Algorithms for GIS
csci3225

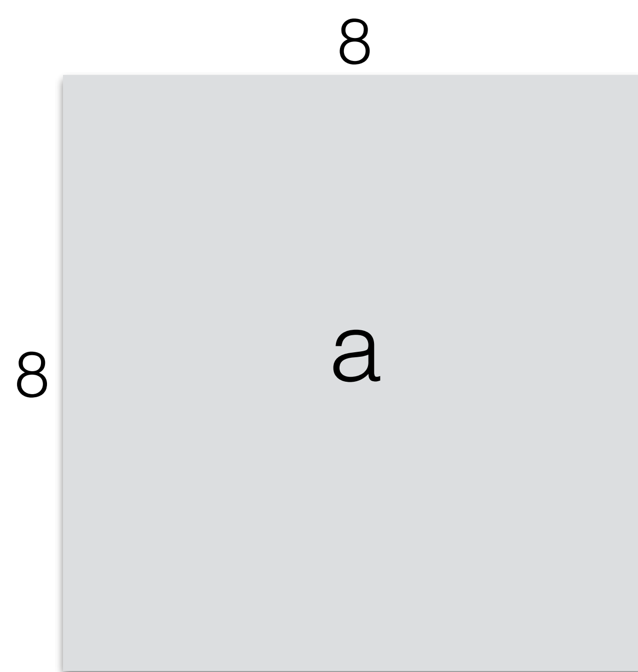
Laura Toma

Bowdoin College

COB multiplication, matrix layout and space-filling curves

Matrix layout

Matrix a is given in row-major order



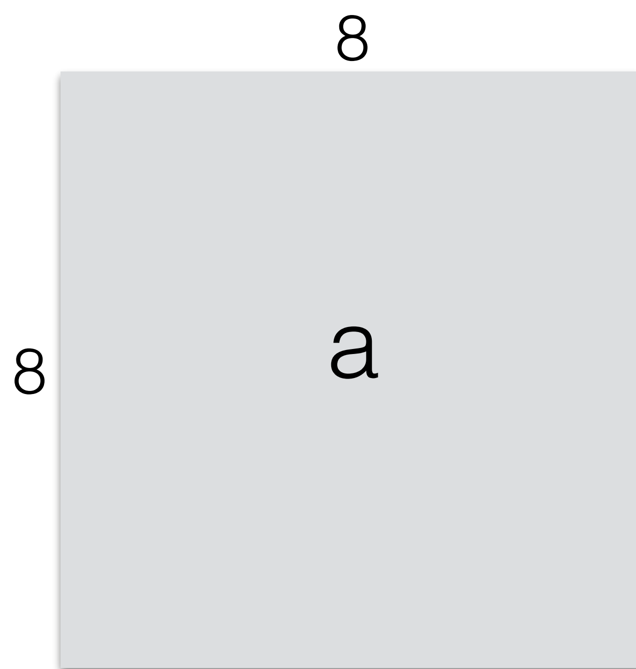
row-major order

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	59	51	52	53	54	55
56	57	58	59	60	61	62	63

a	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31	32 33 34 35 36 37 38 39	40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55	56 57 58 59 60 61 62 63

Matrix layout

Matrix a is given in row-major order



row-major order

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	59	51	52	53	54	55
56	57	58	59	60	61	62	63

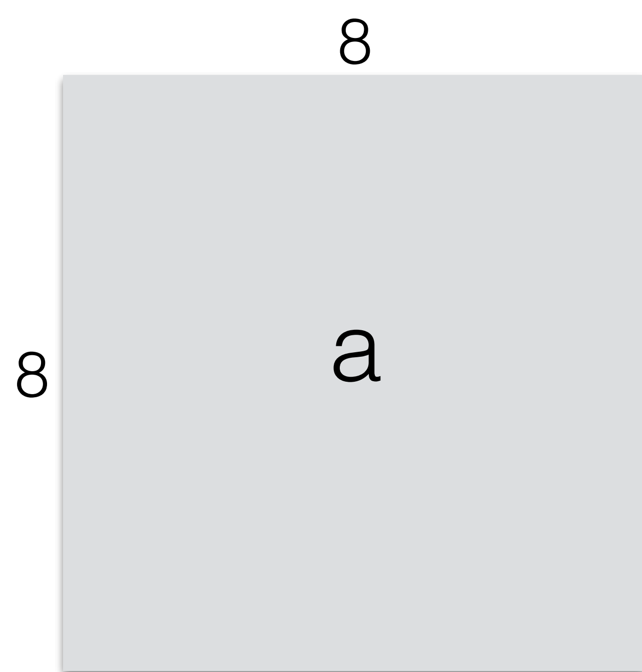
a

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31	32 33 34 35 36 37 38 39	40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55	56 57 58 59 60 61 62 63

Highlight the elements of a_{11} in a

Matrix layout

Matrix a is given in row-major order



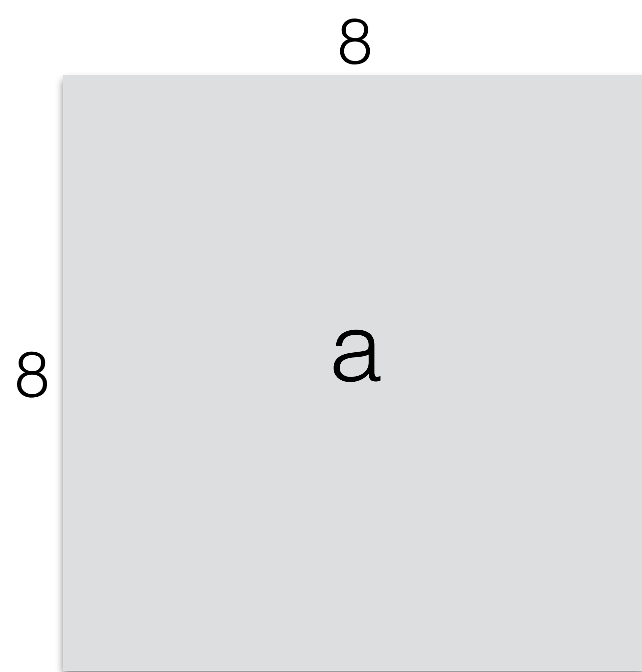
row-major order

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	59	51	52	53	54	55
56	57	58	59	60	61	62	63

a	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31	32 33 34 35 36 37 38 39	40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55	56 57 58 59 60 61 62 63

Matrix layout

Matrix a is given in row-major order



row-major order

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	59	51	52	53	54	55
56	57	58	59	60	61	62	63

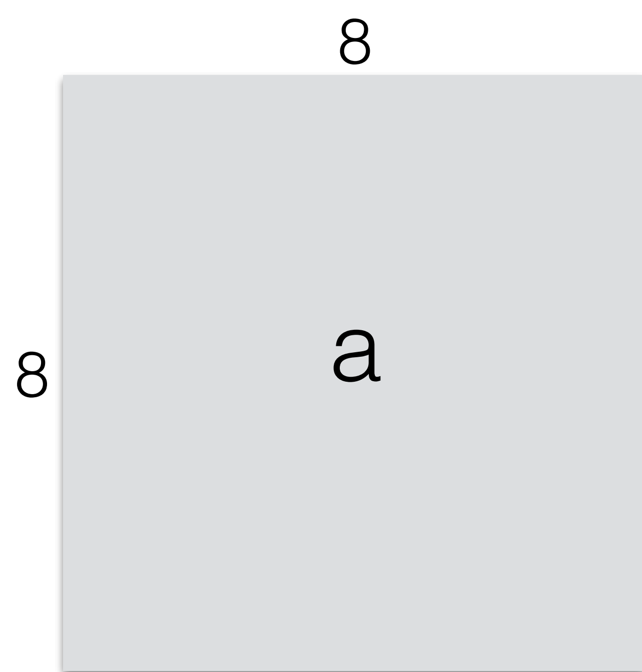
Assume block size is 3

a | 0 1 2 3 4 5 6 7 | 8 9 10 11 12 13 14 15 | 16 17 18 19 20 21 22 23 | 24 25 26 27 28 29 30 31 | 32 33 34 35 36 37 38 39 | 40 41 42 43 44 45 46 47 | 48 49 50 51 52 53 54 55 | 56 57 58 59 60 61 62 63 |

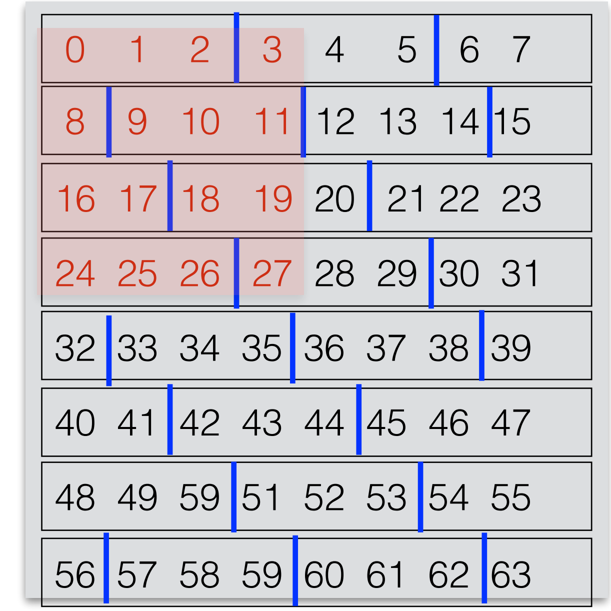
How many blocks span a_{11}?

Matrix layout

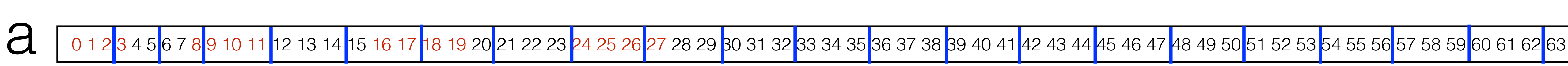
Matrix a is given in row-major order



row-major order



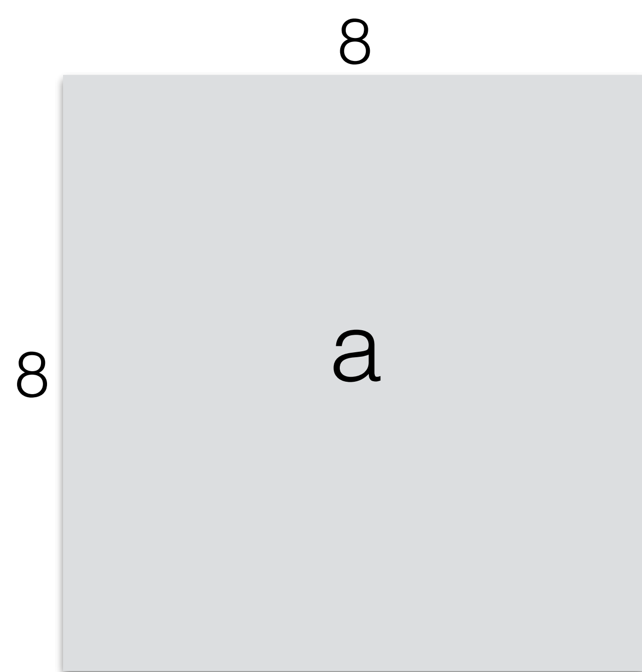
Assume block size is 3



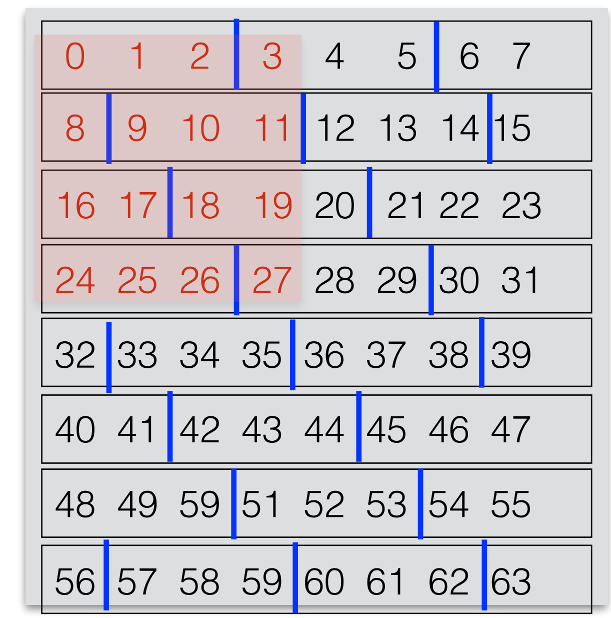
How many blocks span a_{11}?

Matrix layout

Matrix a is given in row-major order

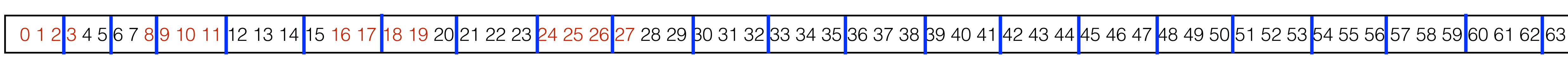


row-major order



Assume block size is 3

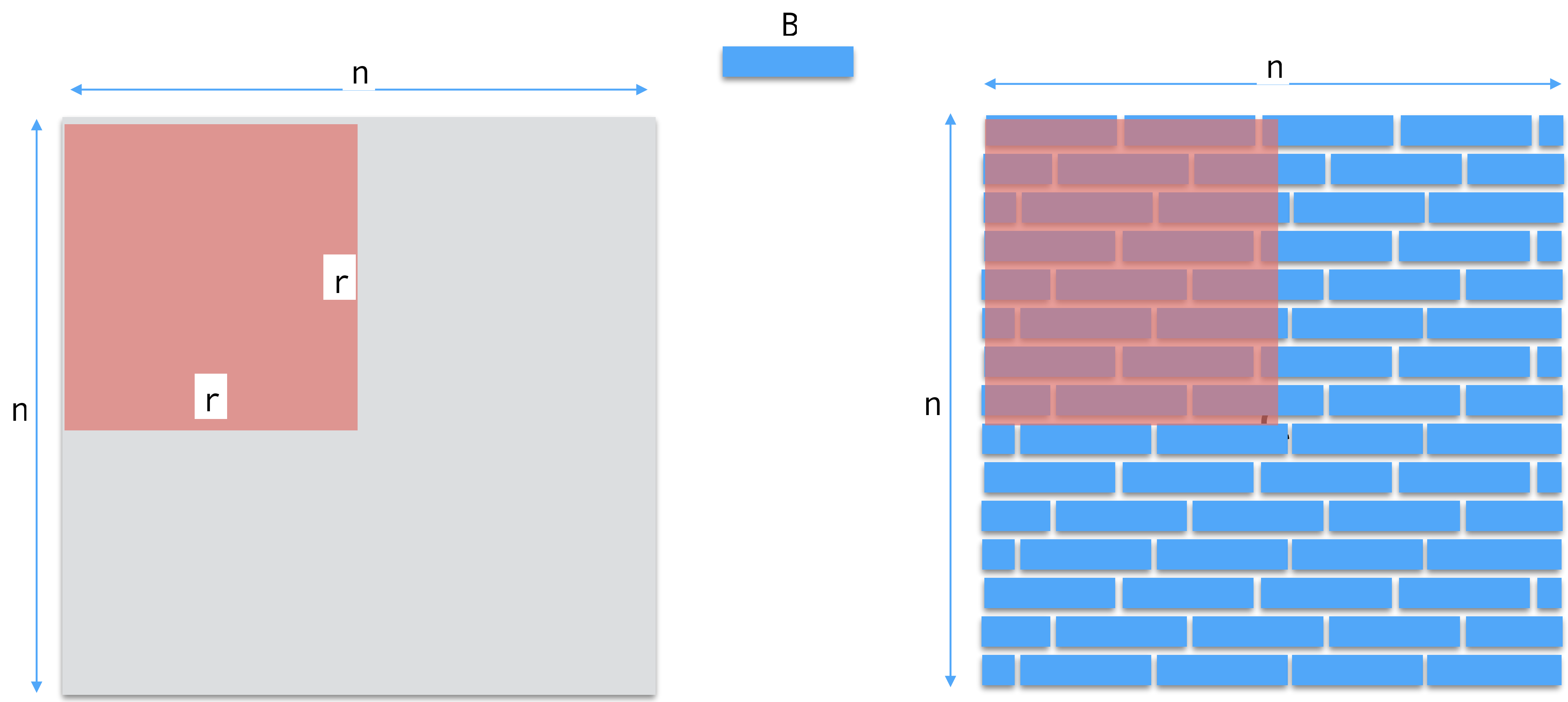
a



How many blocks span a_{11}?

