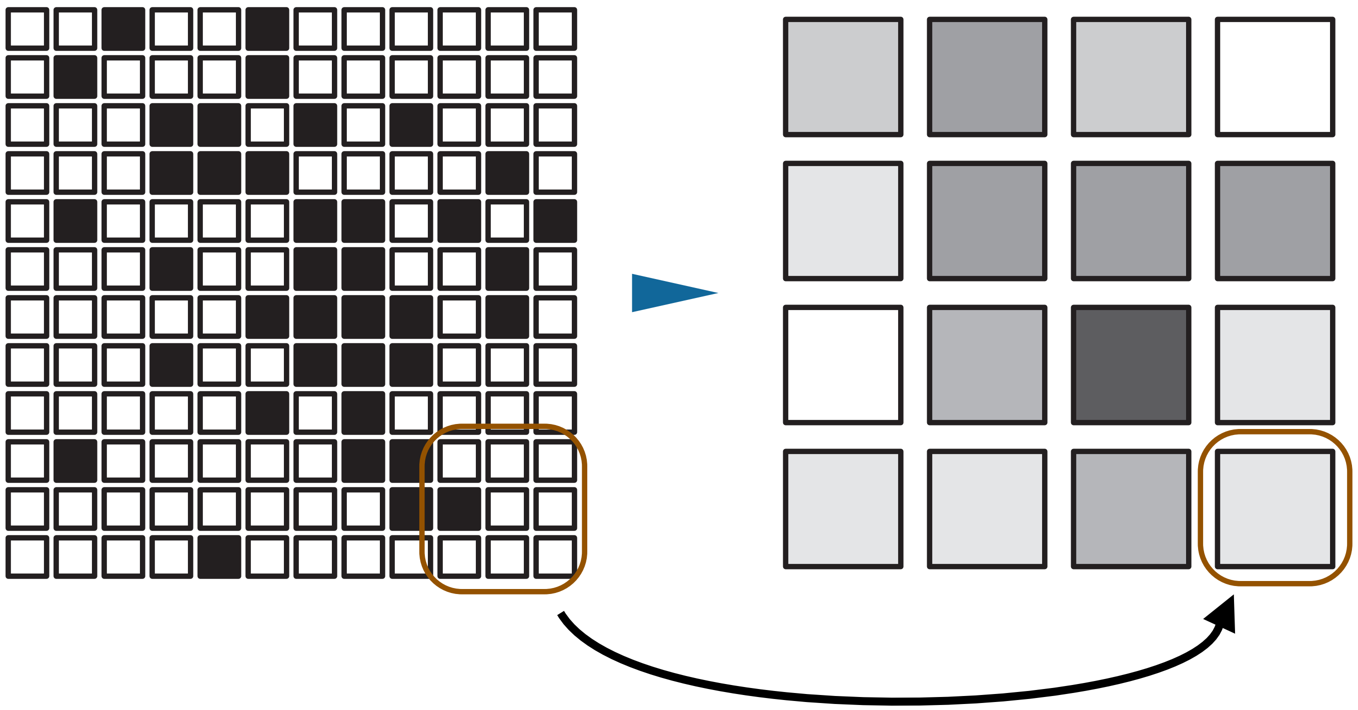
$L = 2n$

$L = 2n + 1$

AFM order with a line of mismatches

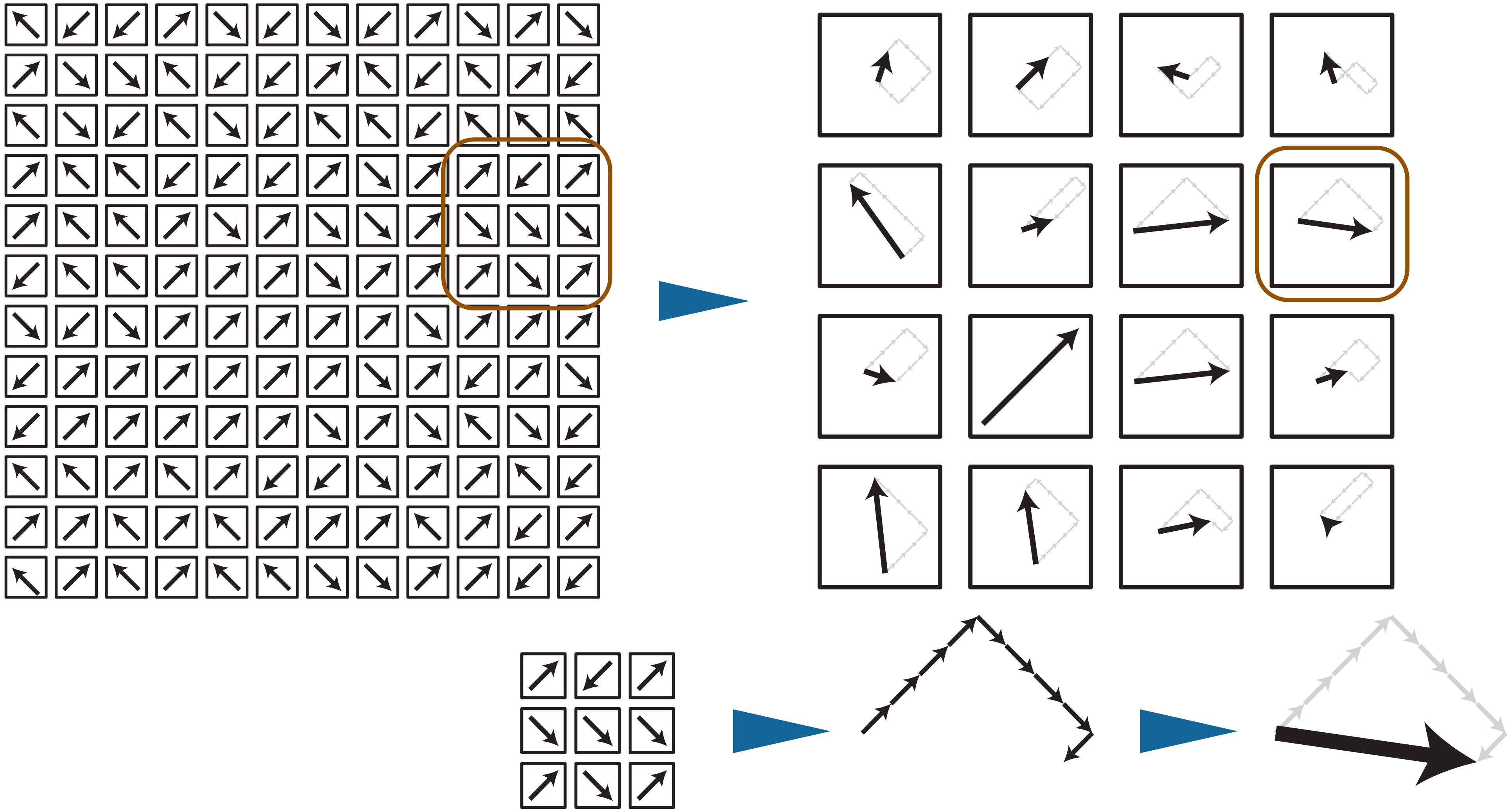
Coarse graining

- Process of spatial averaging over local regions
- E.g., 3x3 averaging of binary cells gives distinct 10 levels
- Recover continuous scalar field in large region limit

