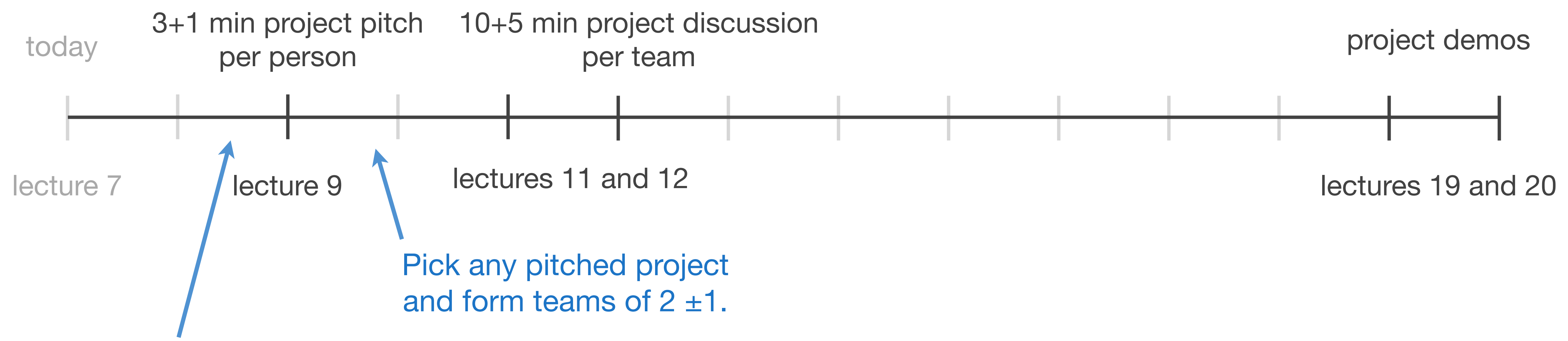


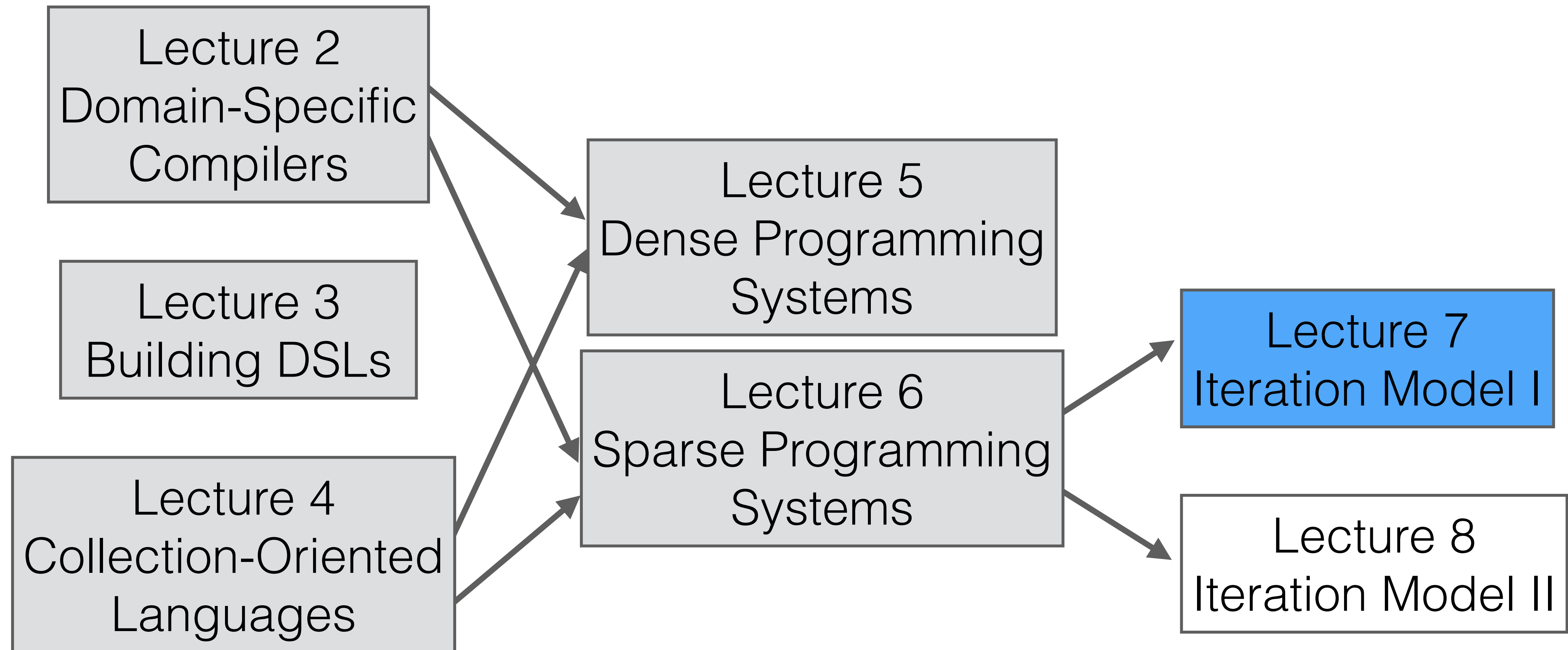
Lecture 7 — Sparse Iteration Model I

Stanford CS343D (Winter 2025)
Fred Kjolstad

Course Project



Each person contributes one pitch slide to a google slide deck.
These pitches are not binding.



Overview of topics

Lecture 7

- Data representation
- Iteration spaces
- Iteration graph IR
- Iteration lattices to represent coiteration

Lecture 8

- Concrete index notation IR
- Code generation algorithm
- Derived iteration spaces
- Optimizing transformations