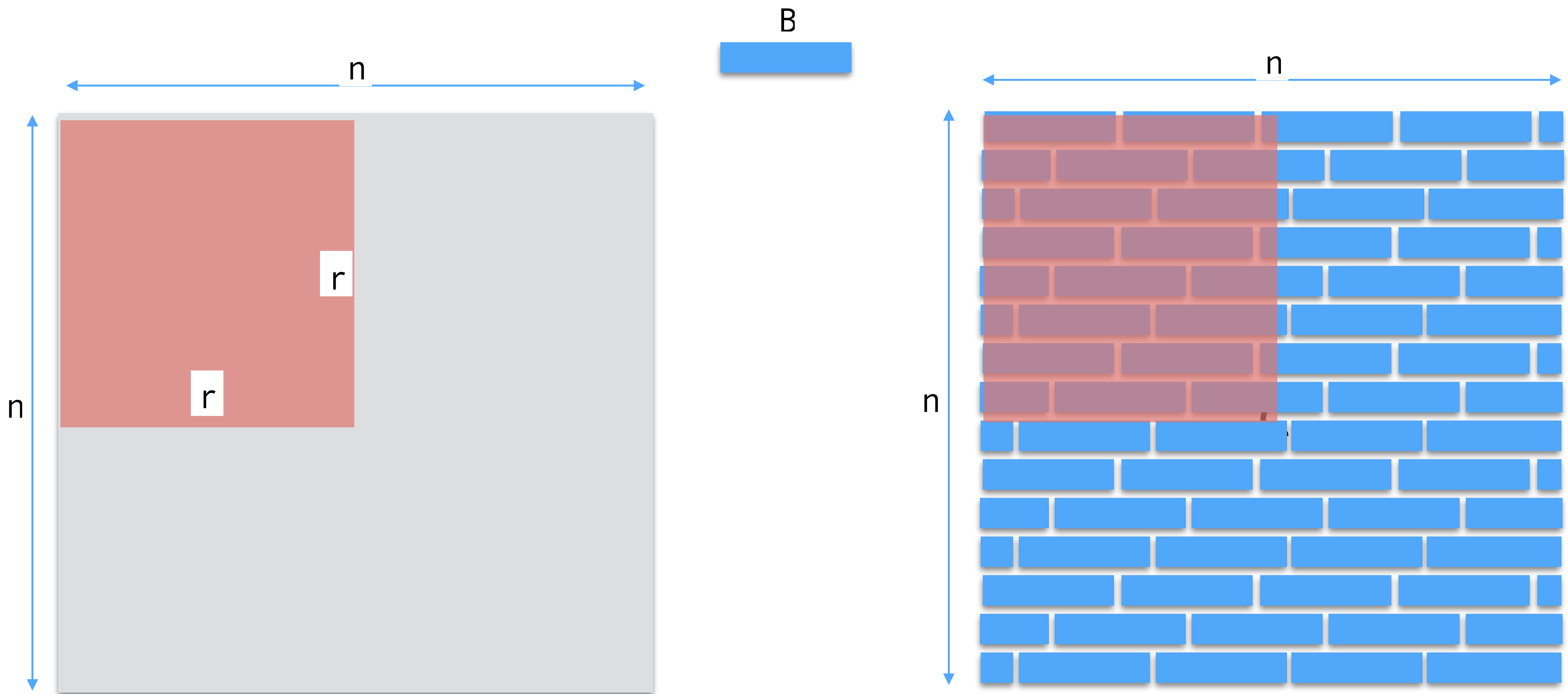
Ideally, 6.
In this case, 8.

In general



How many cache misses to read a block of size r-by-r, in a matrix laid out in row-major order?

In general

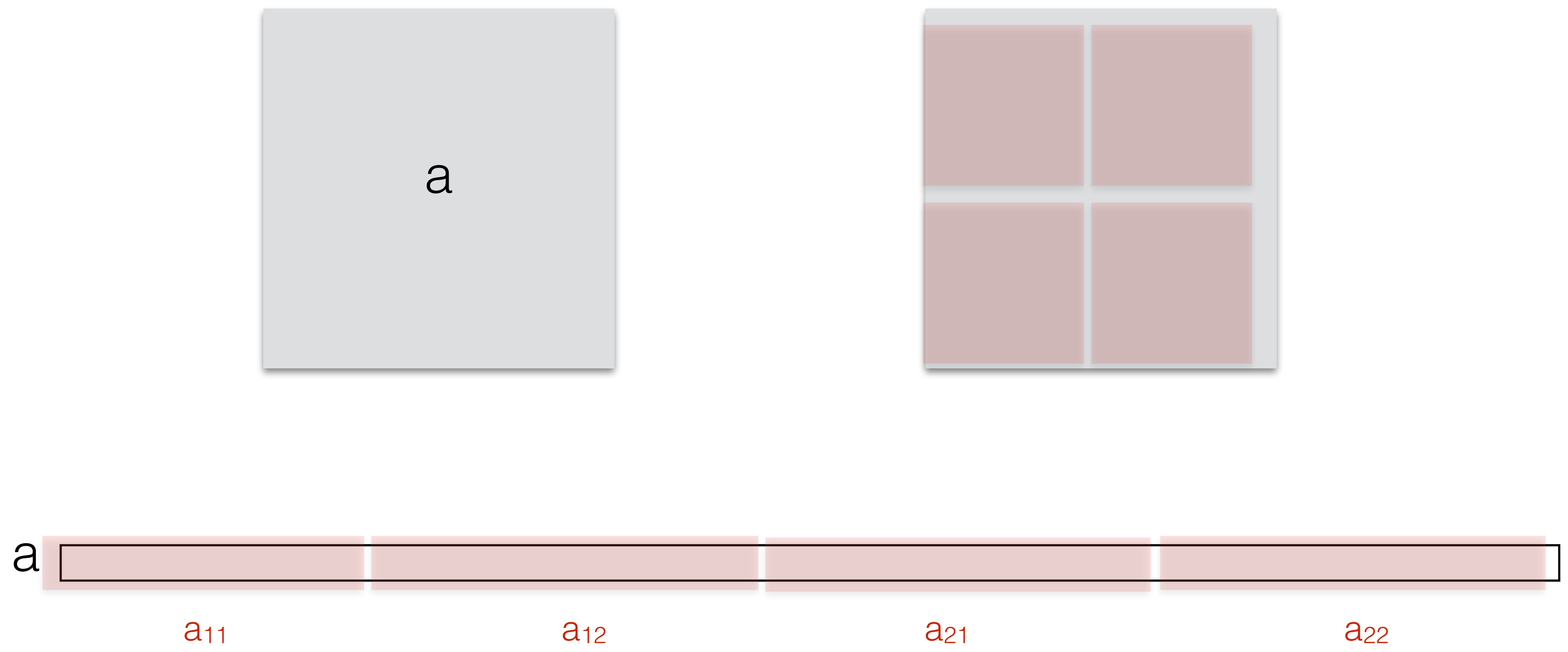


How many cache misses to read a block of size r-by-r, in a matrix laid out in row-major order?

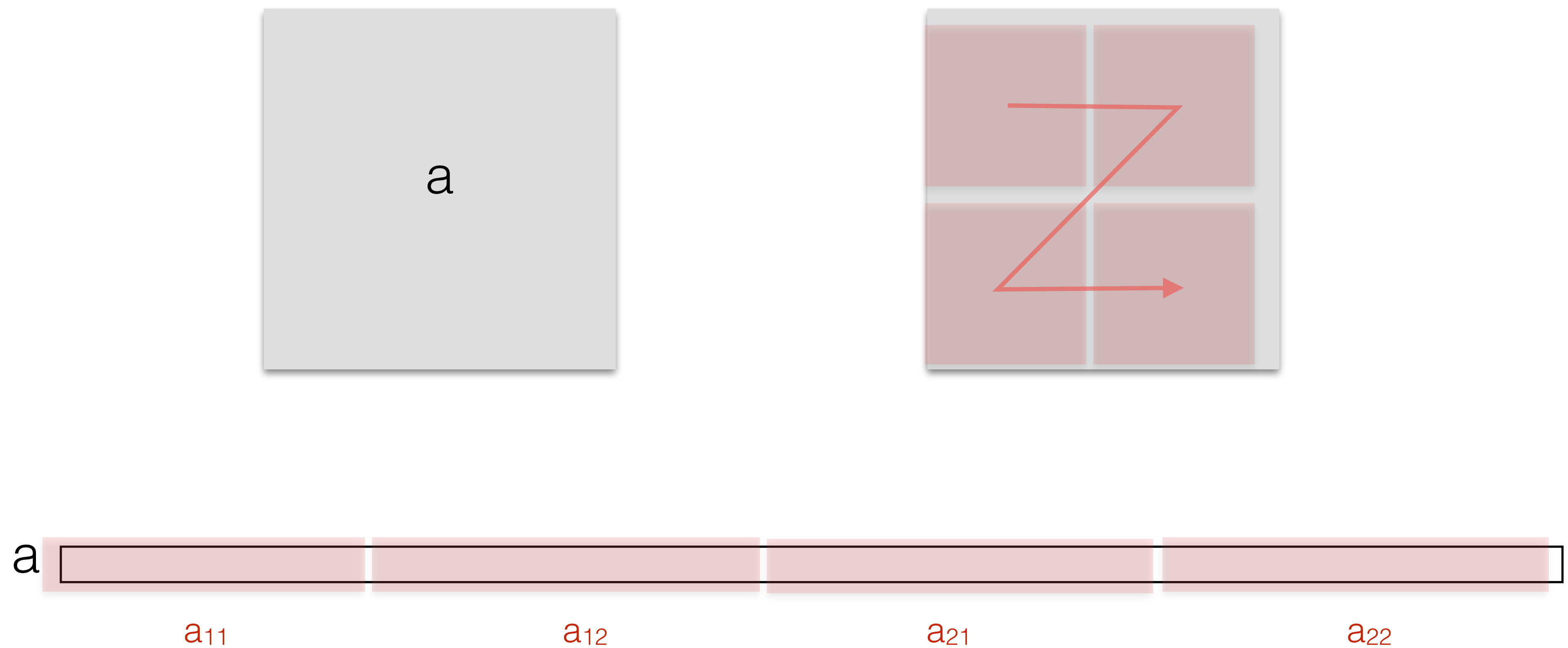
`cache-misses: O(r²/B + r)`

this term is because a_{11} is not contiguous in the row-major order of a

WHAT IF we laid out the matrix so that each quadrant is stored contiguously.
(and this would be true recursively).



WHAT IF we laid out the matrix so that each quadrant is stored contiguously.
(and this would be true recursively).



What would this layout look for a 2-by-2 matrix?

What would this layout look for a 4-by-4 matrix?

…

2-by-2 matrix

row-major:

$$\begin{matrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{matrix}$$

0	1
2	3

a11 a12 a21 a22

the numbers represent the order in which we store the elements

z-order:

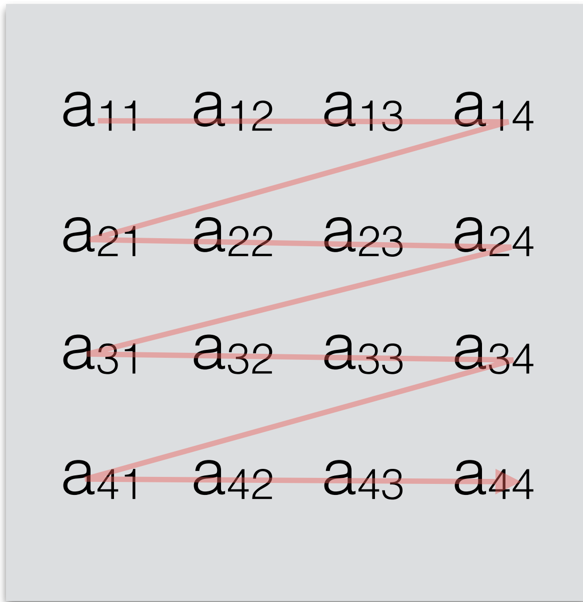
$$\begin{matrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{matrix}$$

0	1
2	3

a11 a12 a21 a22

4-by-4 matrix

row-major



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

a_{14} stored at index 3

z-order

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

a_{14} stored at index 5

the numbers represent the order in which we store the elements

a11 a12 a13 a14 a21 a22 a23 a24 a31 a32 a33 a34 a41 a42 a43 a44

a11 a12 a21 a22 a13 a14 a23 a24 a31 a32 a41 a42 a33 a34 a43 a44

8-by-8 matrix

a_{11} a_{12} a_{13} a_{14} a_{15} a_{16} a_{17} a_{18}

a_{21} a_{22} a_{23} a_{24} a_{25} a_{26} a_{27} a_{28}

a_{81} a_{82} a_{83} a_{84} a_{85} a_{86} a_{87} a_{88}

row-major

a_{11} a_{12} a_{13} a_{14} a_{15} a_{16} a_{17} a_{18}

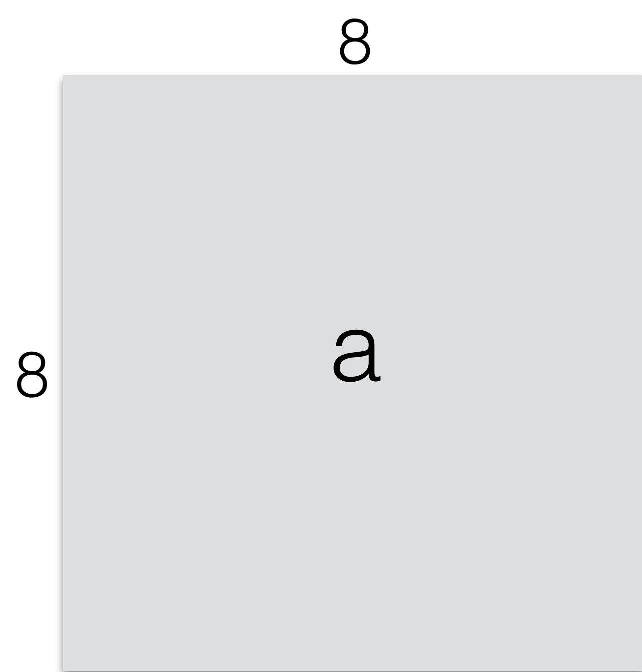
a_{21} a_{22} a_{23} a_{24} a_{25} a_{26} a_{27} a_{28}

a_{81} a_{82} a_{83} a_{84} a_{85} a_{86} a_{87} a_{88}

z-order

the numbers represent the order in which we store the elements

Z-order and cache misses



Z-order

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

a: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 | 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

a_{11} a_{12} a_{21} a_{22}

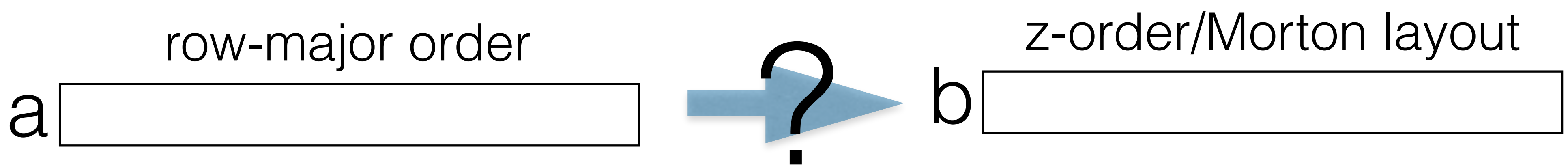
Any canonical sub-matrix will be contiguous in this layout and reading it will cause $O(r^2/B)$ cache misses (where the sub-matrix has size r-by-r).