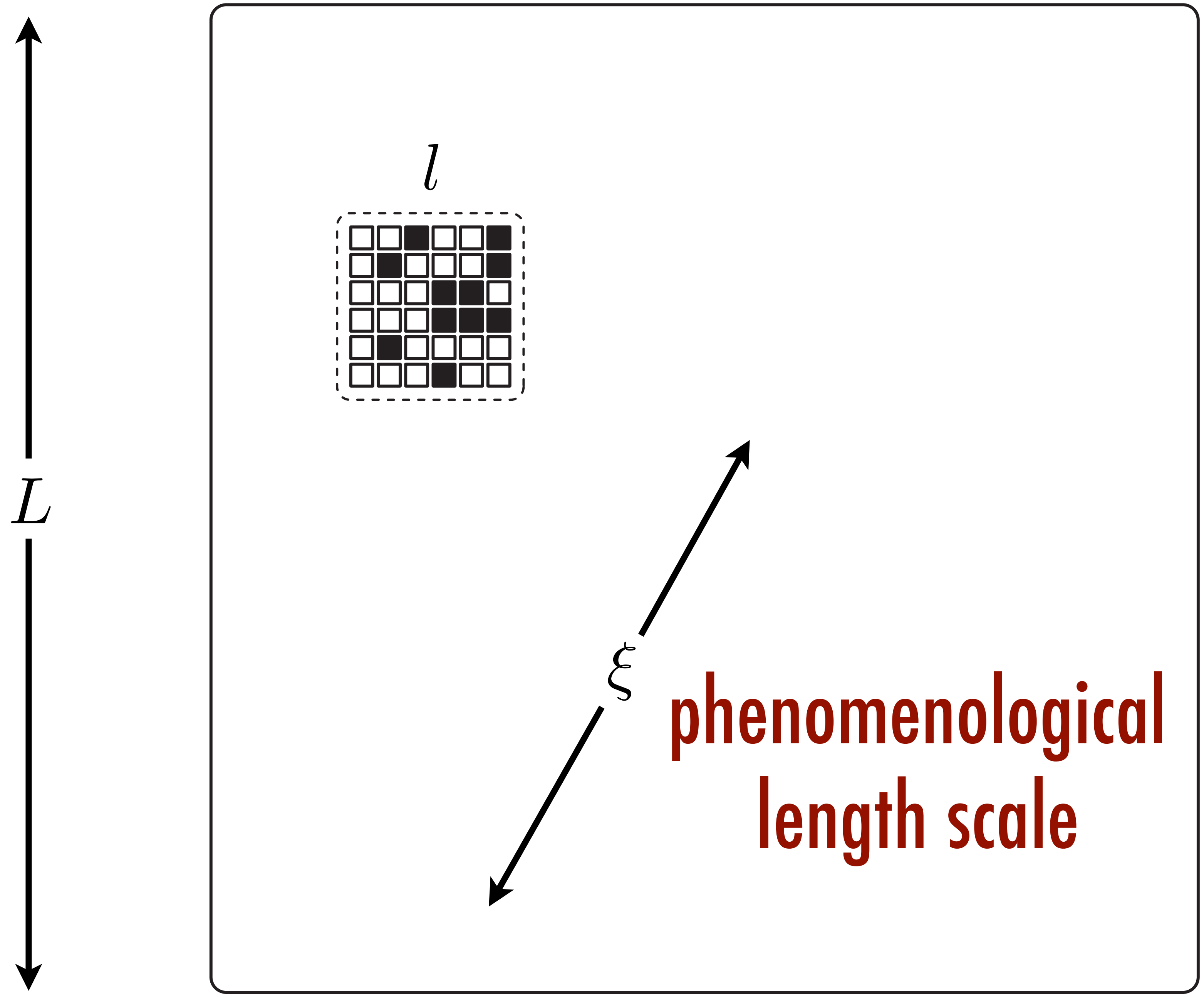


Coarse graining

- E.g., 3x3 averaging of 4-state clocks
- Recover continuous vector field in large region limit



Hierarchy of length scales



sample size

phenomenological length scale

key requirement:

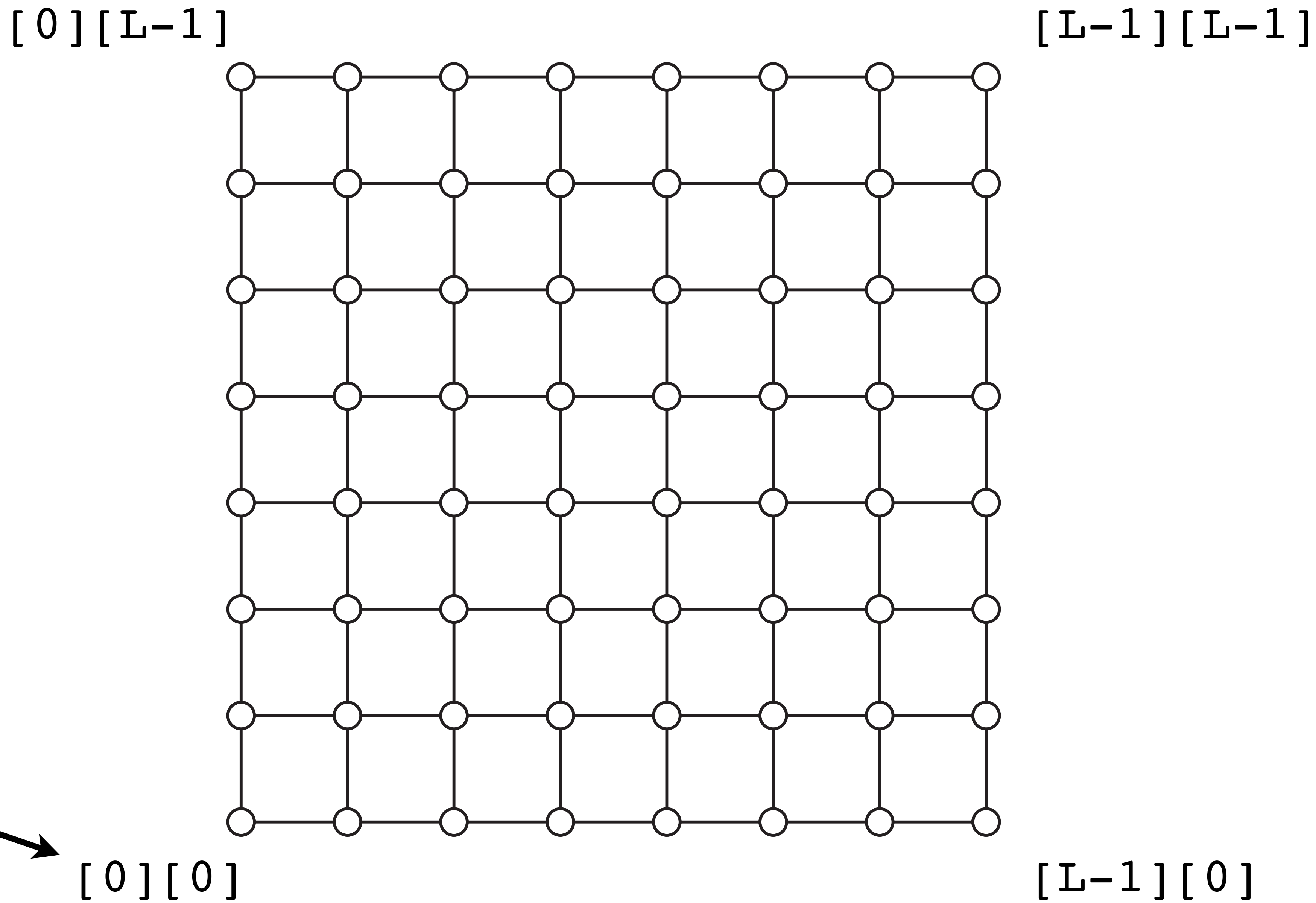
$$l \ll \xi \ll L$$

Spatial data structures

- Associate properties with each site (or link) of a lattice
- Encode some sense of which sites are neighbours
- For hypercubic lattices, the setup is trivial with C arrays:

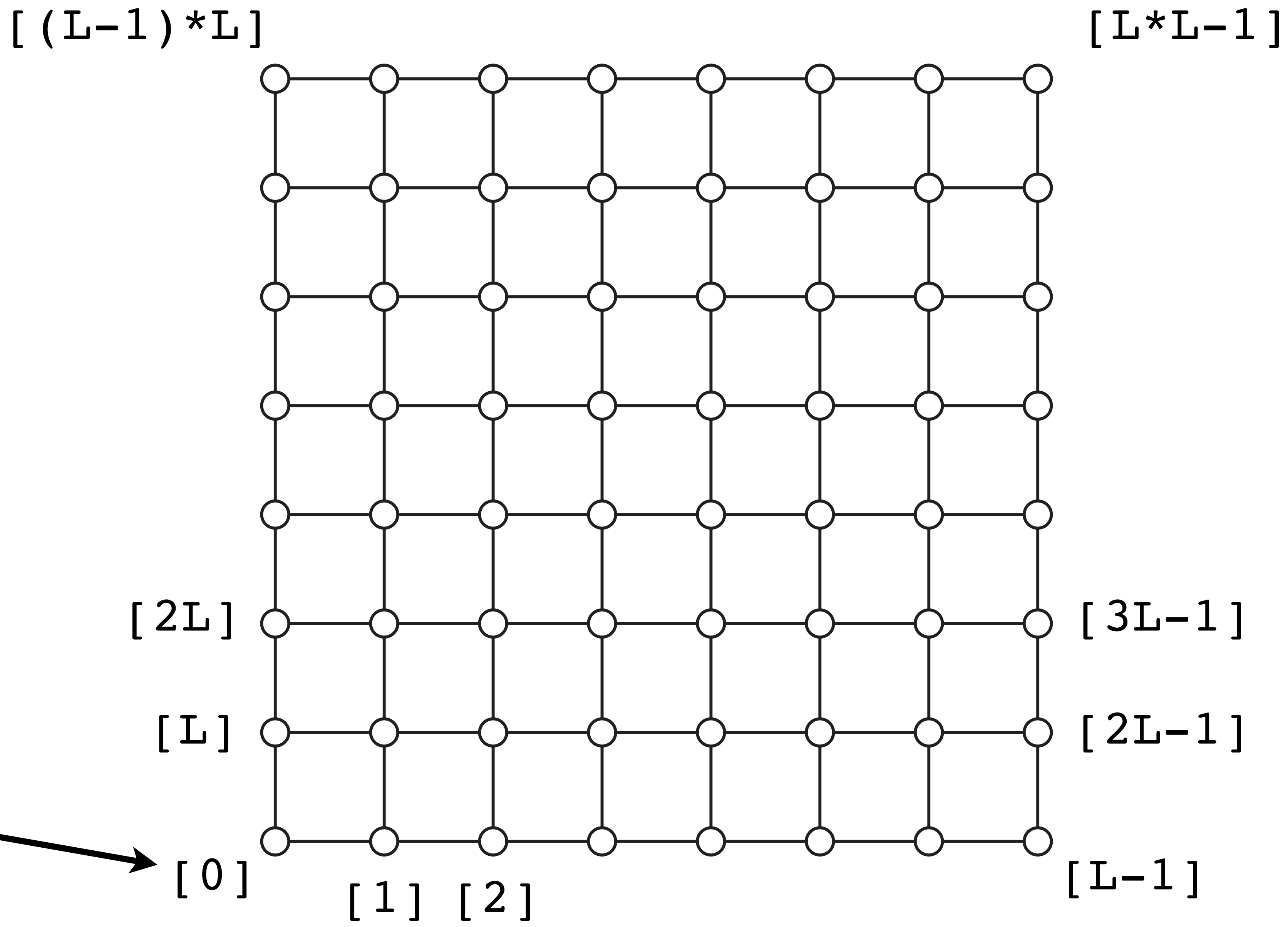
```
// square lattice as 2D array
int lattice[100][100];
lattice[60][99] = 0;

// square lattice as 1D array
int lattice[100*100];
inline int index(int i, int j)
{ return i+j*L; }
lattice[index(60,99)] = 0;
```