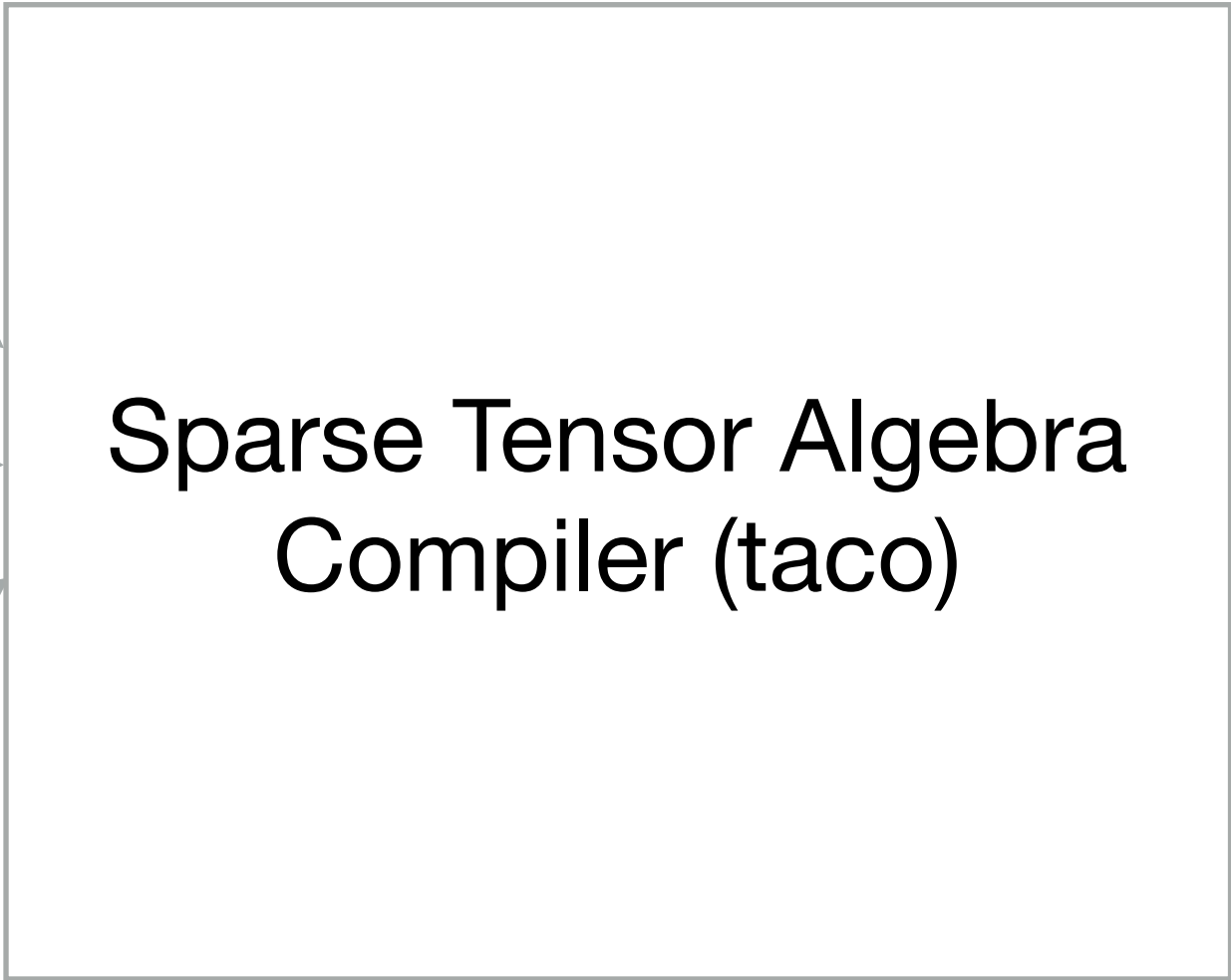
Sparse Tensor Algebra Compilation

Tensor Index Notation Expression

$$\begin{aligned} &A = Bc + a \qquad a = Bc \\ &A = B \odot C \qquad A = B + C \qquad a = \alpha Bc + \beta a \\ &A = BCD \qquad A = \alpha B \qquad A = 0 \qquad A = BC \\ &\qquad a = b \odot c \qquad A = B \odot (CD) \\ &A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \qquad A = B^T \qquad a = B^T Bc \\ &A_{ijk} = \sum_l B_{ikl} C_{lj} \qquad A_{ik} = \sum_j B_{ijk} c_j \qquad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij} \\ &A_{ij} = (\sum_k B_{ijk} C_{ijk}) + D_{ij} \\ &C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}} \overline{P_{il}} \qquad \tau = \sum_i z_i (\sum_j z_j \theta_{ij}) (\sum_k z_k \theta_{ik}) \\ &a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} \overline{P_{no}} \overline{M_{po}} \overline{P_{ip}} \end{aligned}$$

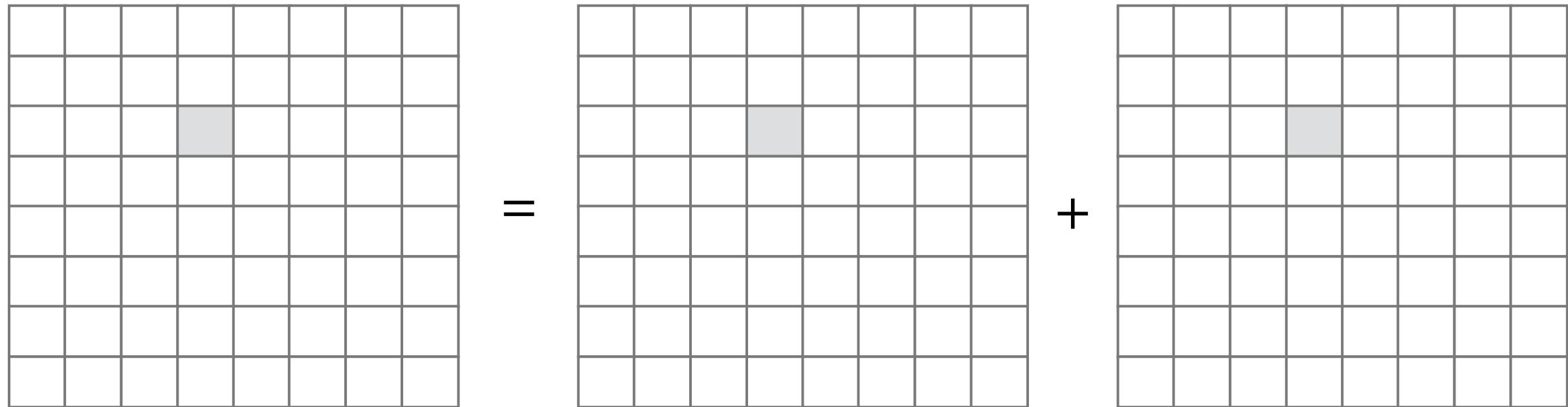
Formats

Schedule



Tensor index notation for expressing functionality

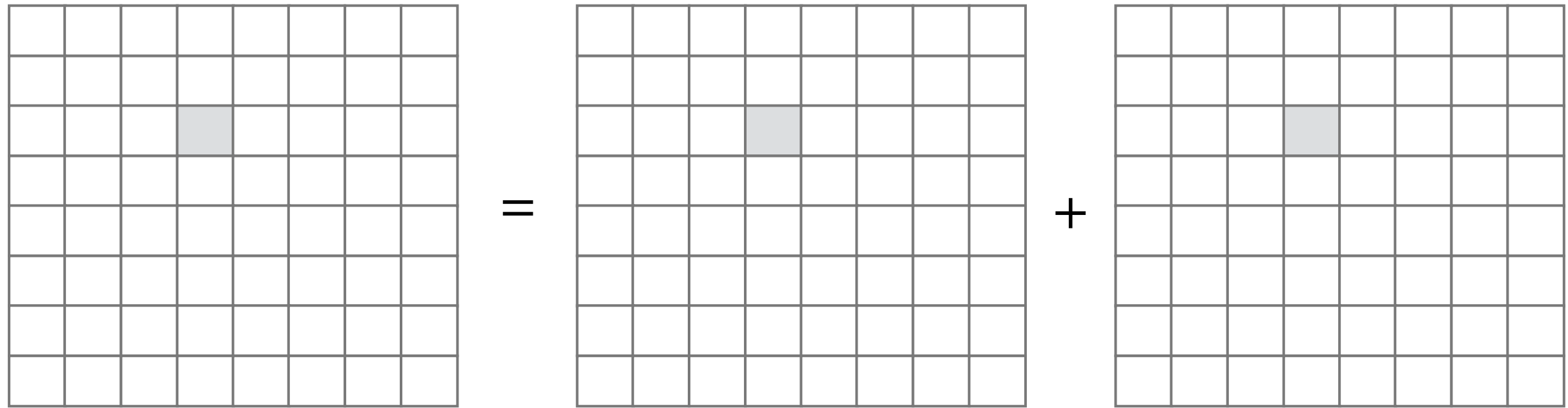
$$A_{ij} = B_{ij} + C_{ij}$$



element-wise

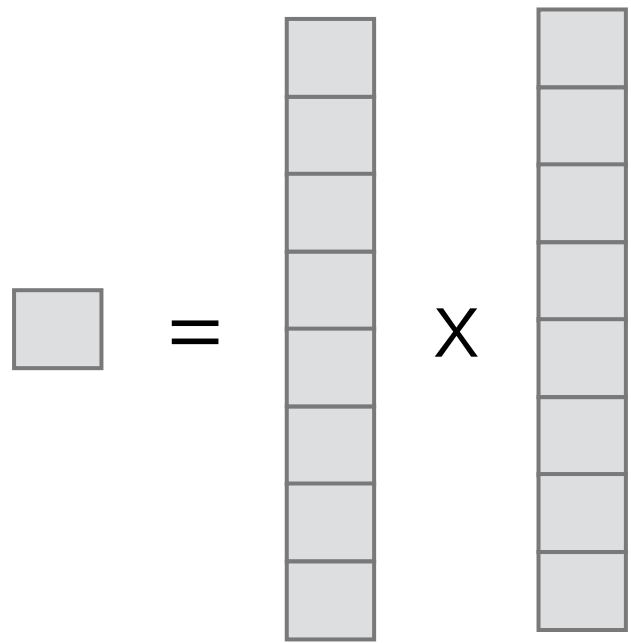
Tensor index notation for expressing functionality