D&C matrix multiplication with Z-order layout would (also) become easier and faster

```
c = (double*) calloc(sizeof(double), n*n)

void mmult(double* a, double* b, double*c, int n) {
      //base case
      if (n==1)
              c += a*b
              return
     else
          mmult(a, b, c, n/2)
          mmult(a+n^2/4, b+n^2/2,  c,  n/2)
          …
          …
          …
          …
          …
          …
   }
```

a11 starts at a
a12 starts at a + n*n/4
a21 starts at a + n*n/2
a22 starts at a + 3n*n/4
….

Ok, so having a matrix laid out in this recursive order would be handy and cache-efficient for matrix multiplication



How would we obtain this layout?

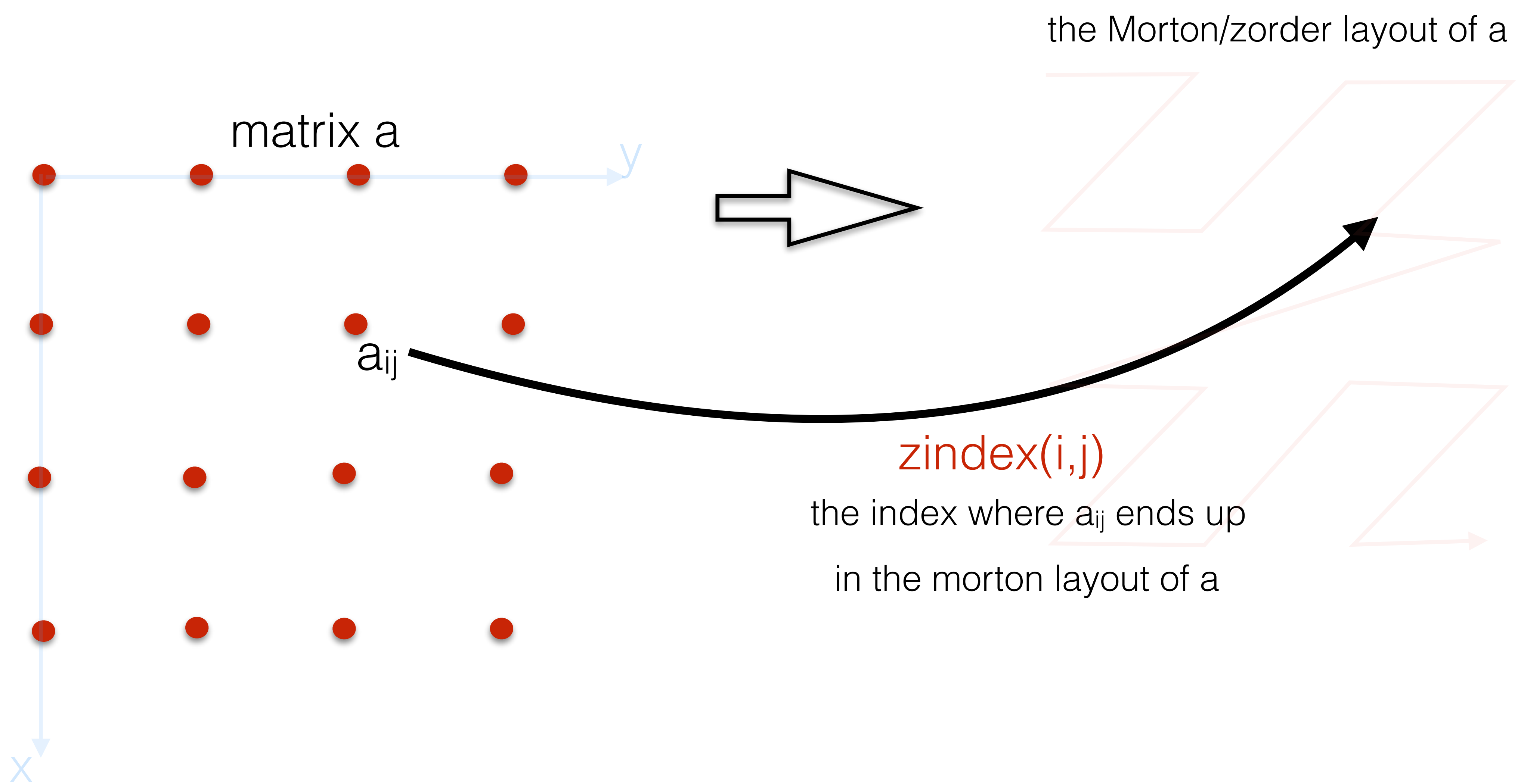
```
//b will store the Morton layout of a
b = calloc(n*n*sizeof(double))


/* a is a matrix of size n-by-n in row-major order
   for simplicity assume n=2^k so that the matrix stays square
   through recursion
   this function will fill in b, which stores a in Morton order
*/
morton(double* a, double*b, int n) {




 }
```

Hint: think recursively

```
//b will store the Morton layout of a
b = calloc(n*n*sizeof(double))


/* a is a matrix of size n-by-n in row-major order
   for simplicity assume n=2^k so that the matrix stays square
   through recursion
   this function will fill in b, which stores a in Morton order
*/
morton(double* a, double*b, int n) {
   if (n==1)
       b[0] = a[0]
   else
       morton(a11, b, n/2)
       morton(a12, b+n*n/4, n/2)
       morton(a21, b+n*n/2, n/2)
       morton(a22, b+3n*n/4, n/2)
  }
```

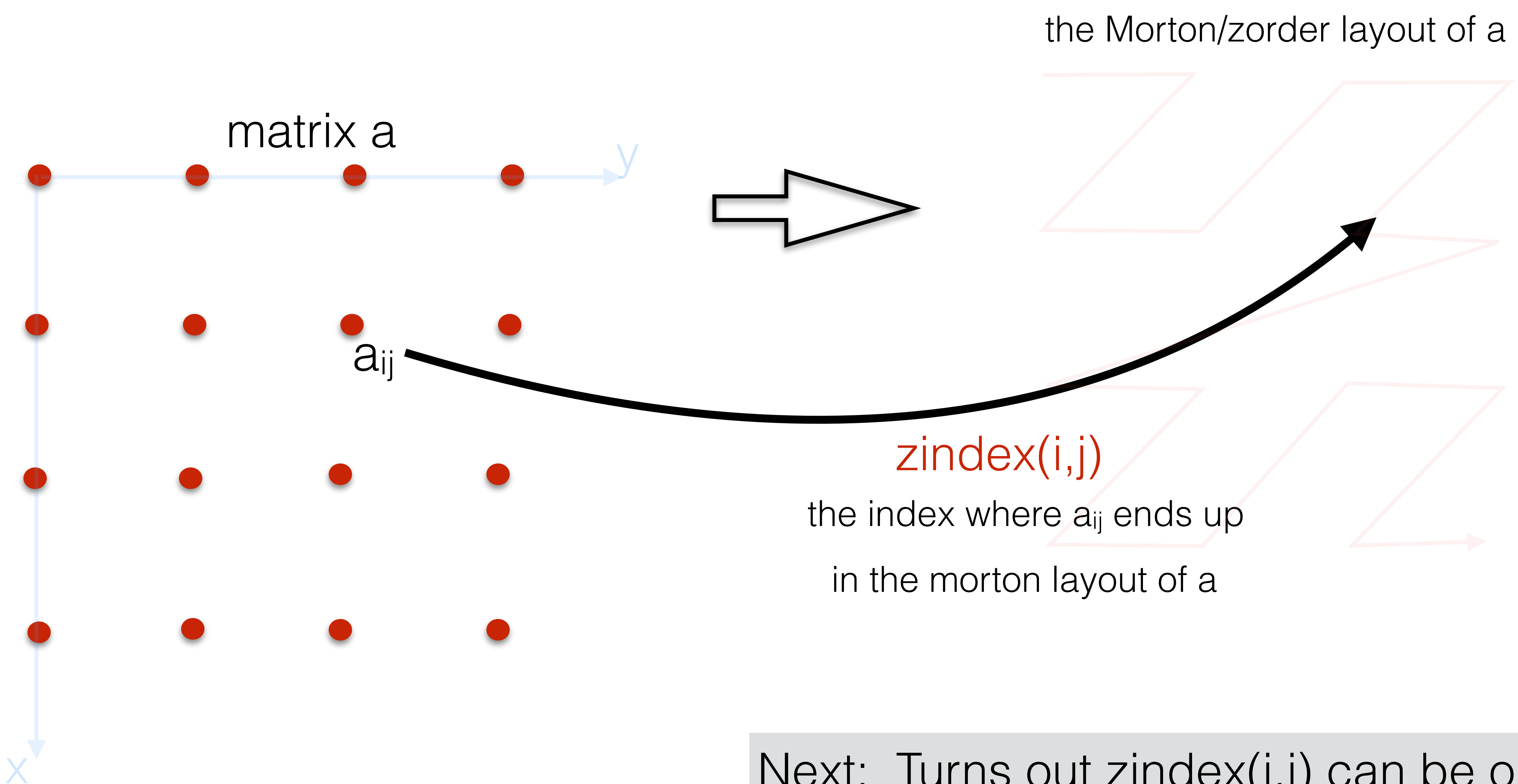
A different way to think about layouts is to view them as functions

Matrix layout as a function




```
for i=0 to n
  for j=0 to n
    copy a[i*n + j] to b[zindex(i,j)]
```

Matrix layout as a function



Next: Turns out zindex(i,j) can be obtained by interleaving the bits of i and j !

```
for i=0 to n
  for j=0 to n
     copy a[i*n + j] to b[zindex(i,j)]
```

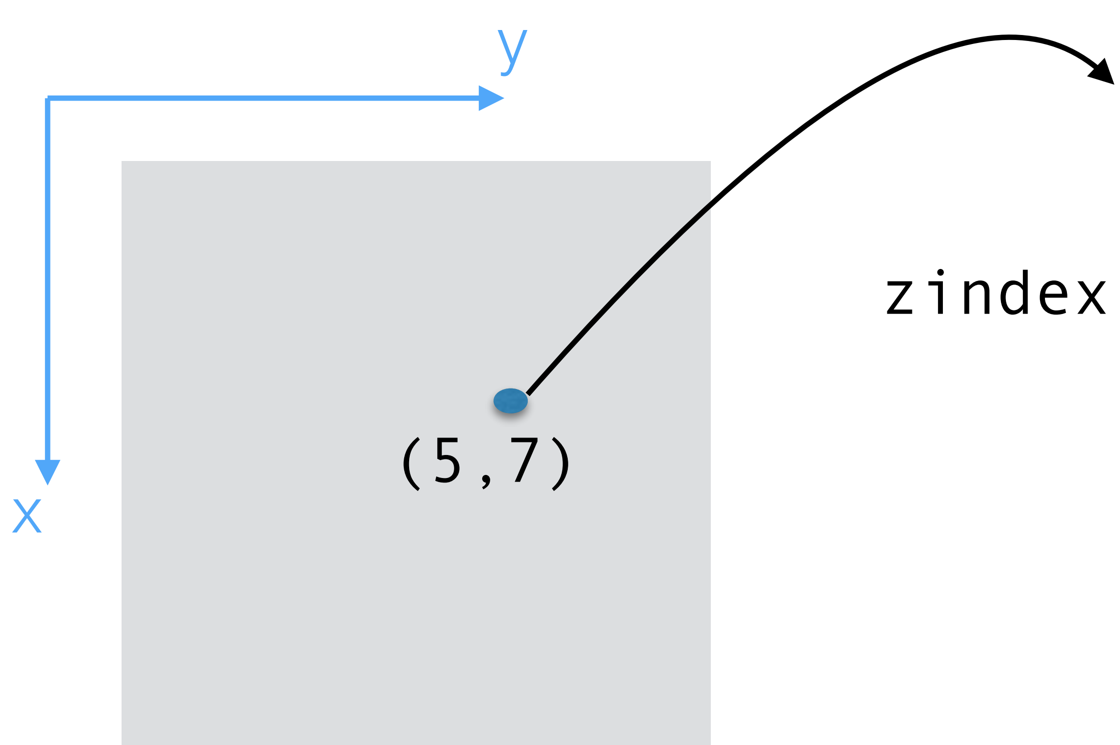
Z-indices via bit manipulation

- Consider element a_{xy} at row x and column y in matrix a
- Assume x, y are integers on k bits

$$x = x_1 x_2 x_3 \ldots x_k, \quad y = y_1 y_2 y_3 \ldots y_k$$

- Define zindex(p) as the interleaving of bits from x and y

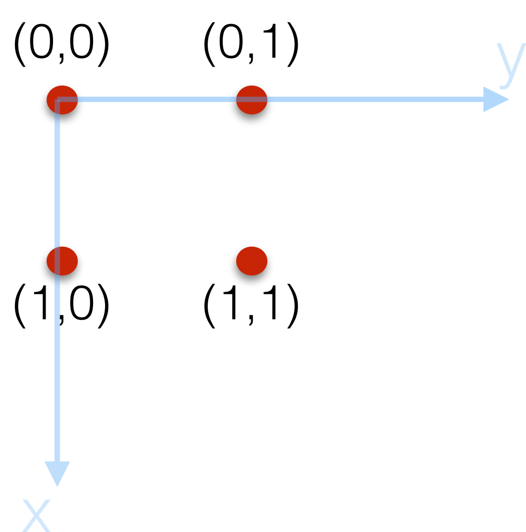
$$\mathrm{zindex}(p) = x_1 y_1 x_2 y_2 \ldots x_k y_k$$