Square lattice



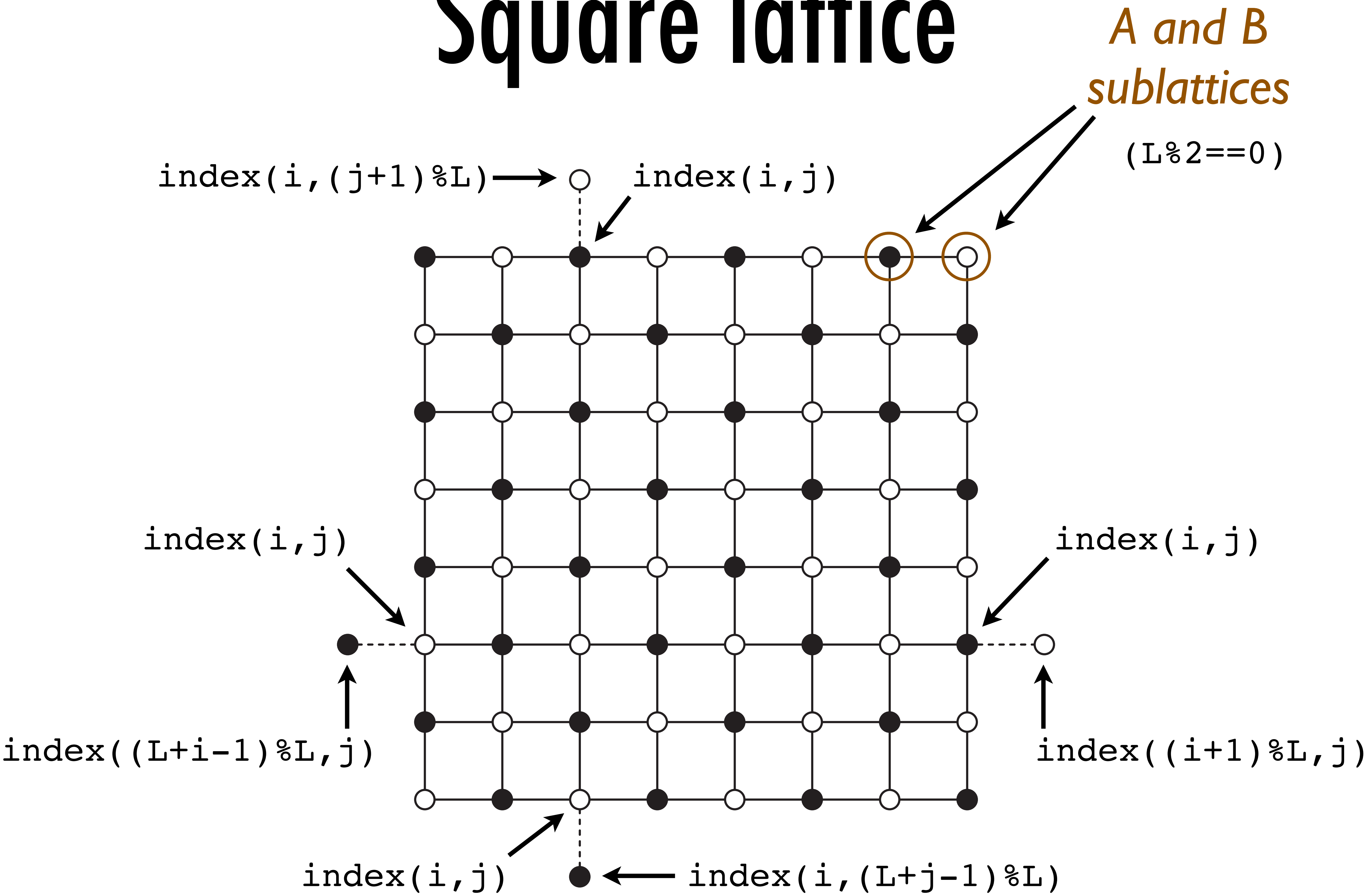
matrix-style storage

Square lattice



row-major storage

Square lattice

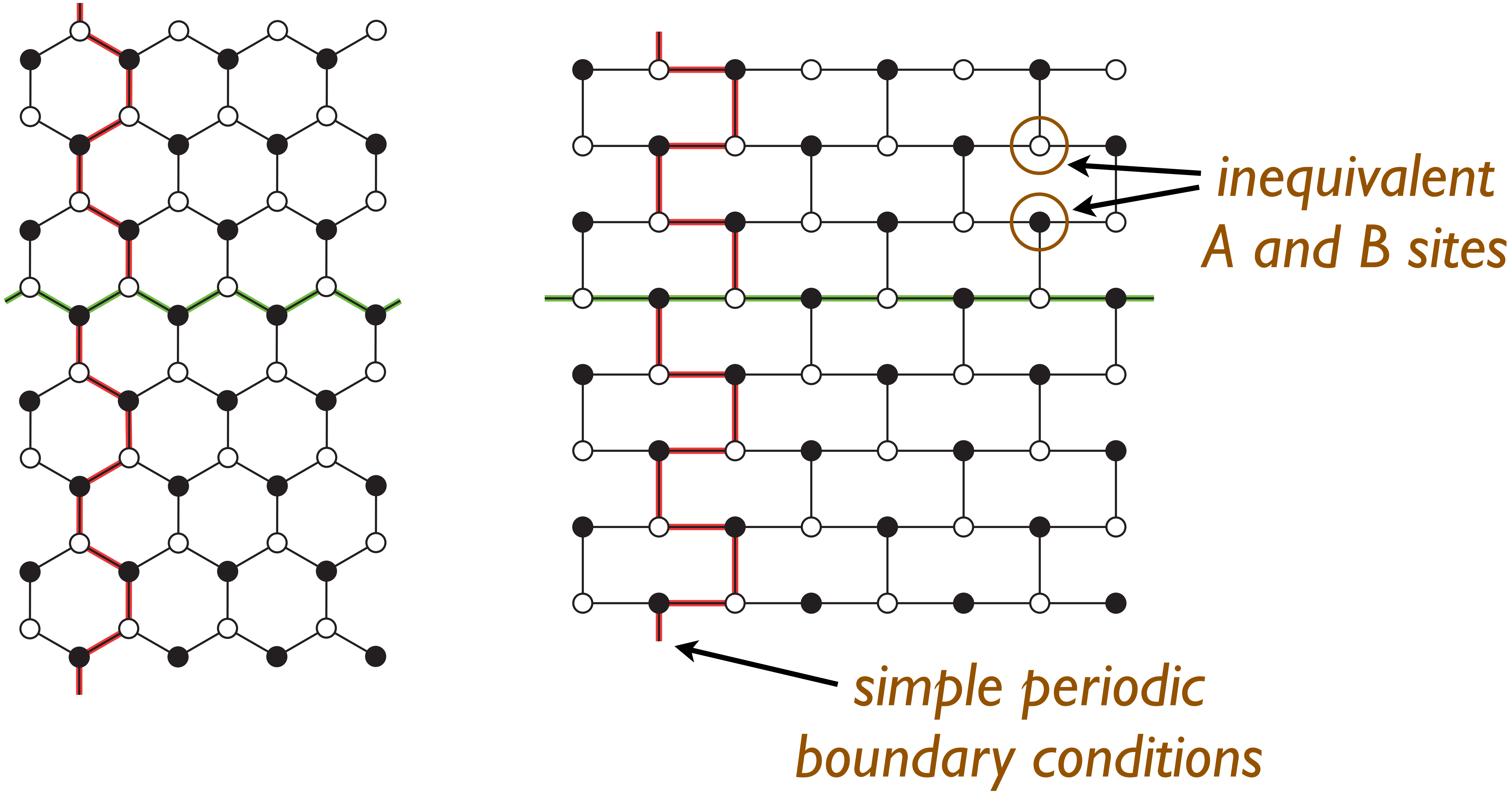


Square lattice class

```
#include <vector>
using std::vector;

template <typename T>
class square_lattice
{
private:
   const int L; vector<T> data;
public:
   square_lattice(int L_) : L(L_), data(L*L) {}
   T& operator()(int i, int j) { return data[i+j*L]; }
   int length(void) { return L; }
};

struct cell { int speciesA, speciesB; };
square_lattice<cell> lattice(20);
for (int i = 0; i < lattice.length(); ++i)
   lattice(4,i).speciesA = 3;
```