$$A_{ij} = B_{ij} + C_{ij}$$



element-wise

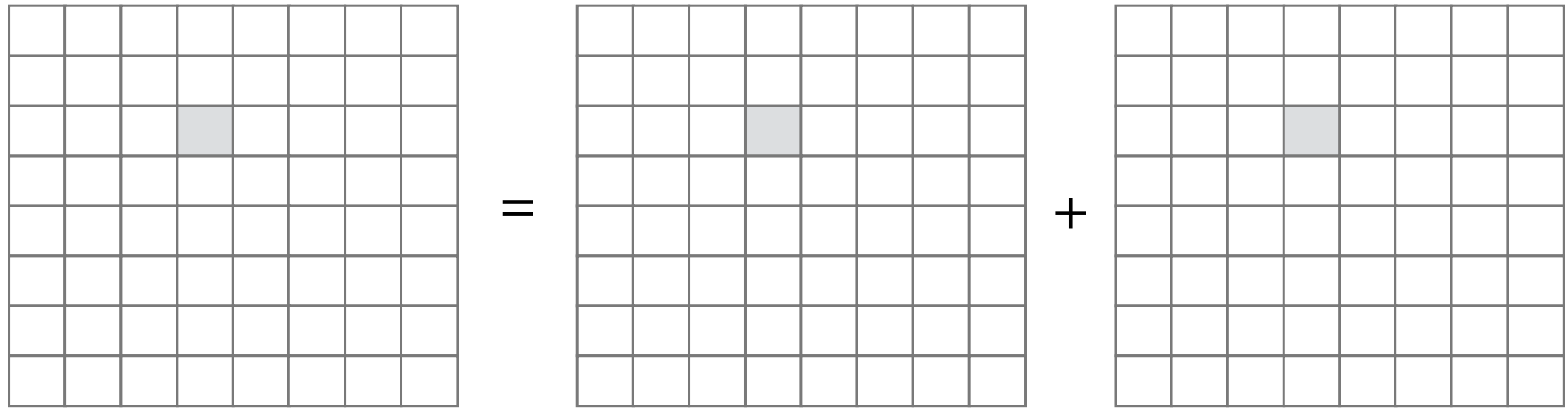
$$\alpha = \sum_i b_i c_i$$



reduction over i

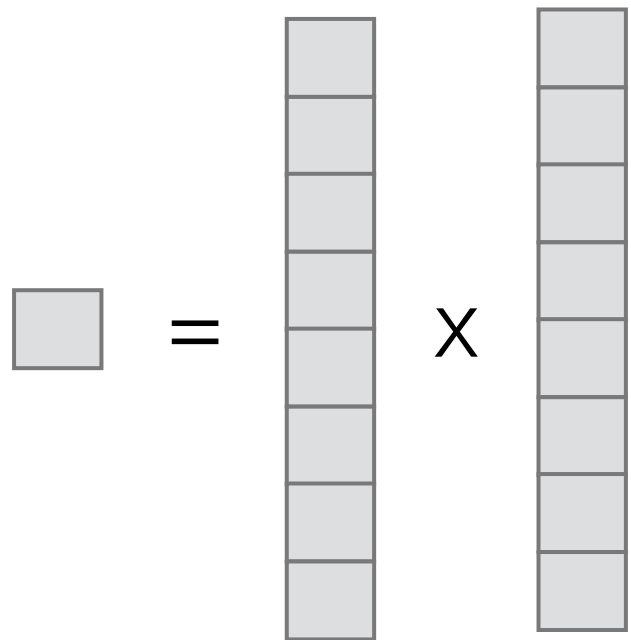
Tensor index notation for expressing functionality

$$A_{ij} = B_{ij} + C_{ij}$$



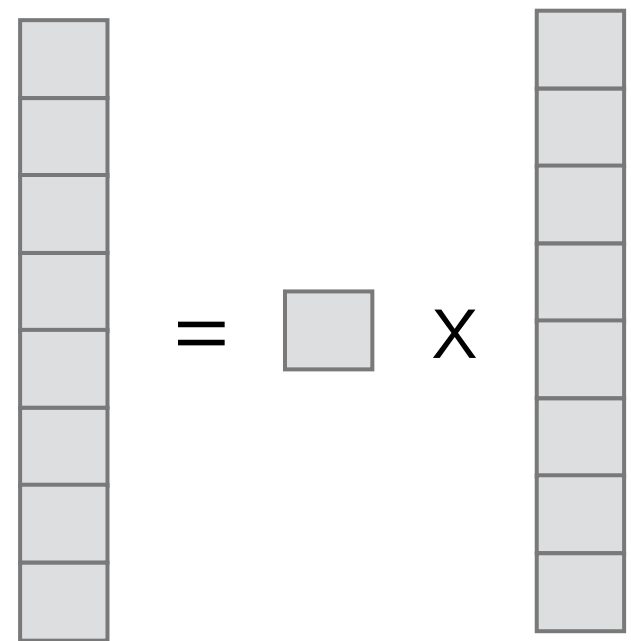
element-wise

$$\alpha = \sum_i b_i c_i$$



reduction over i

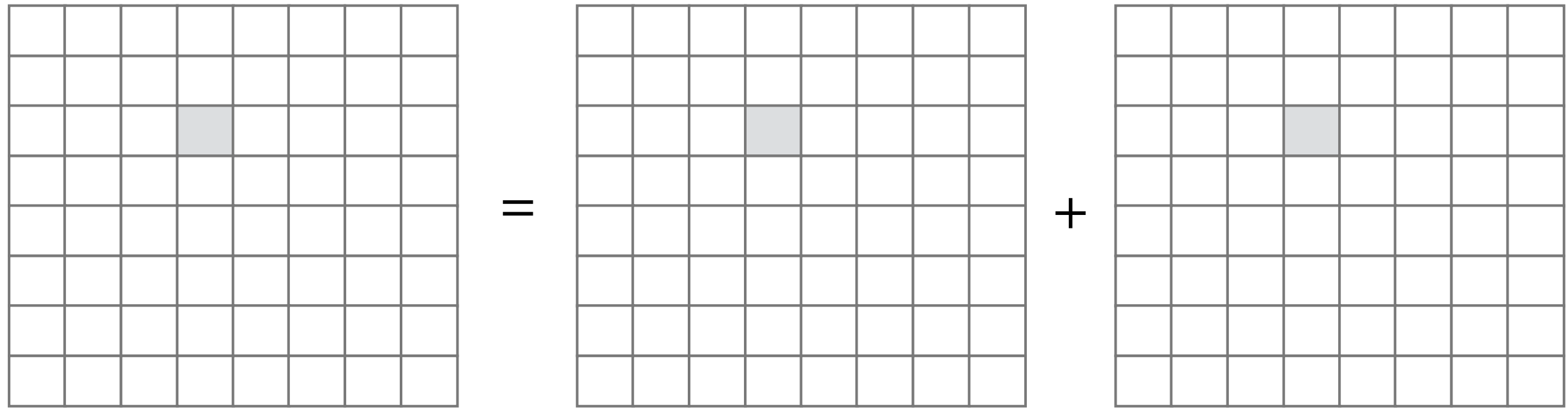
$$a_i = \beta c_i$$



broadcast β over c_i

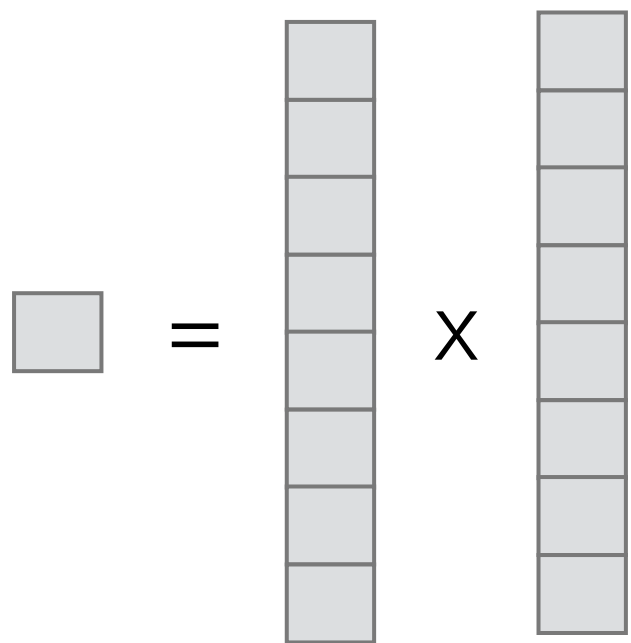
Tensor index notation for expressing functionality