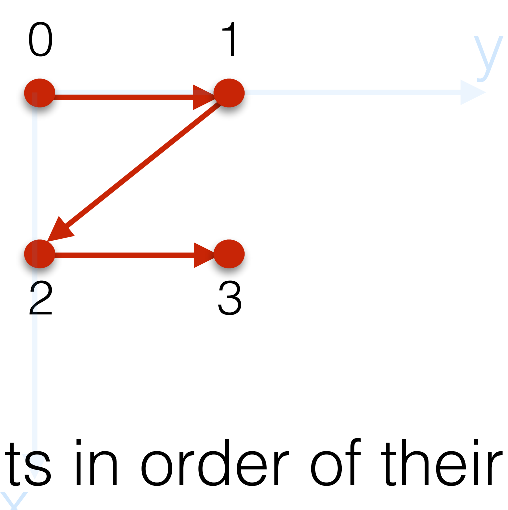

```
x=5  =  0101
y=7  =  0111

zindex(5,7)= 00110111 = 54
```

point (5,7) maps to index 54 in zorder

points with coordinates on k=1 bit

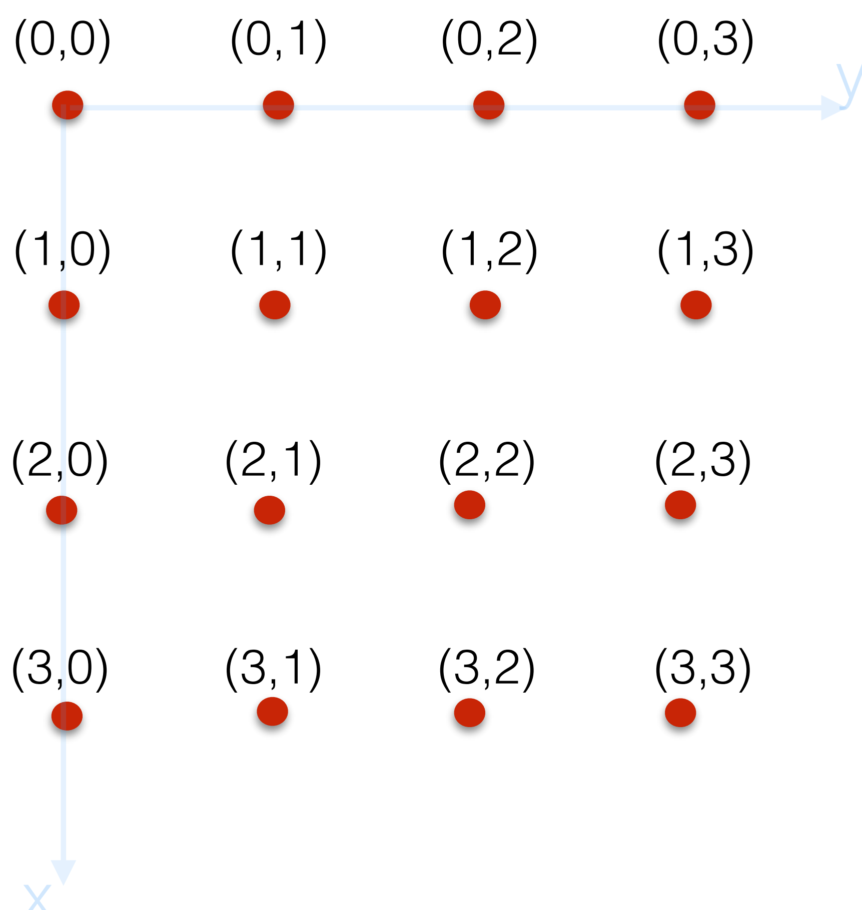


p	Zindex(p)
(0,0)	00=0
(0,1)	01=1
(1,0)	10=2
(1,1)	11=3



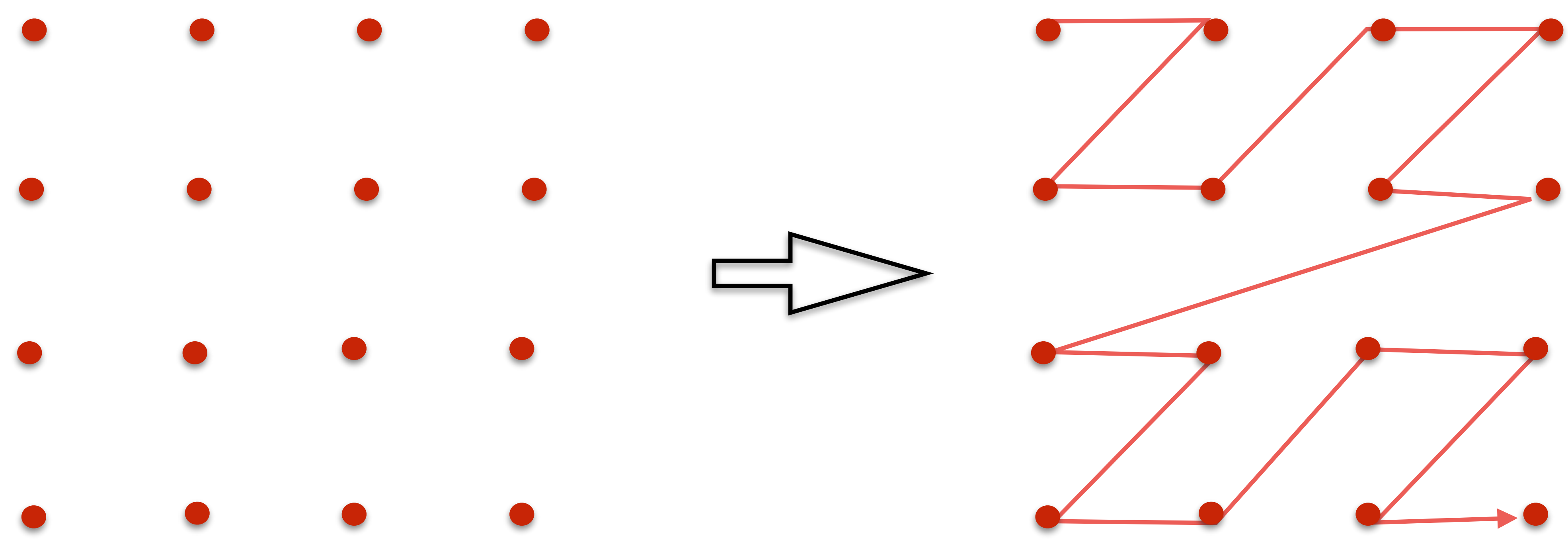
Z-order: points in order of their zindices

points with coordinates on k=2 bits



p	Z_index(p)
(00,00)	0000=0
(00,01)	0001=1
(00,10)	
(00,11)	
(01,00)	
(01,01)	
(01,10)	
(01,11)	
(10,00)	
(10,01)	
(10,10)	
(10,11)	
(11,00)	
(11,01)	
(11,10)	
(11,11)	

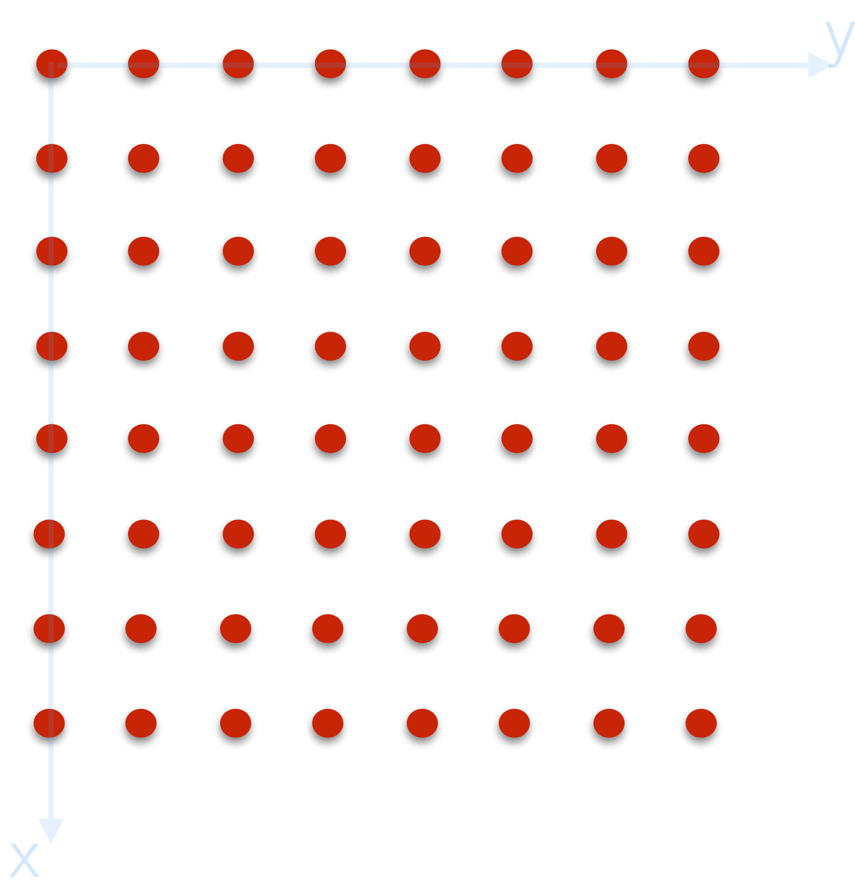
points with coordinates on k=2 bits



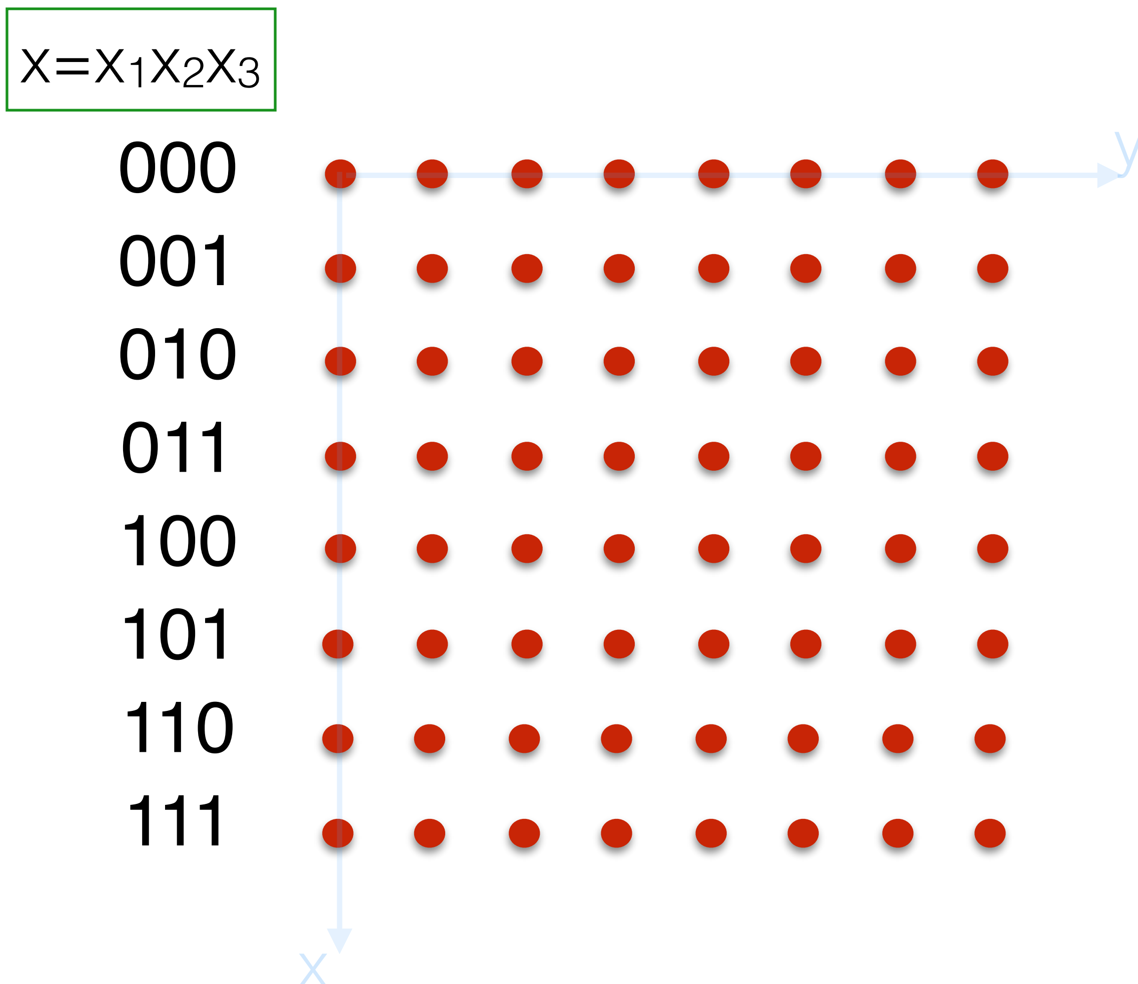
set of points

Z-order: points in order of their zindices

points with coordinates on k=3 bits



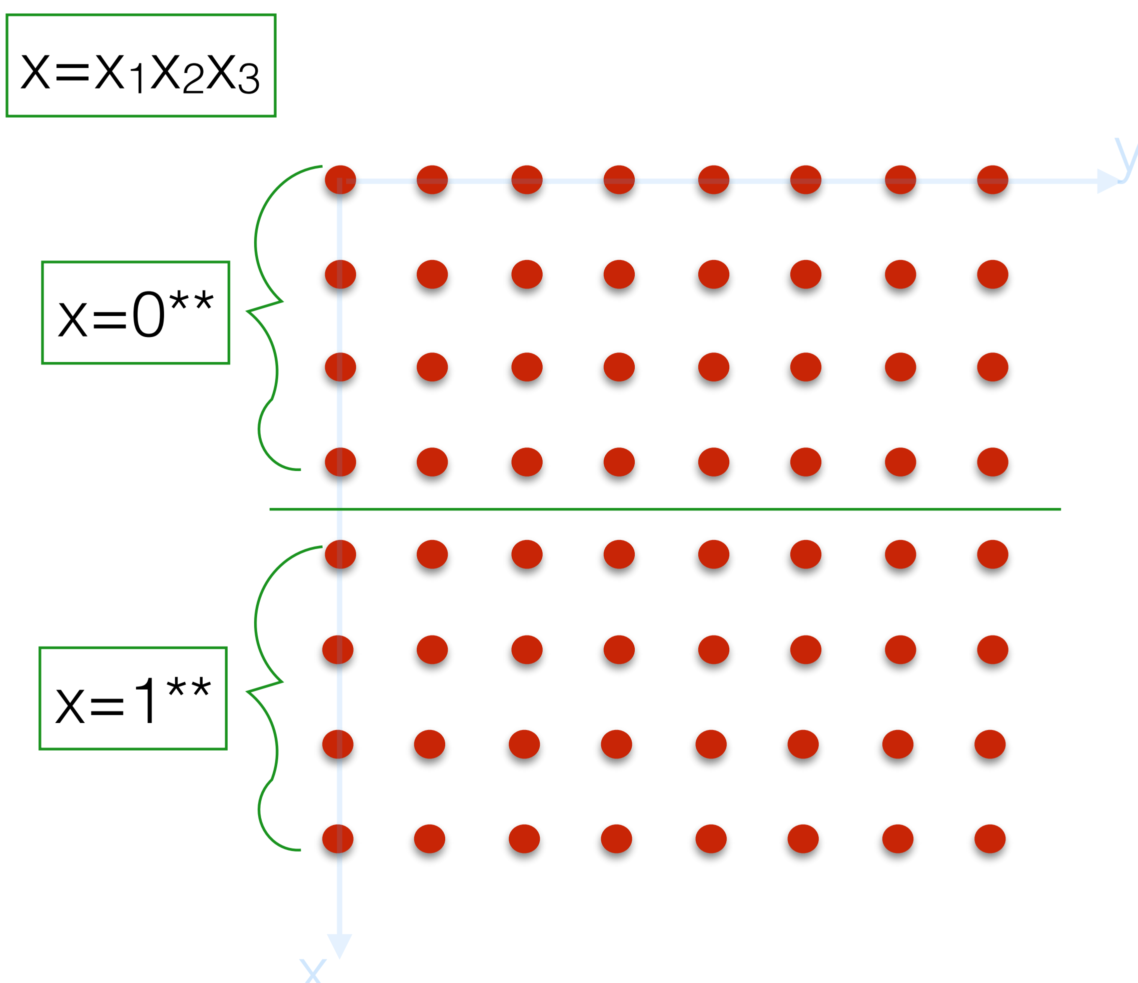
points with coordinates on k=3 bits



Consider a row $x=x_1x_2x_3$ in $[0,\ldots 8)$

- $x_1=0$ means the point will reside in first half
- $x_1=1$ means the point will reside in second half