Honeycomb lattice

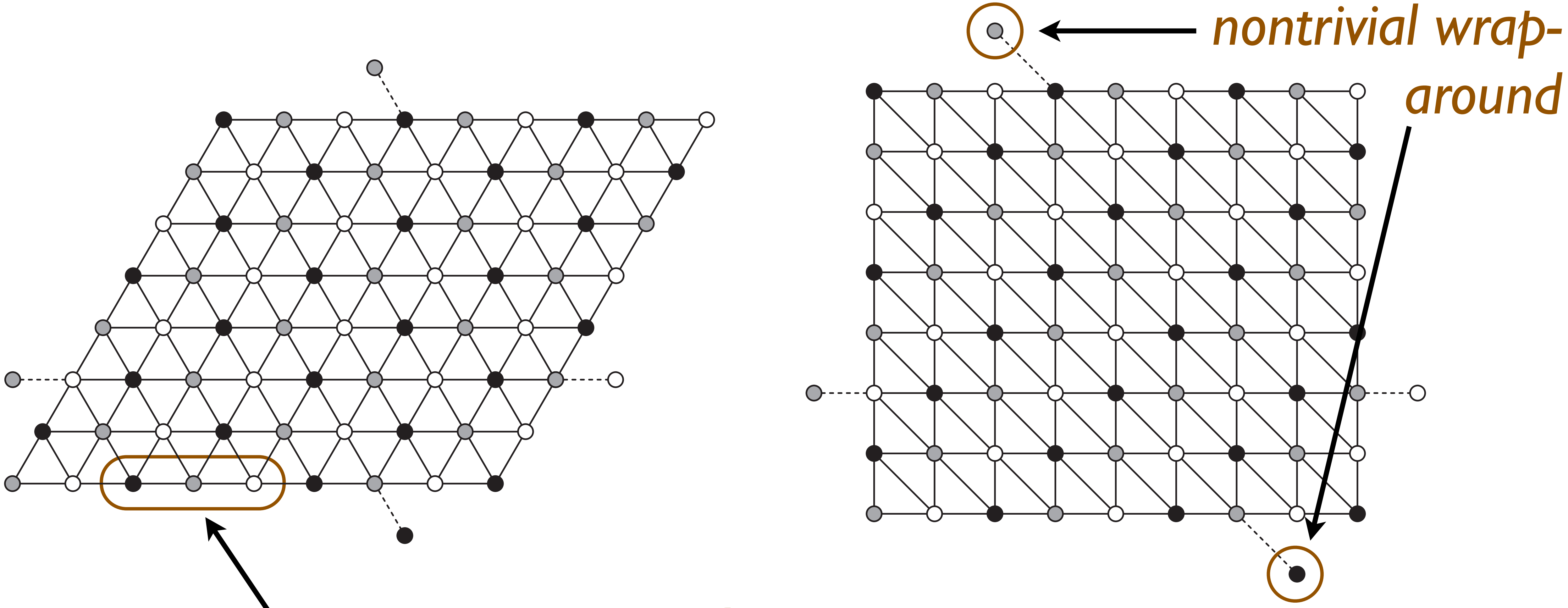
- Topology preserved when distorted to a brick-wall lattice


inequivalent A and B sites

simple periodic boundary conditions

Triangular lattice

- Topology preserved when sheared to orthogonal axes



nontrivial wrap-around

tripartite: A, B, and C sublattices

`(Lx == Ly+1)`

Kagomé lattice

- Further deplete the triangular lattice