$$A_{ij} = B_{ij} + C_{ij}$$



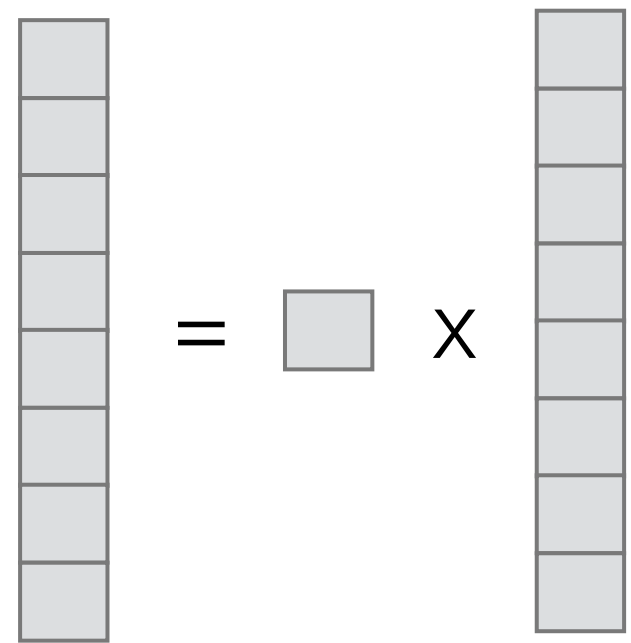
element-wise

$$\alpha = \sum_i b_i c_i$$



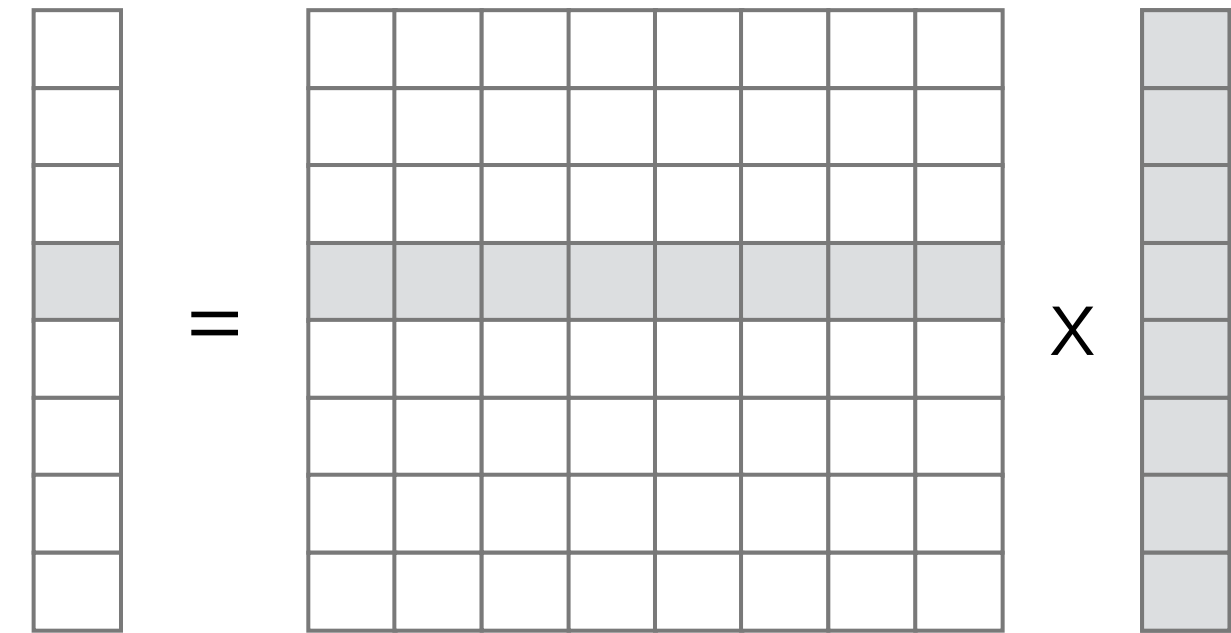
reduction over i

$$a_i = \beta c_i$$



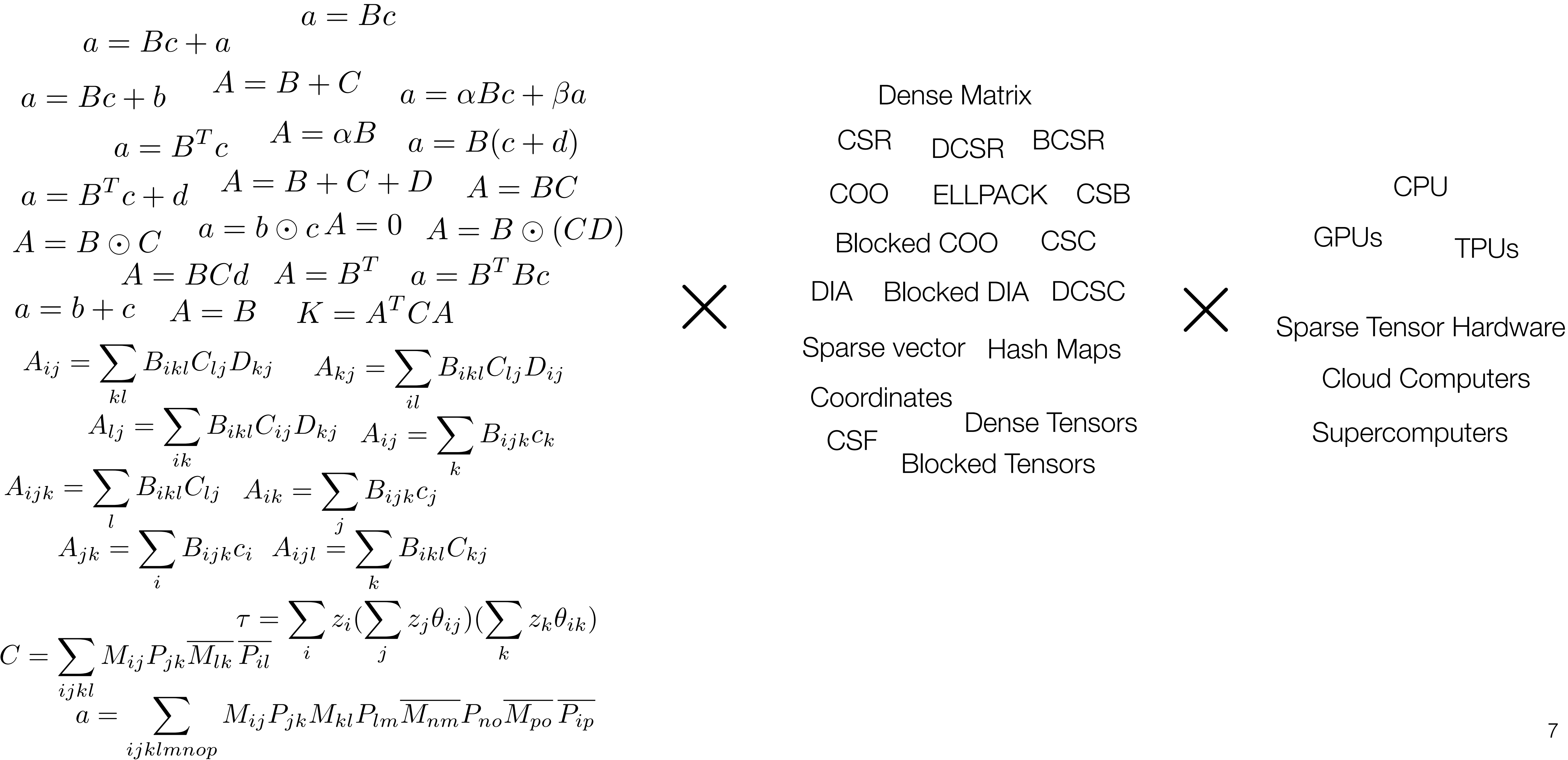
broadcast β over c_i

$$a_i = \sum_j B_{ij} c_j$$