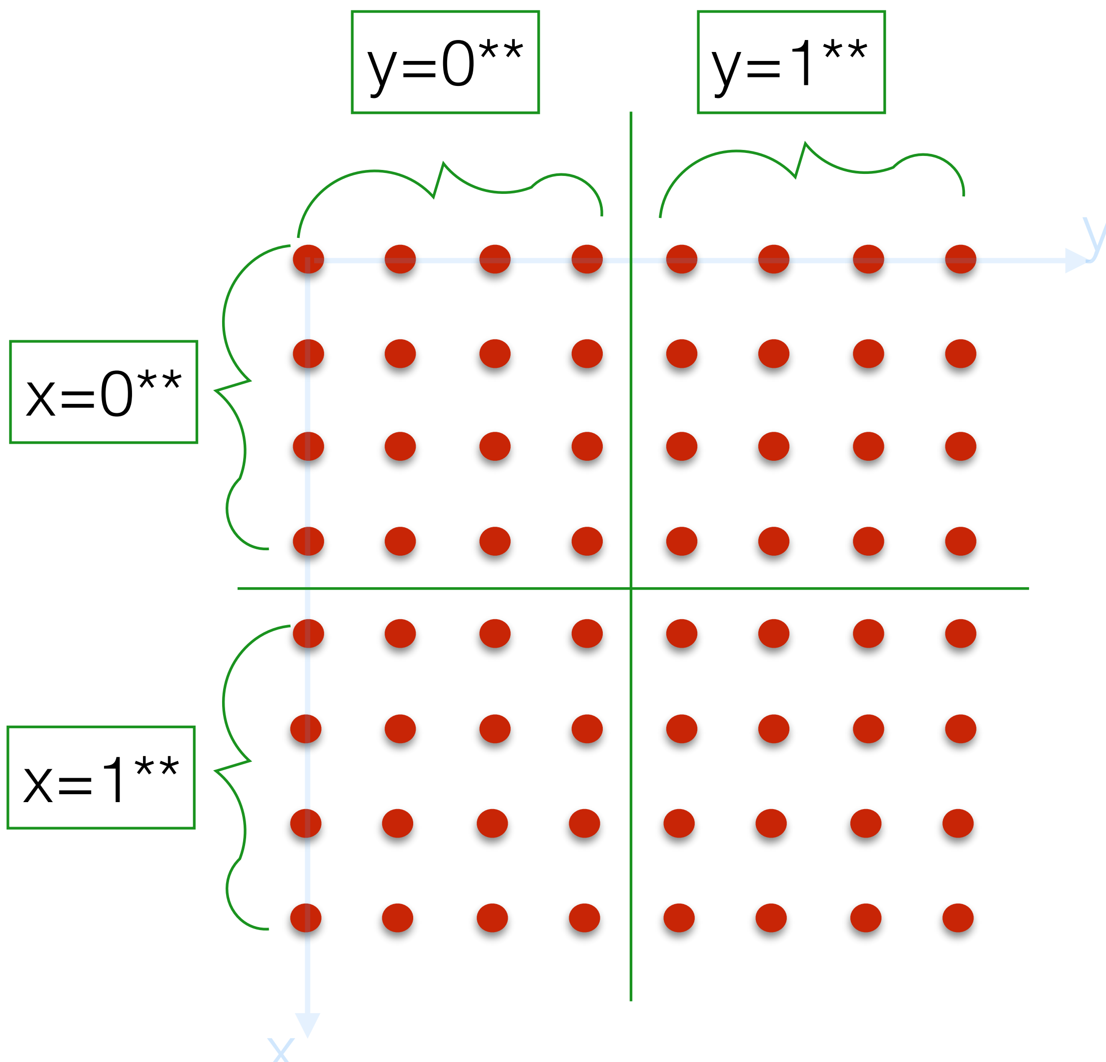
k=3 bits



Consider a row $x=x_1x_2x_3$ in [0,…8)

- $x_1=0$ means the point will reside in first half
- $x_1=1$ means the point will reside in second half

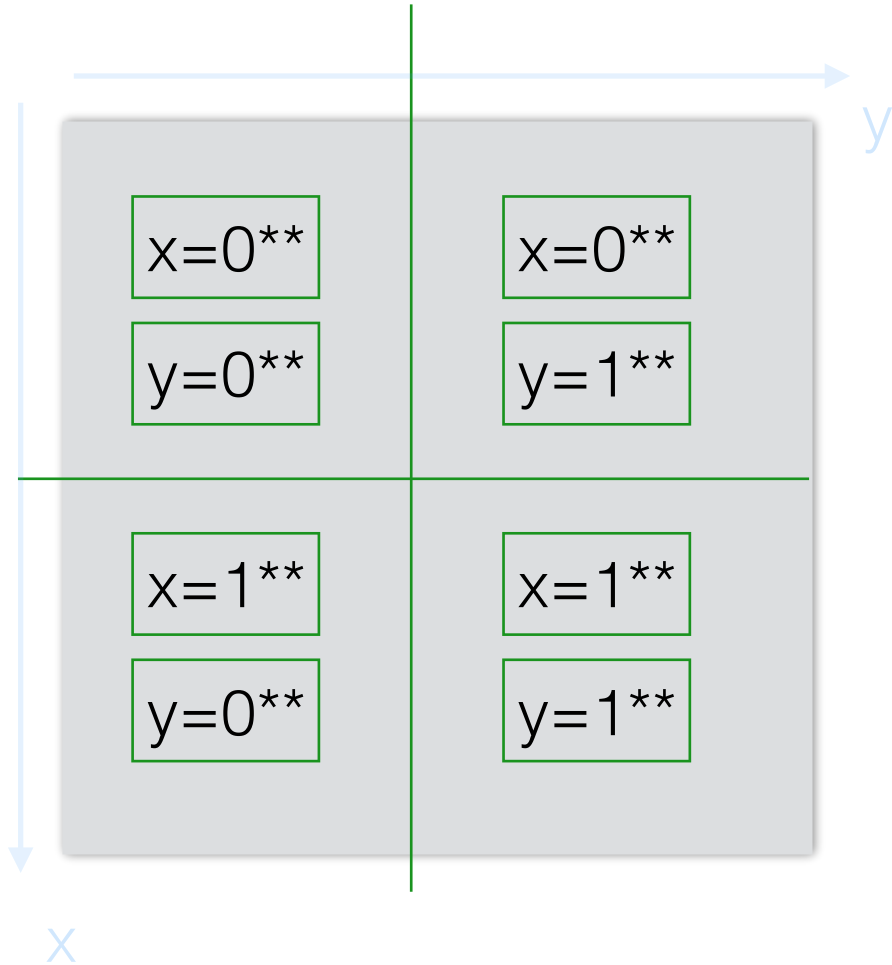
k=3 bits



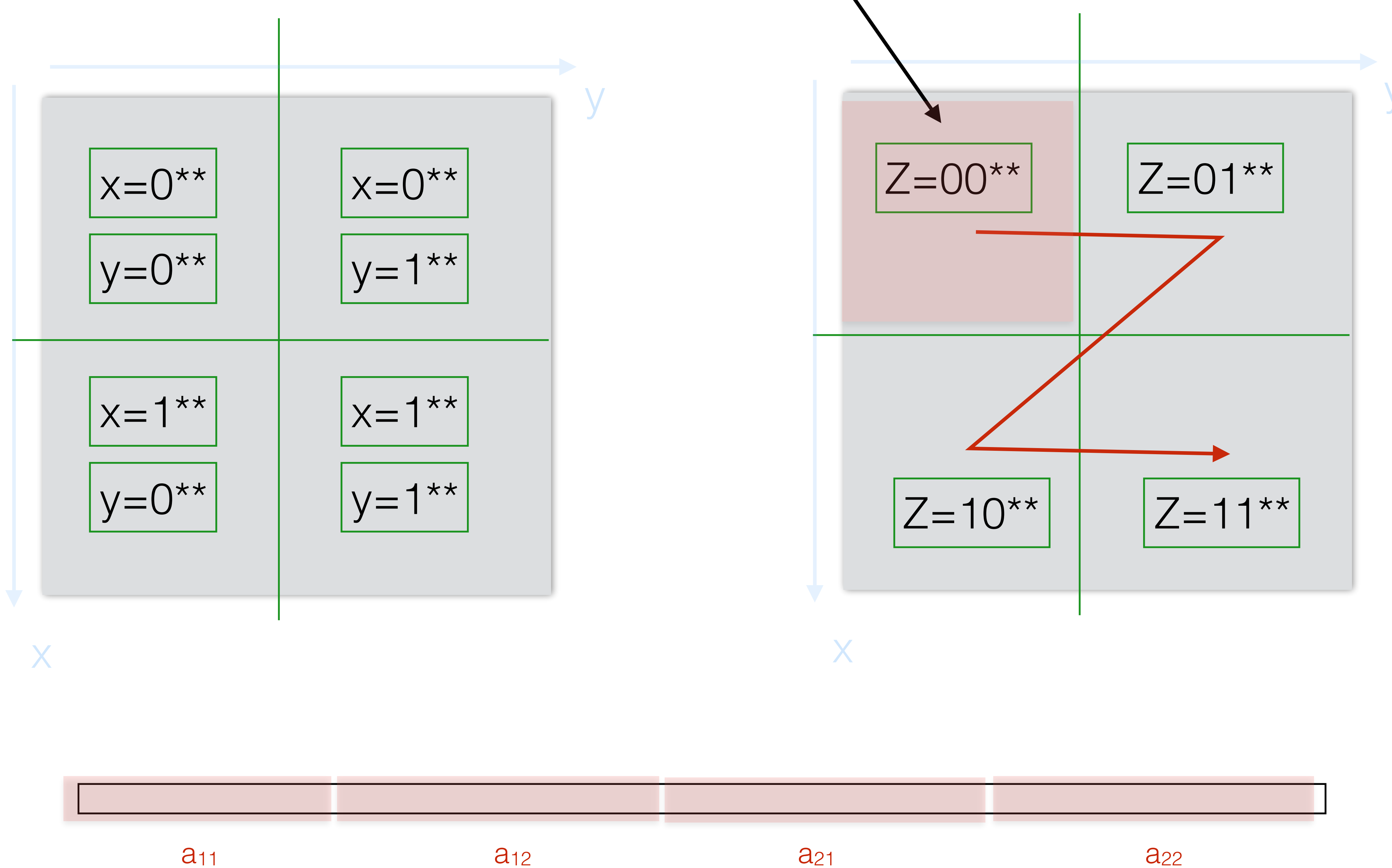
Consider a column $y_1y_2y_3$ in [0,…8)

- y_1=0 means the point will reside in first half
- y_1=1 means the point will reside in second half

k=3 bits

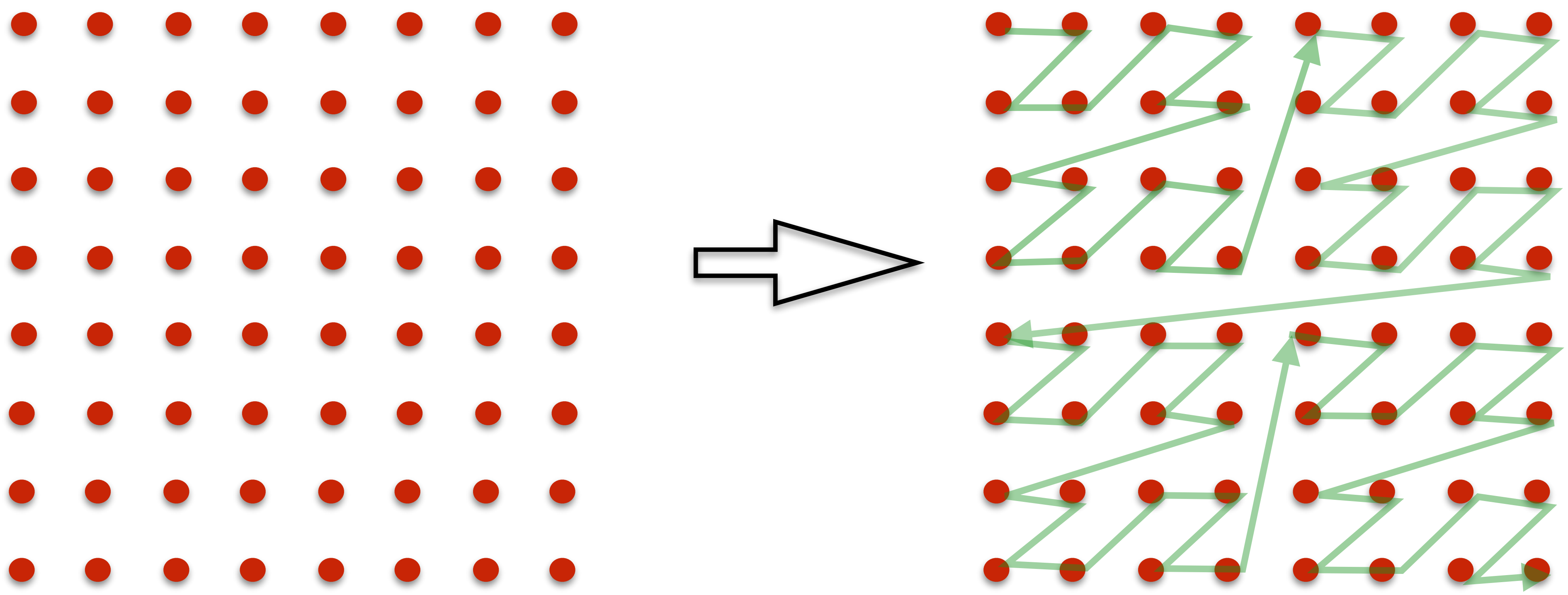


zindices of all points in a11 start with 00



Same order as before!