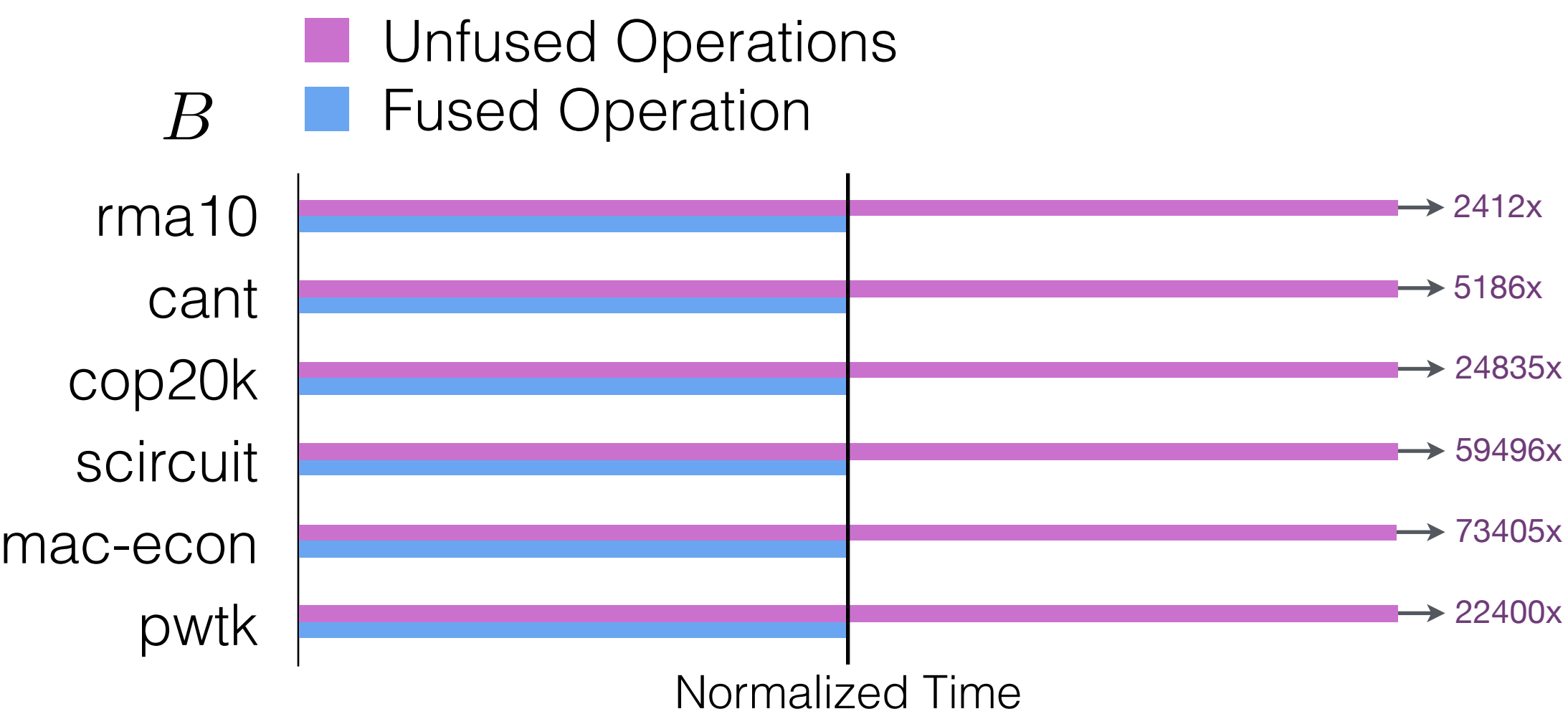
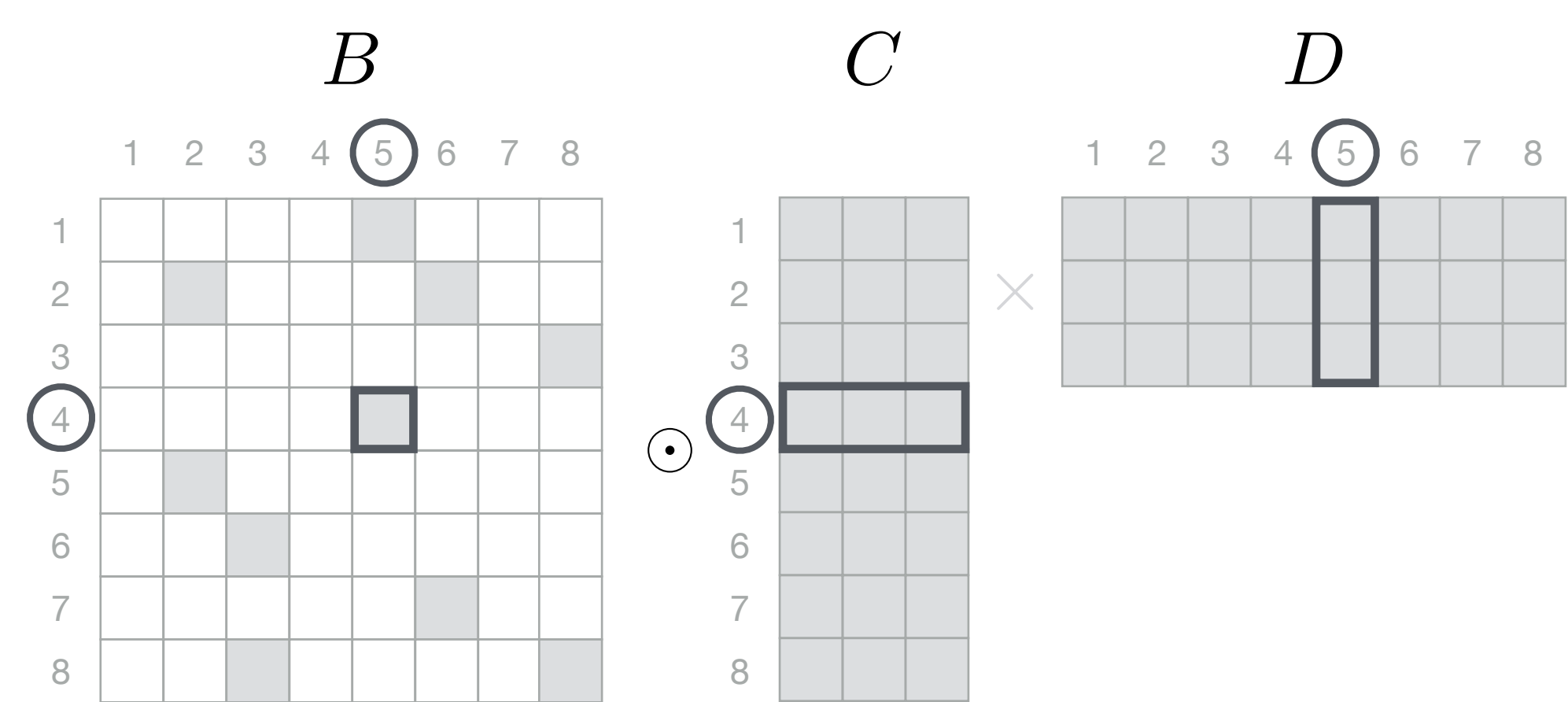
all three patterns

Generates fast code for any tensor index notation expression with the given formats and schedule



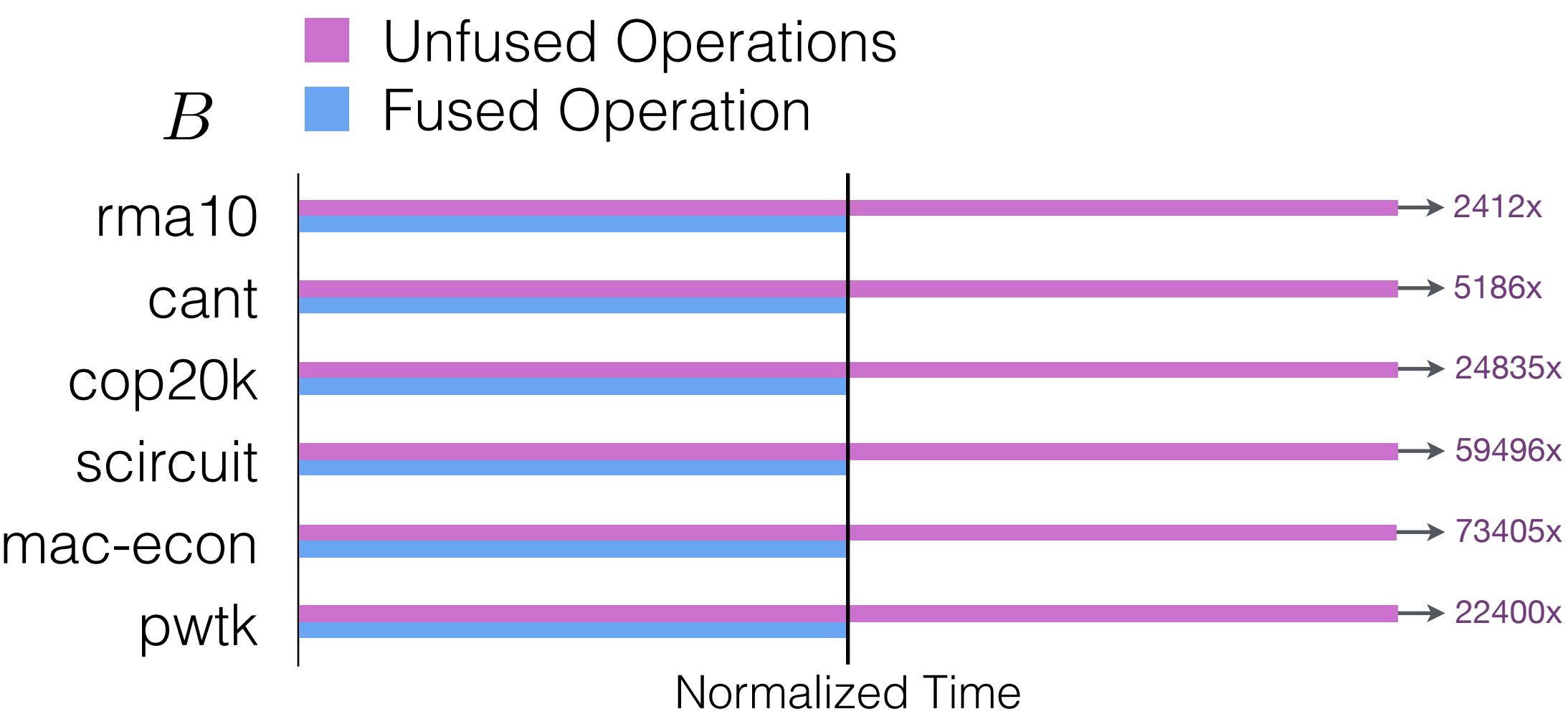
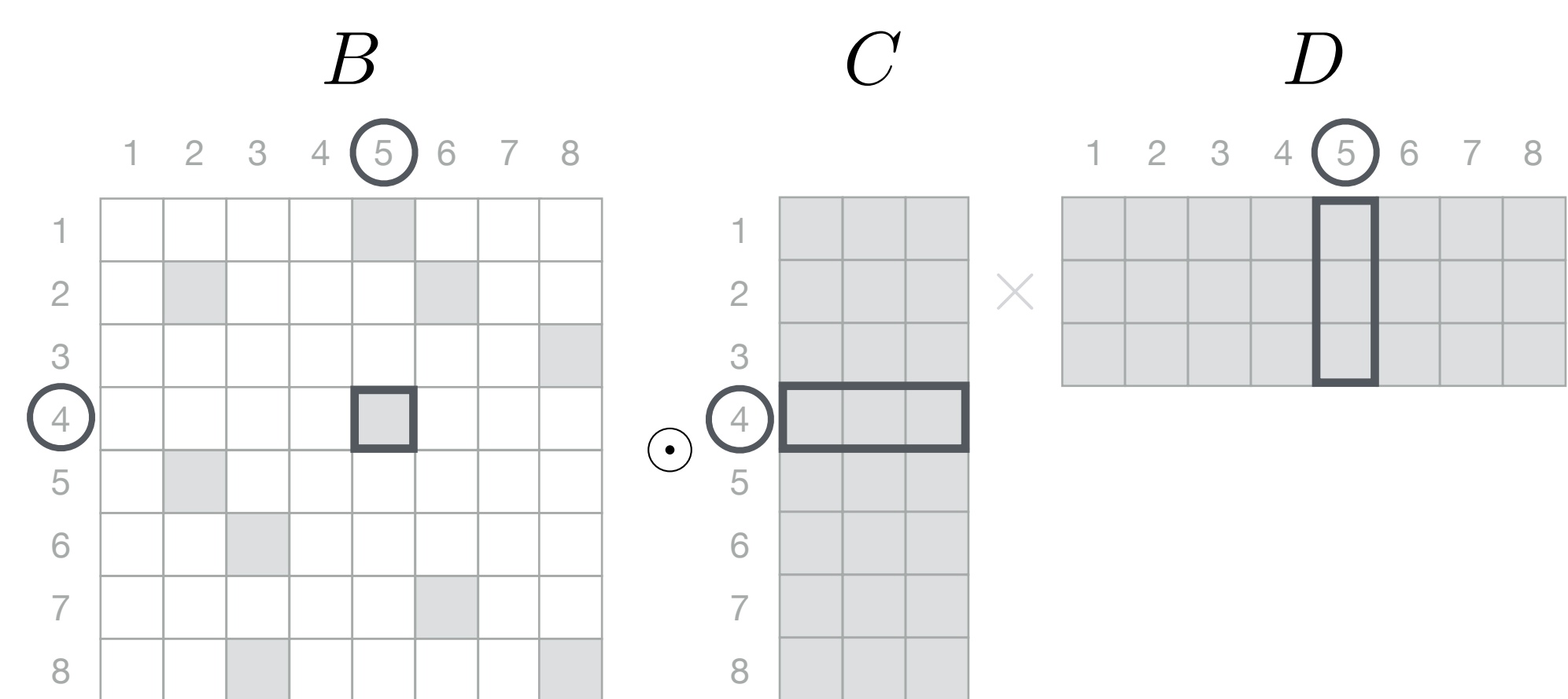
Compound expressions matter for performance