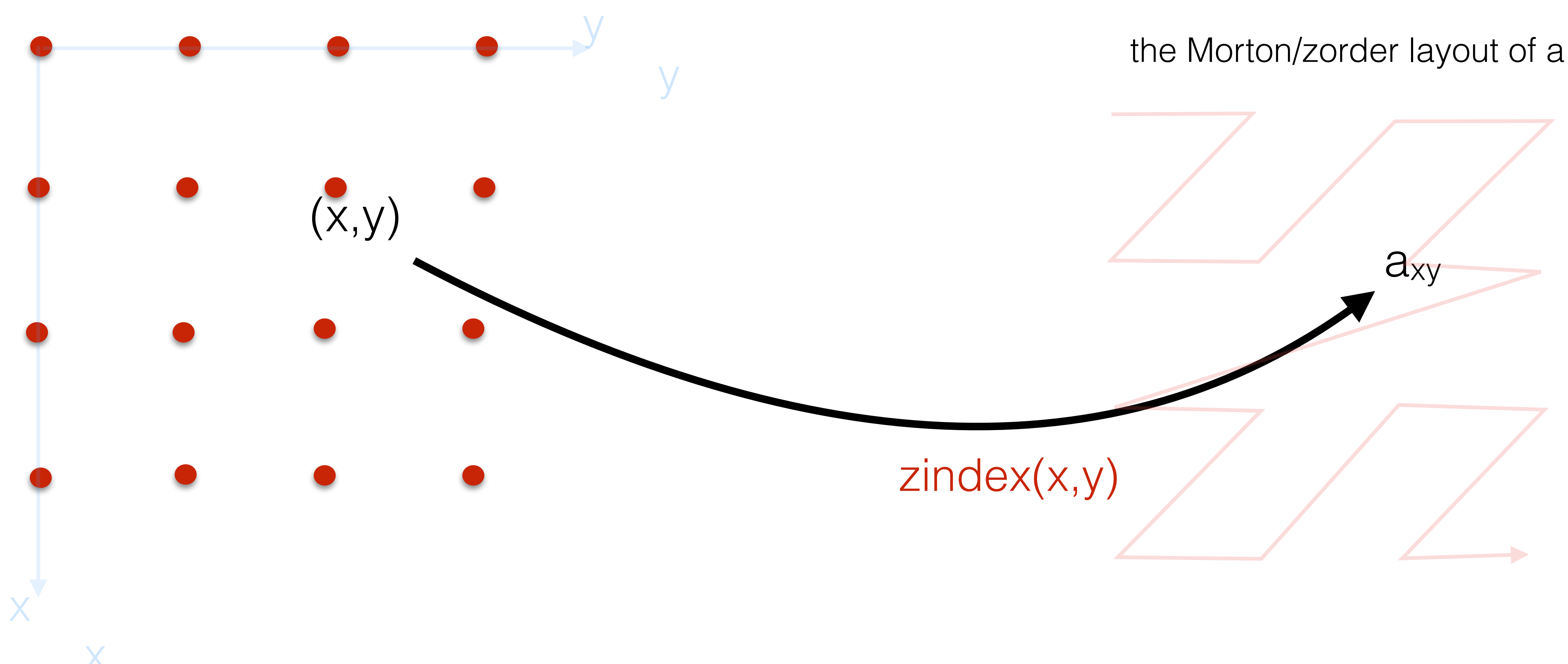
Z-order for k=3 bits



set of points

Z-order: points in order of their zindices

Zindex as a function from 2D to 1D



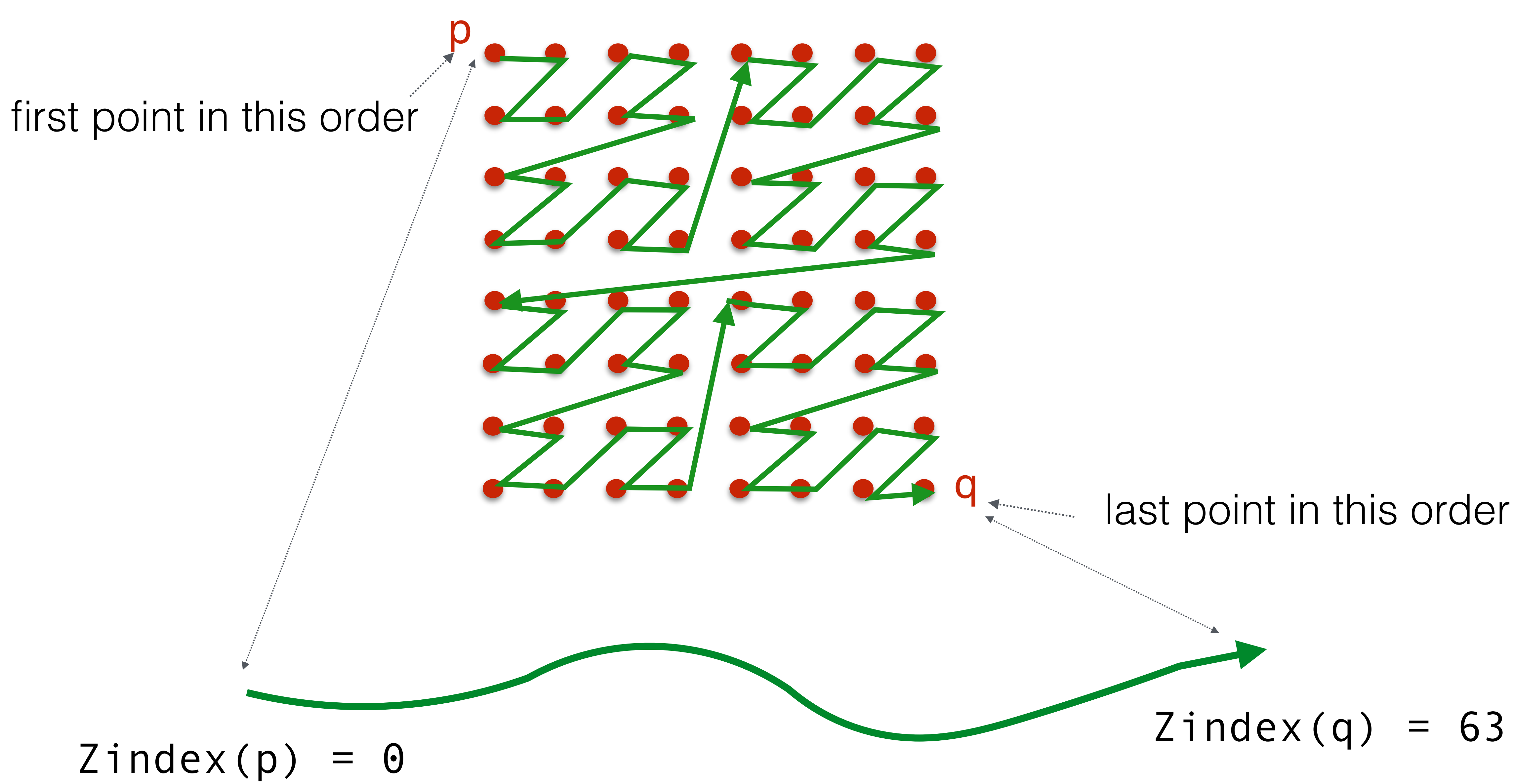
Zindex: $[0,2^k) \times [0,2^k) \dashrightarrow [0,2^{2k})$

We are mapping a 2d coordinate to a 1d coordinate

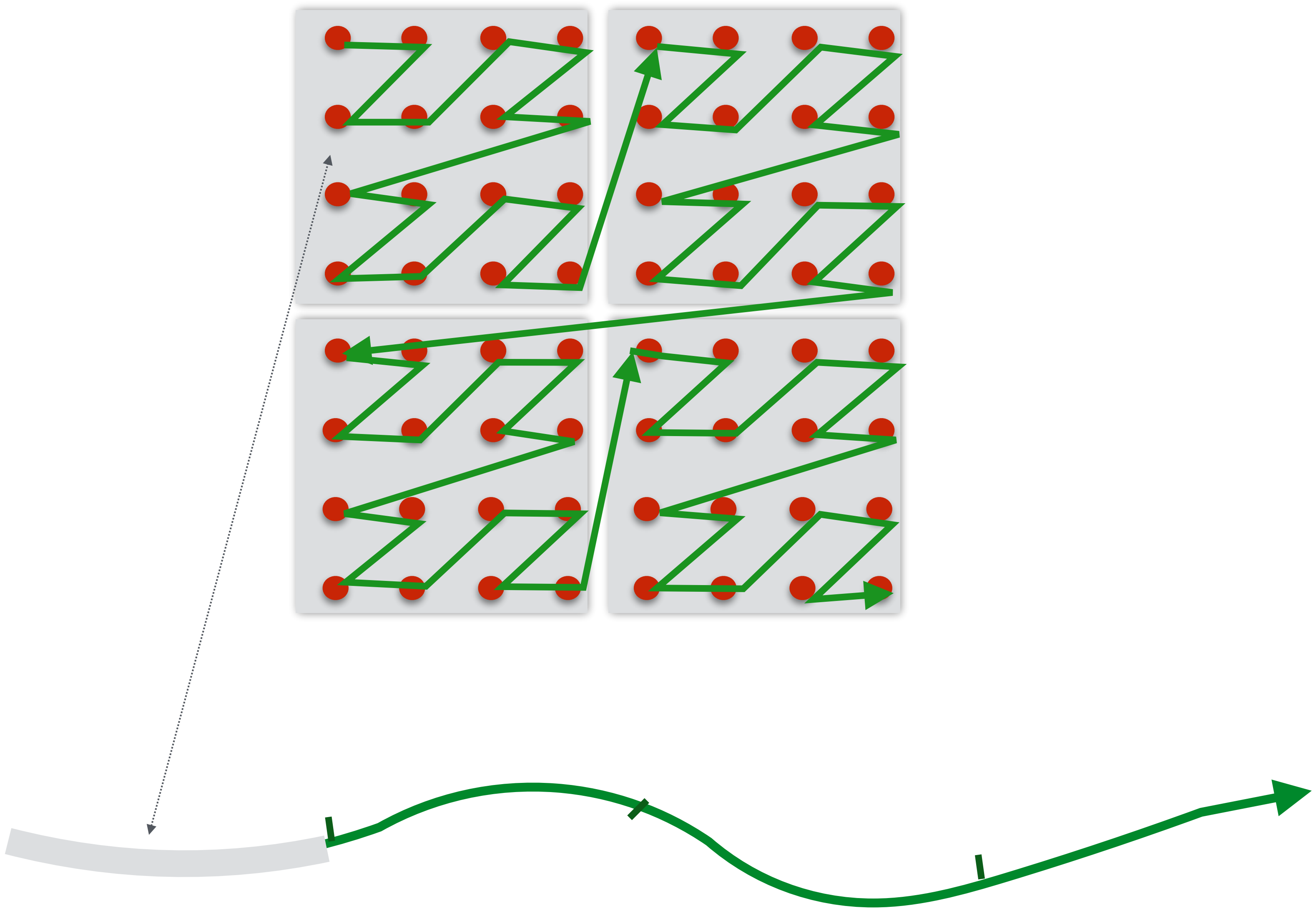
We are "serializing" a 2d space (like putting pearls on a thread)

Properties of z-indices

```
For k=3,  Zindex: [0,8) x [0,8) --> [0,64)
```



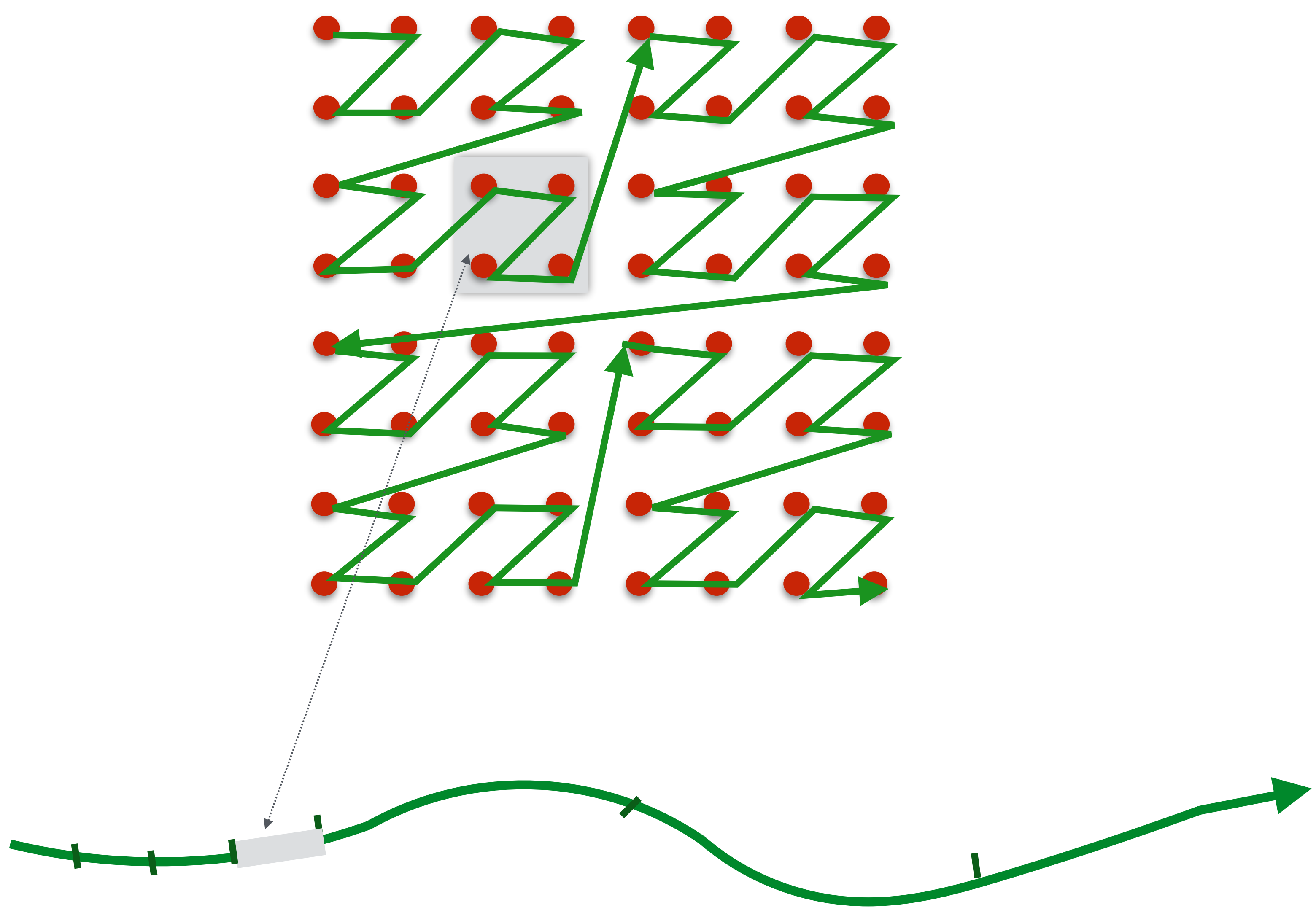
Properties of z-indices



all the points in this
quadrant have first
two bits 00

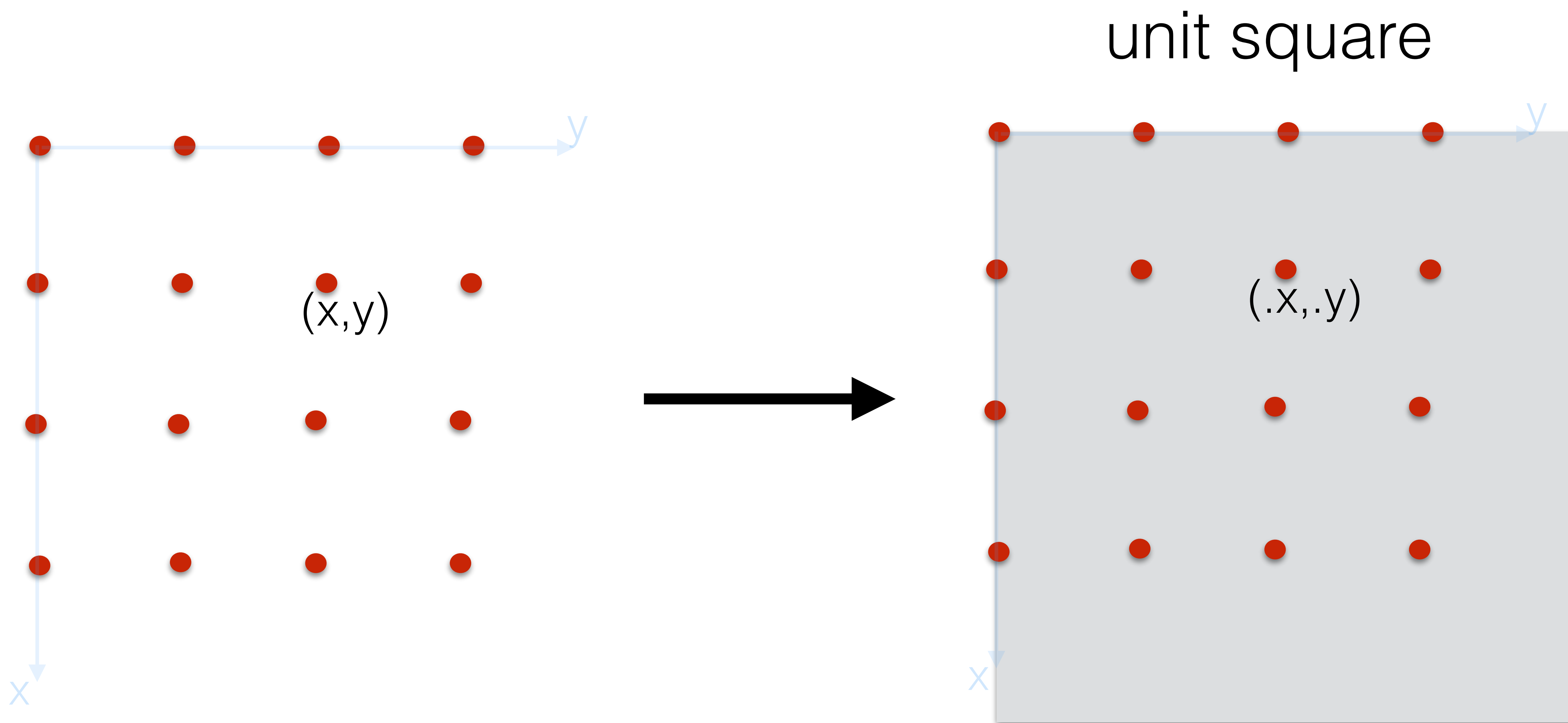
Properties of z-indices

For a matrix stored in this order, any canonical block matrix maps to an interval of the z-curve and will be stored contiguously.



```
all the points in this
 quadrant have first
   four bits 0011
```