$$A = B \odot (CD)$$



Compound expressions matter for performance

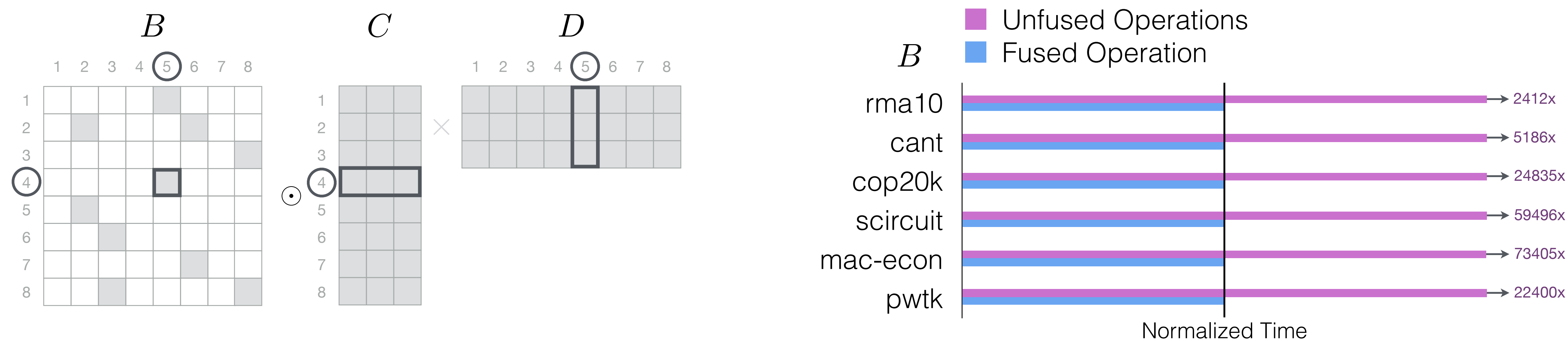
$$A = B \odot (CD)$$



Unfused: $\Theta(n^2k)$

Compound expressions matter for performance

$$A = B \odot (CD)$$

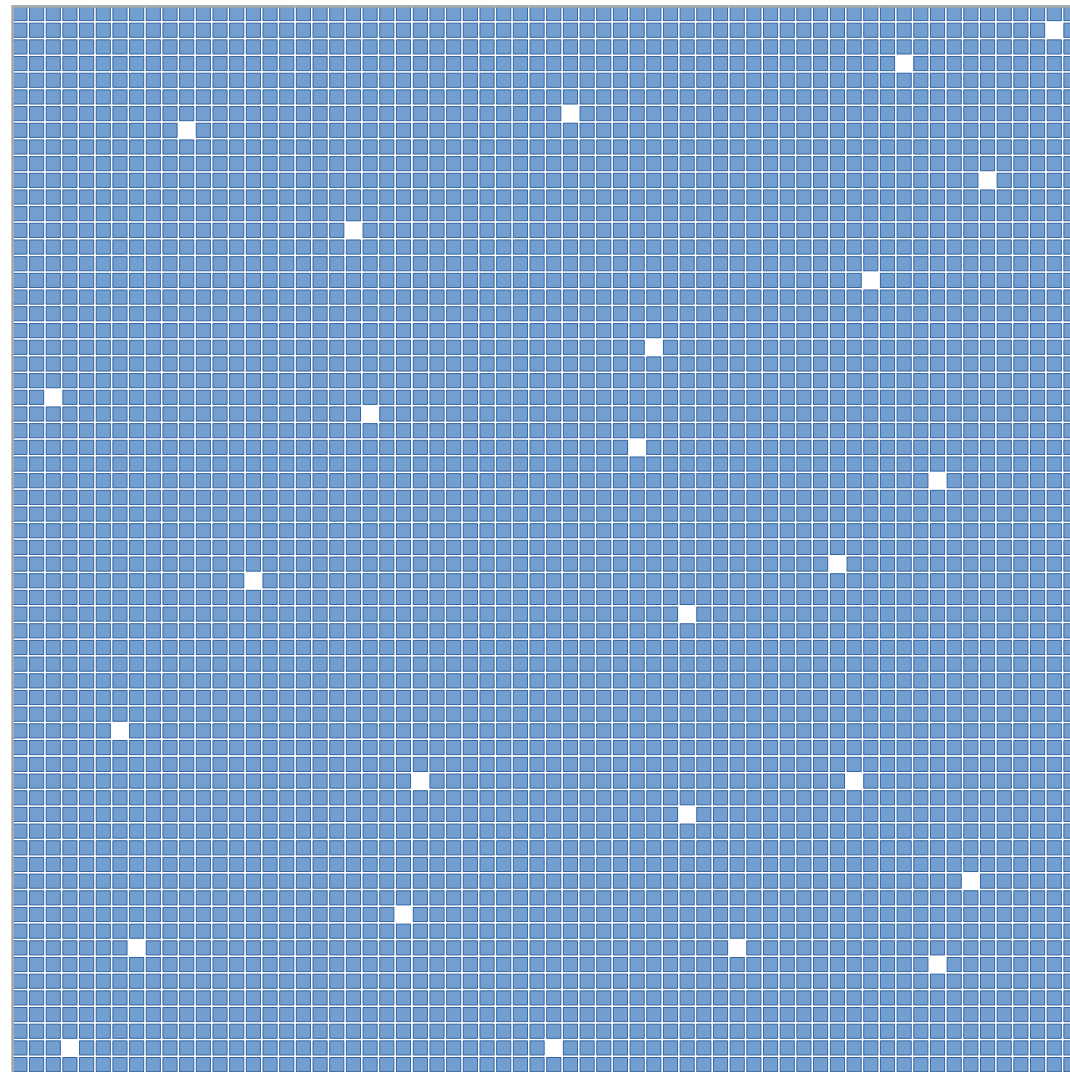


Unfused: $\Theta(n^2k)$

Fused: $\Theta(\text{nnz}_B \cdot k)$

Formats matter for performance