From integers to real numbers



Assume integers on k bits:

$x = x_1x_2x_3\ldots x_k$, $y = y_1y_2y_3\ldots y_k$

x, y in $[0,2^k)$

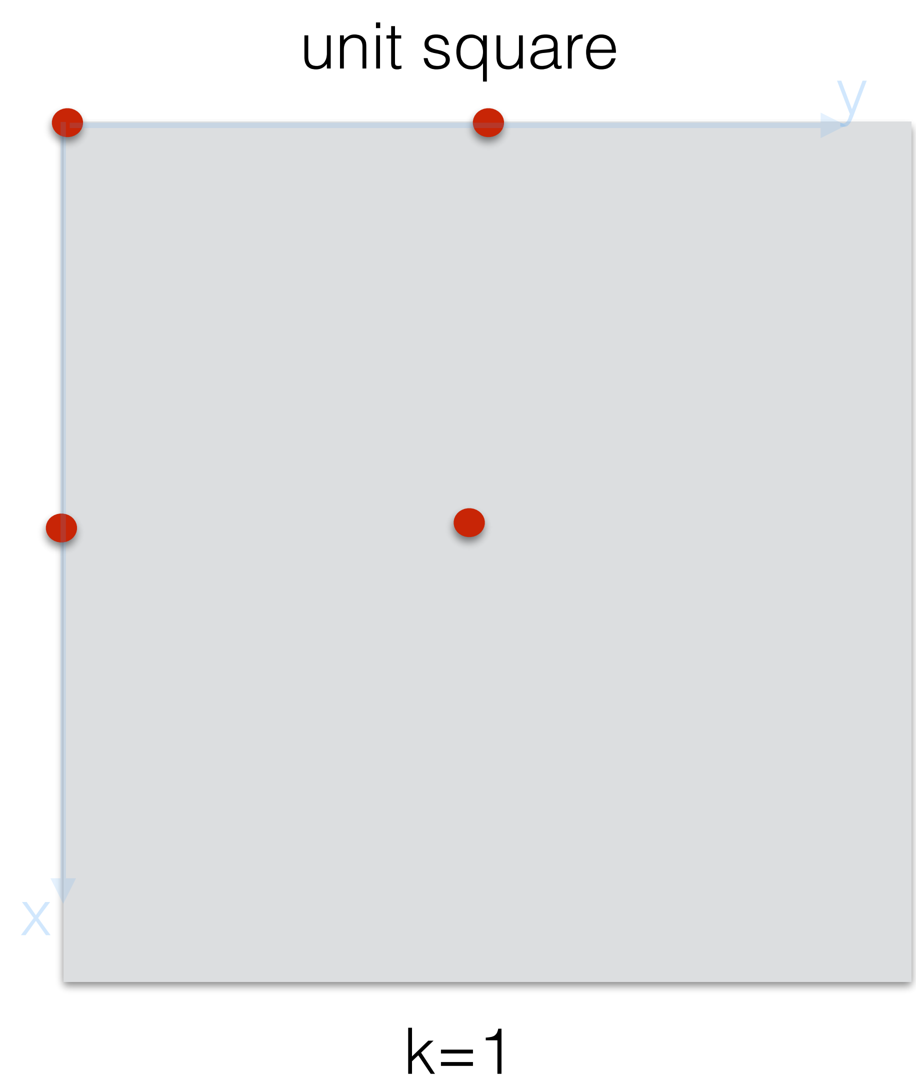
divide by 2^k

$x/2^k = .x_1x_2x_3\ldots x_k$

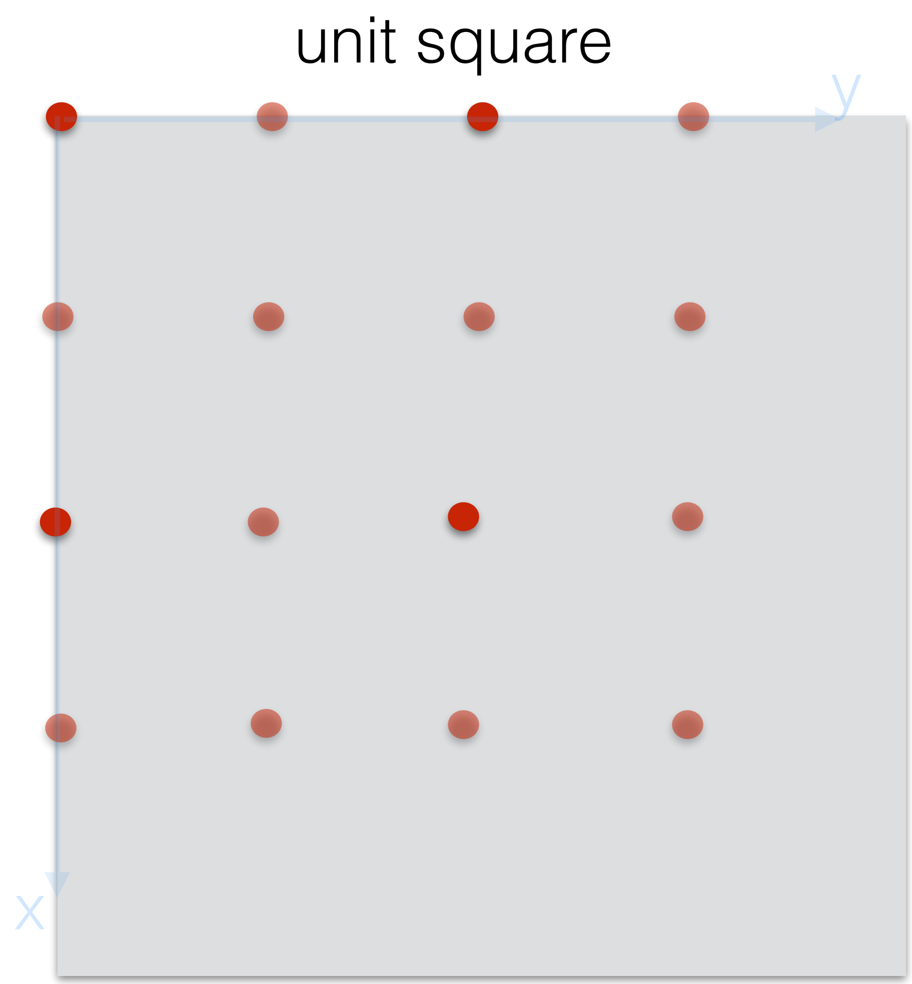
$y/2^k = .y_1y_2y_3\ldots y_k$

in $[0,1)$ with k bits of precision

From integers to real numbers



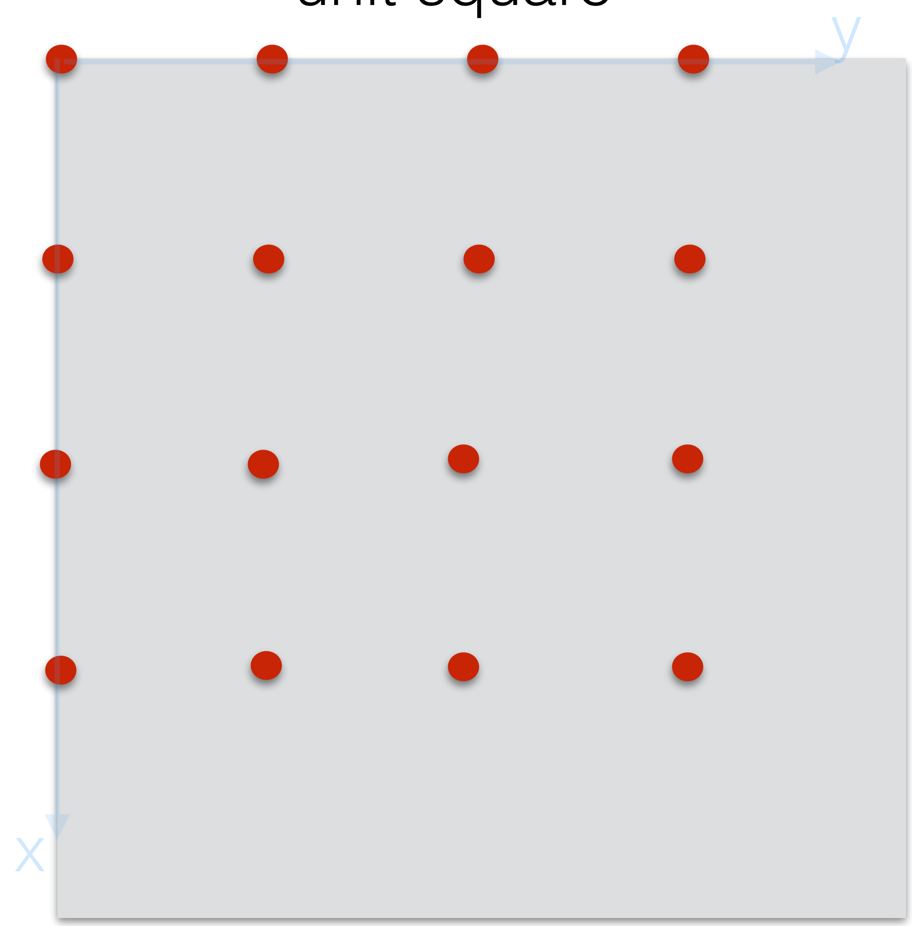
From integers to real numbers



k=2

From integers to real numbers

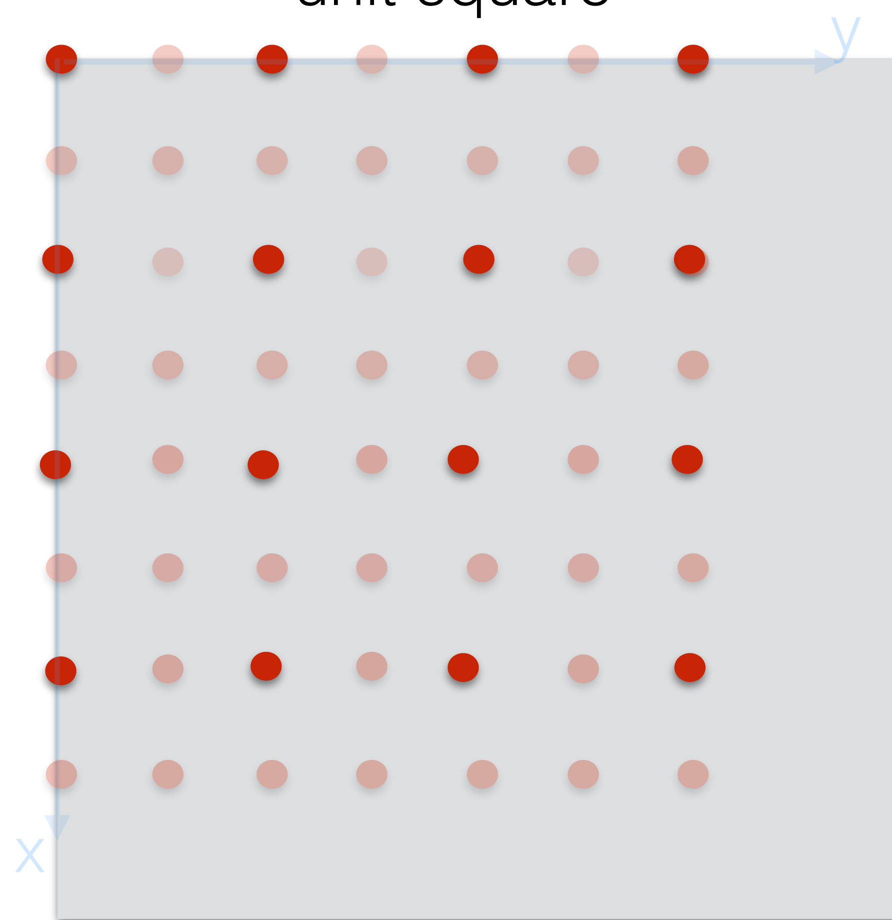
unit square



k=2

From integers to real numbers

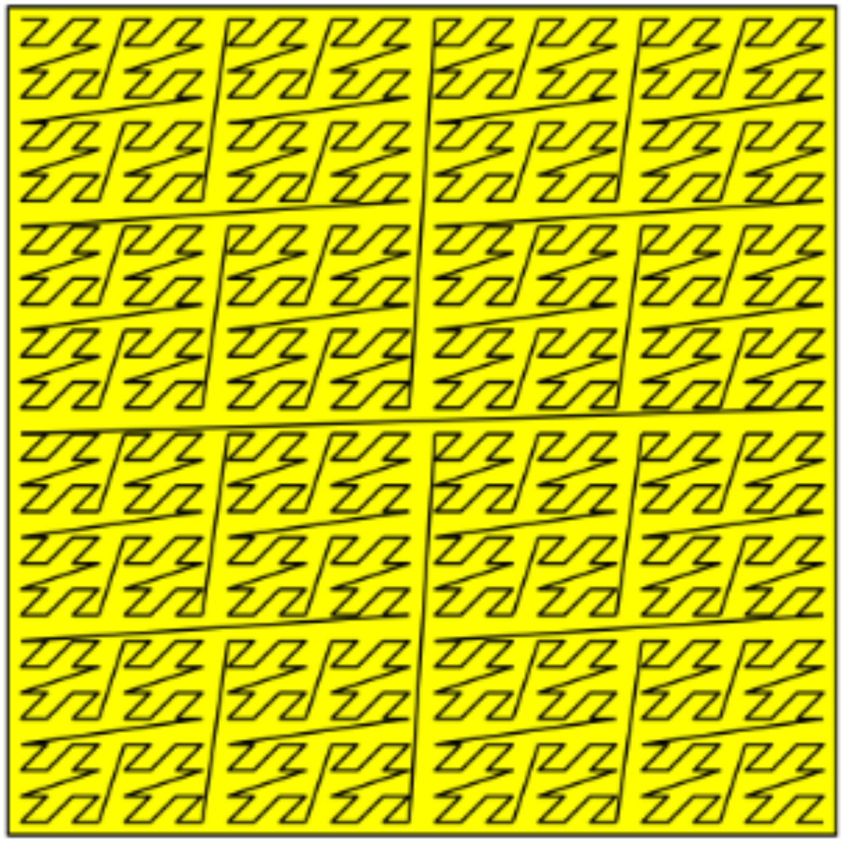
unit square



k=3

As k -> infinity, at the limit of this recursive process, the points completely fill the unit square

Space filling curves



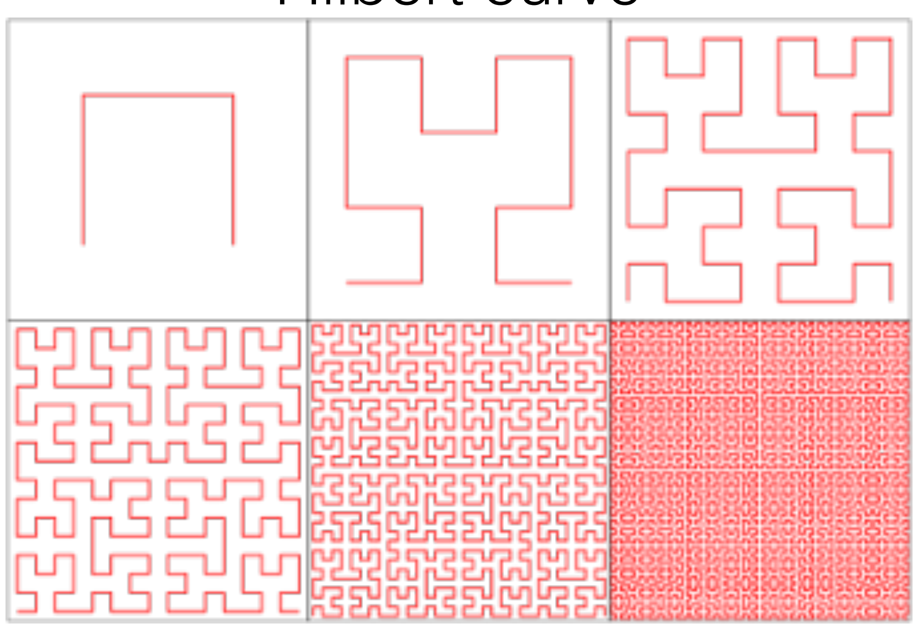
`Zindex: [0,1) x [0,1) --> [0,1)`

As k -> infinity, the z-order of the points completely fills the unit square

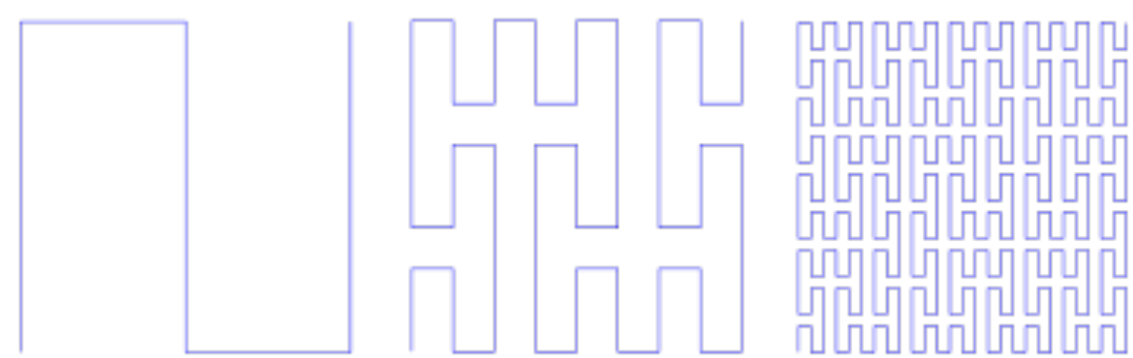
Space filling curves

- A space filling curve is a curve that covers an area (or volume)
- Mathematically, a mapping Z : [0,1) x [0,1) —-> [0,1) that’s continuous and surjective
- Presented late 1800’s by Peano, Hilbert, Sierpinski, etc
- Pretty incredible (“topological monsters”)
- Construction of SFC: start with a generator that establishes an order of traversal of the initial domain (unit square), then recurse; place same or rotated/reflected,.., etc versions of the generator at the next-level

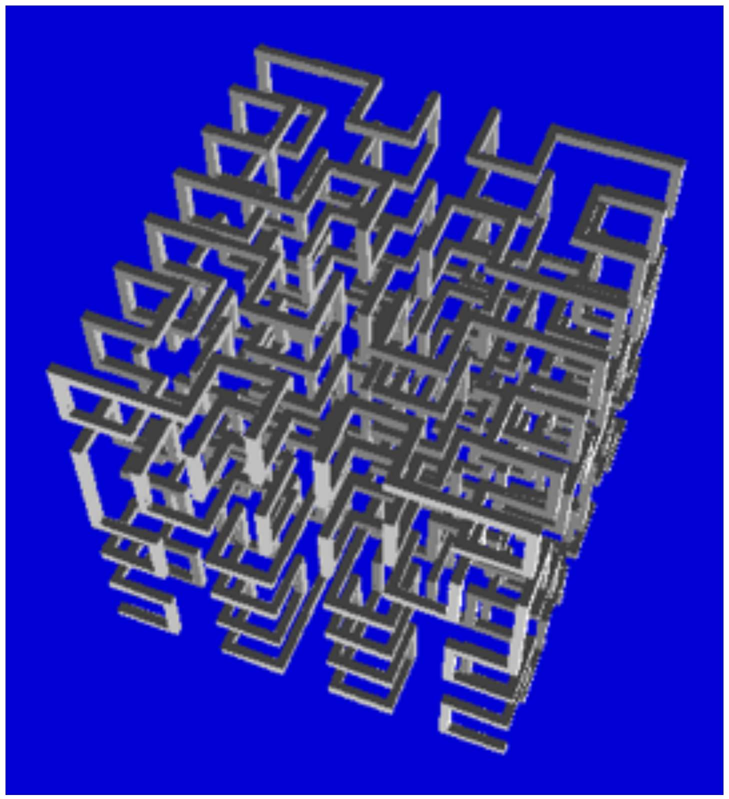
Hilbert curve

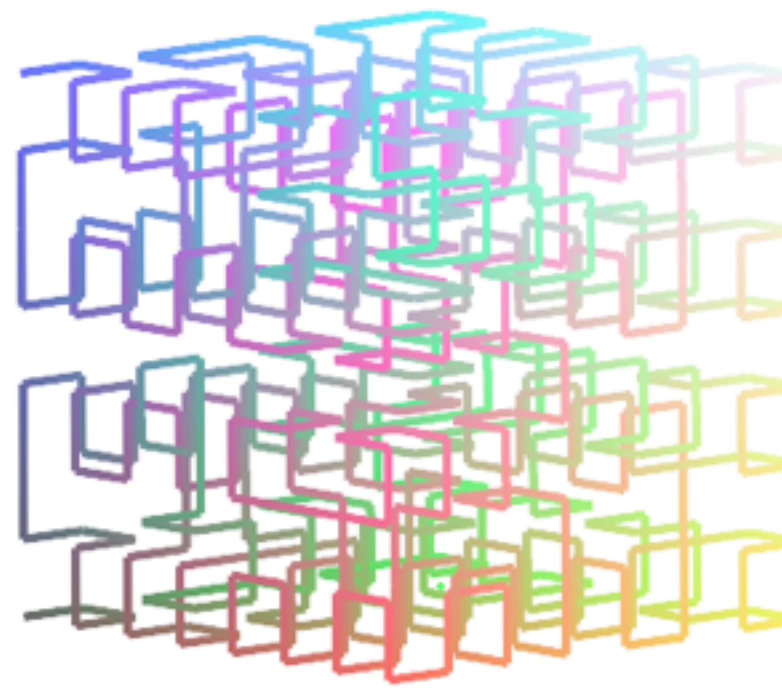


Peano curve



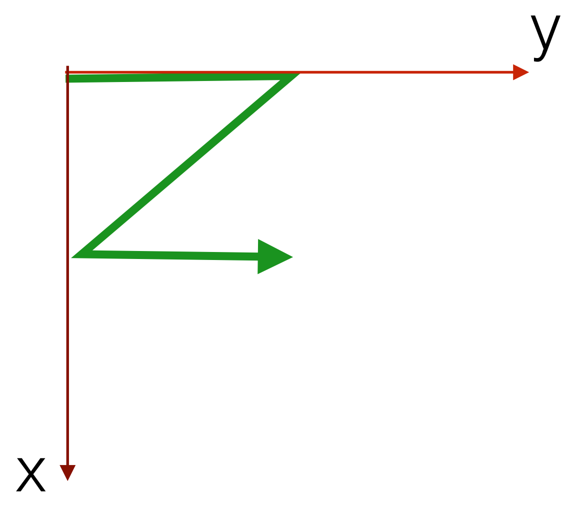
Hilbert curve in 3D

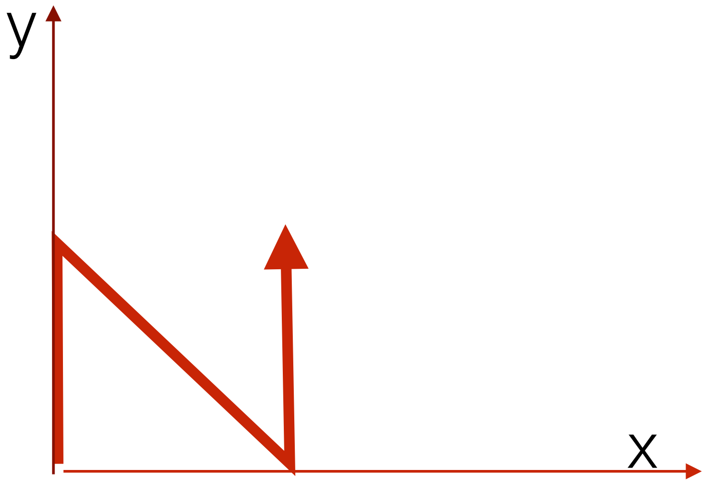




Symmetrical Z-orders

- Other Z-orders can be obtained similarly



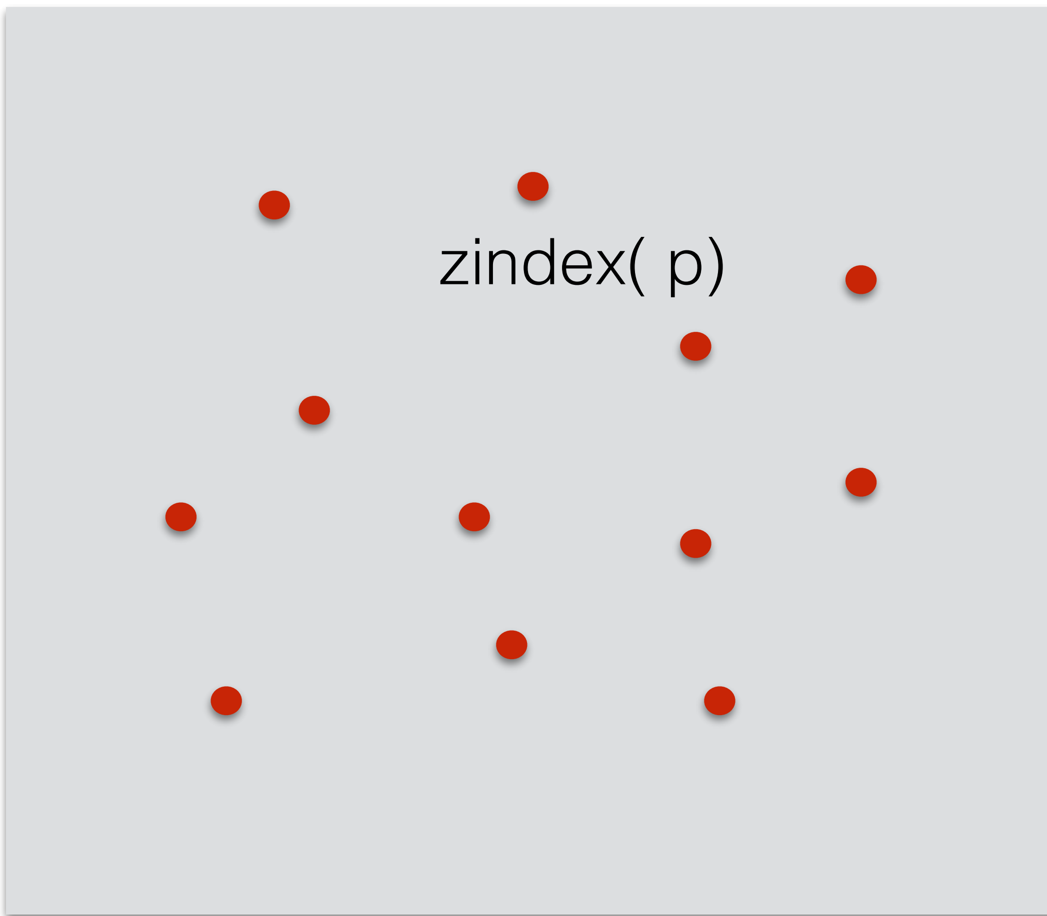


Space filling curves in CS

- Used to sequentialize high-dimensional data
  - e.g.: pixels in an image, points in a scene, voxels in a geometric model, particles in a simulation, entries in a database, …
  - The data appears sequential along the SFC (like pearls on a thread)
- SFC have good spatial locality
  - points that are close in space, are close on the SFC
- Used to improve spatial locality
  - points in the same canonical block are stored contiguously
- Used in parallel computing for load distribution and load balancing
  - order elements by the SFC and divide them into equal chunks

Example

A set of 2D points with integer coordinates



For all p: Compute zindex(p)

Sort points by their zindex and store them in this order

Example

Recursive Array Layouts and Fast Parallel Matrix Multiplication[*]

Siddhartha Chatterjee[†] Alvin R. Lebeck[‡] Praveen K. Patnala[†] Mithuna Thottethodi[‡]

Abstract

Matrix multiplication is an important kernel in linear algebra algorithms, and the performance of both serial and parallel implementations is highly dependent on the memory system behavior. Unfortunately, due to false sharing and cache conflicts, traditional column-major or row-major array layouts incur high variability in memory system performance as matrix size varies. This paper investigates the use of recursive array layouts for improving the performance of parallel recursive matrix multiplication algorithms.

We extend previous work by Frens and Wise on recursive matrix multiplication to examine several recursive array layouts and three recursive algorithms: standard matrix multiplication, and the more complex algorithms of Strassen and Winograd. We show

nel in linear algebraic algorithms, and is enshrined in the `dgemm` routine in the BLAS 3 library [10]. There is an intimate relationship between the layout of the arrays in memory and the performance of the routine. On modern shared-memory multiprocessors with multi-level memory hierarchies, the column-major layout assumed in the BLAS 3 library can result in performance anomalies as the matrix size is varied. These anomalies result from unfavorable access patterns in the memory hierarchy that cause interference misses and false sharing and increase memory system overheads experienced by the code. In this paper, we investigate recursive array layouts accompanied by recursive control structures as a means of delivering high and robust performance for parallel dense linear algebra.

The use of quad- or oct-trees (or, in a dual interpretation, space-

https://engineering.purdue.edu/~mithuna/pubs/spaa99.pdf

Computing the zindex

```
//compute the zindex(x,y) and return it

int64 zindex(int32 x,int32 y)
```

??

Working with bits in C

- << (shift left)

  `e.g.:  1<<3 gives 8`

- >> (shift right)

  `e.g.: 15 >> 2 gives 3`

- & (bit AND)

  `e.g. 5 & 3 gives 1`

- | (bit OR)

  `e.g. 5 | 3 gives 7`

- `~ (bit complement : flips every bit)`

  `e.g. ~101 gives 010`

Working with bits

We'll first write some helper functions

```
//return the i-th bit from right to left
int getbit(int32 x, int i) {



  }
```

Working with bits

We'll first write some helper functions

```
//return the i-th bit from right to left
int getbit(int32 x, int i) {
     //this is (x & (1<<i) ) >> i
     mask = 1 << i
     thebit = (n & mask) >> i
     return thebit
  }

//could also write it as  (x >> i) & 1
```

Working with bits

```
//set the i-th bit from right to left to 1

int setbit(int32 x, int i) {




}
```

Computing the zindex

```
//compute the zindex(x,y) and return it

int64 zindex(int32 x,int32 y)
```