Dense Matrix



Formats

Best performance

Dense

List of Rows

CSR

DCSR

$$y = Ax$$



Normalized time

Formats matter for performance

Thermal Matrix



Formats

Best performance

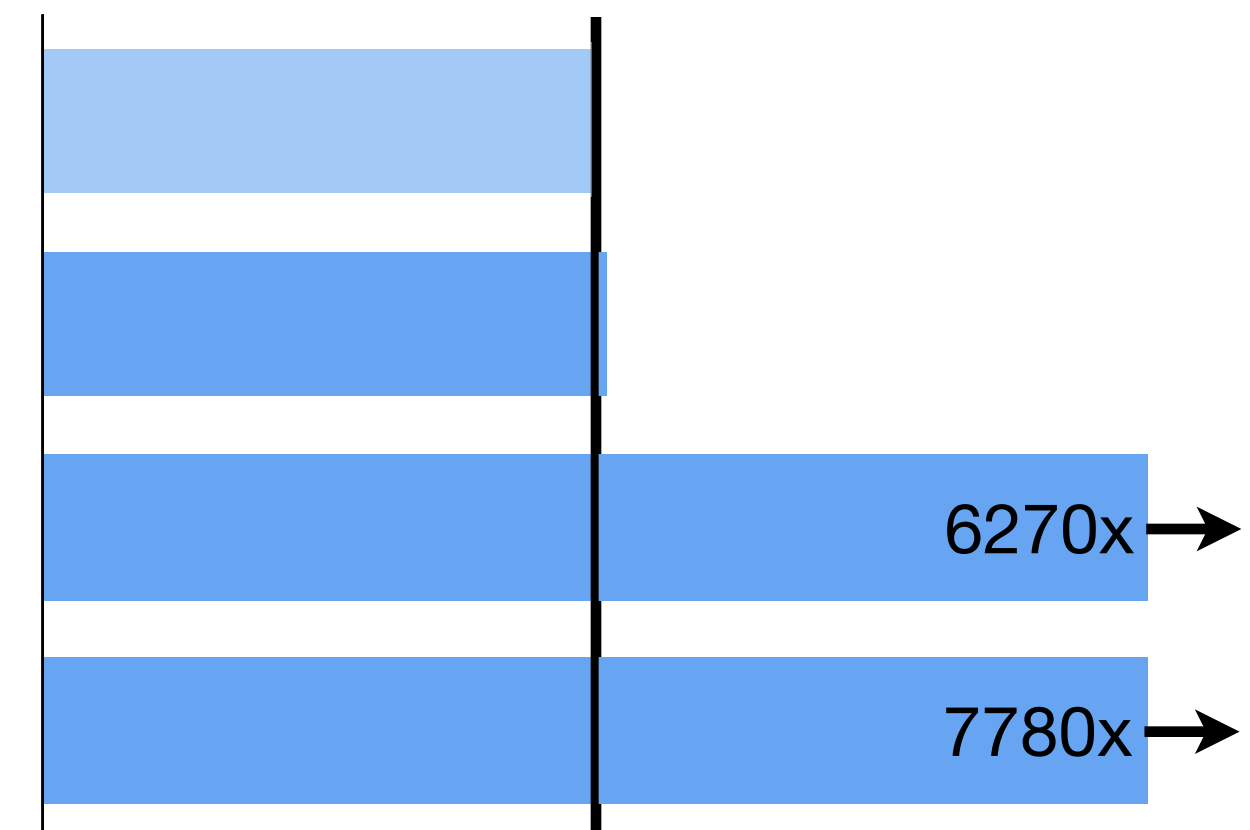
CSR

DCSR

Dense

List of Rows

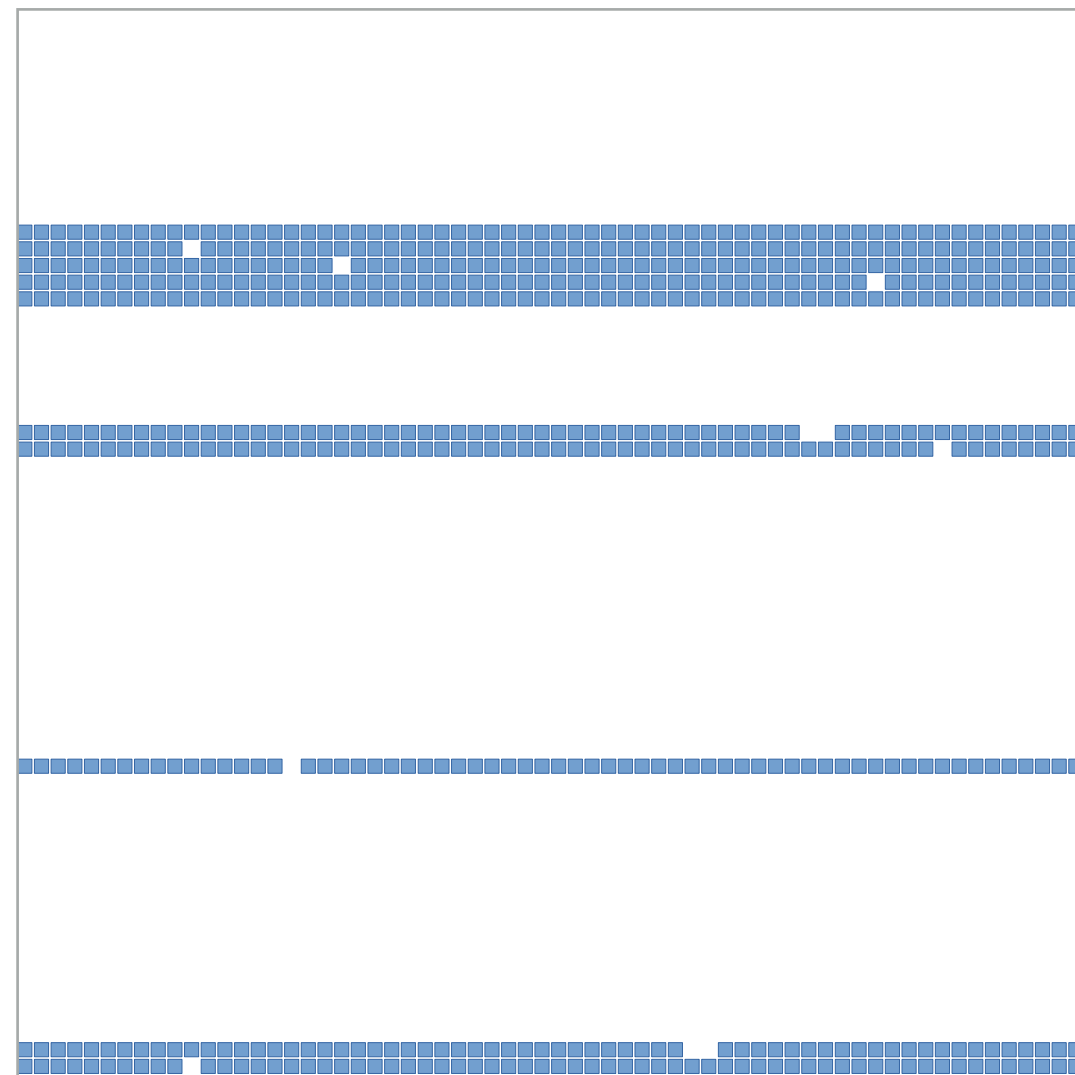
$$y = Ax$$



Normalized time

Formats matter for performance

Row-sliced Matrix



Formats

Best performance

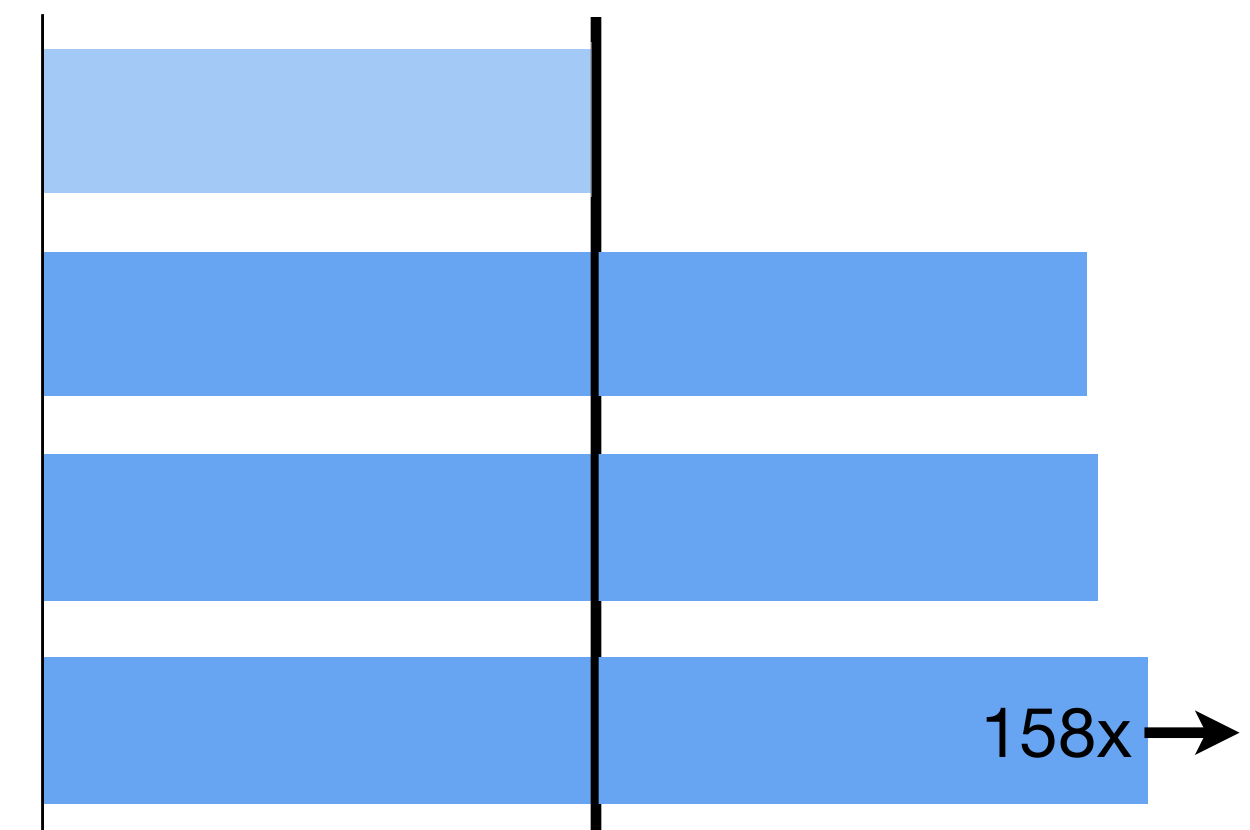
List of Rows

DCSR

CSR

Dense

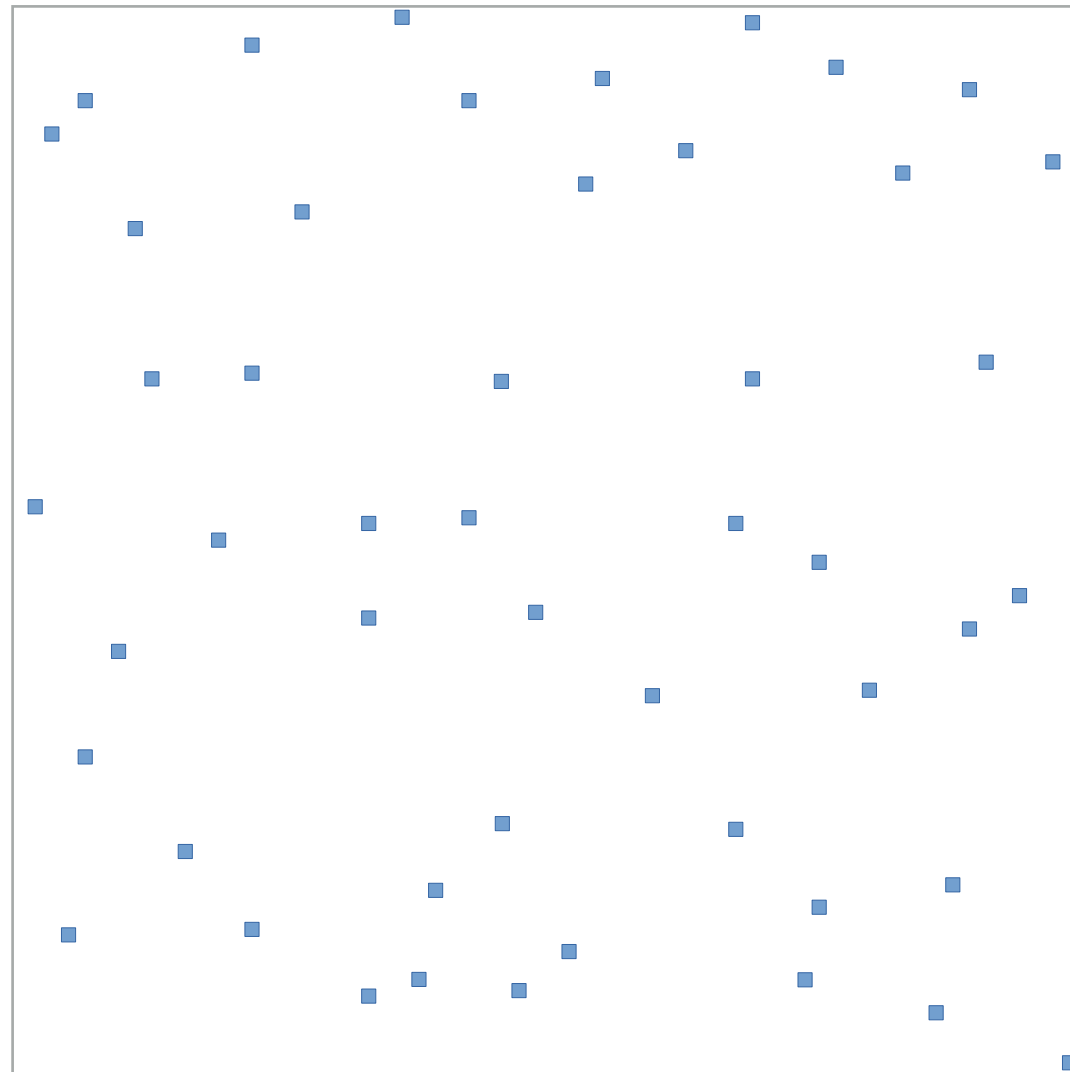
$$y = Ax$$



Normalized time

Formats matter for performance

Hypersparse Matrix



Formats

Best performance

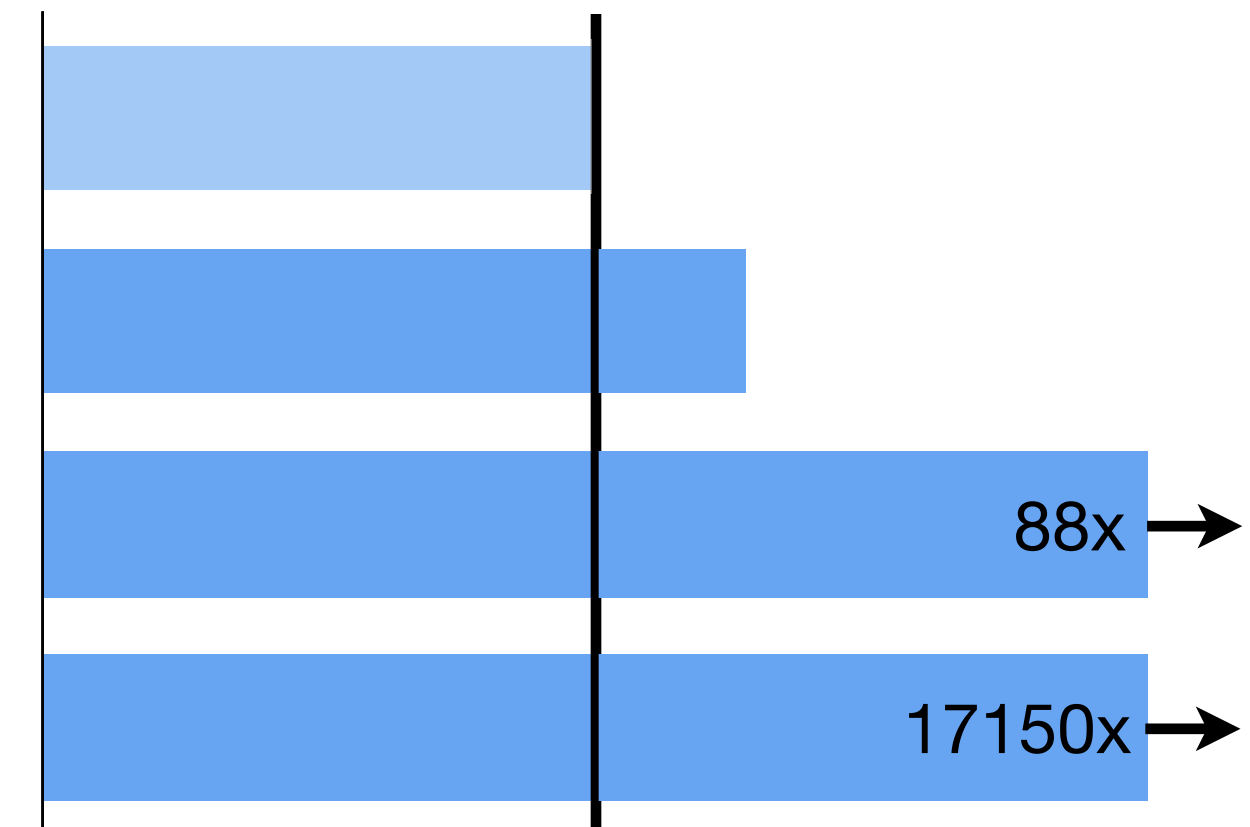
DCSR

CSR

List of Rows

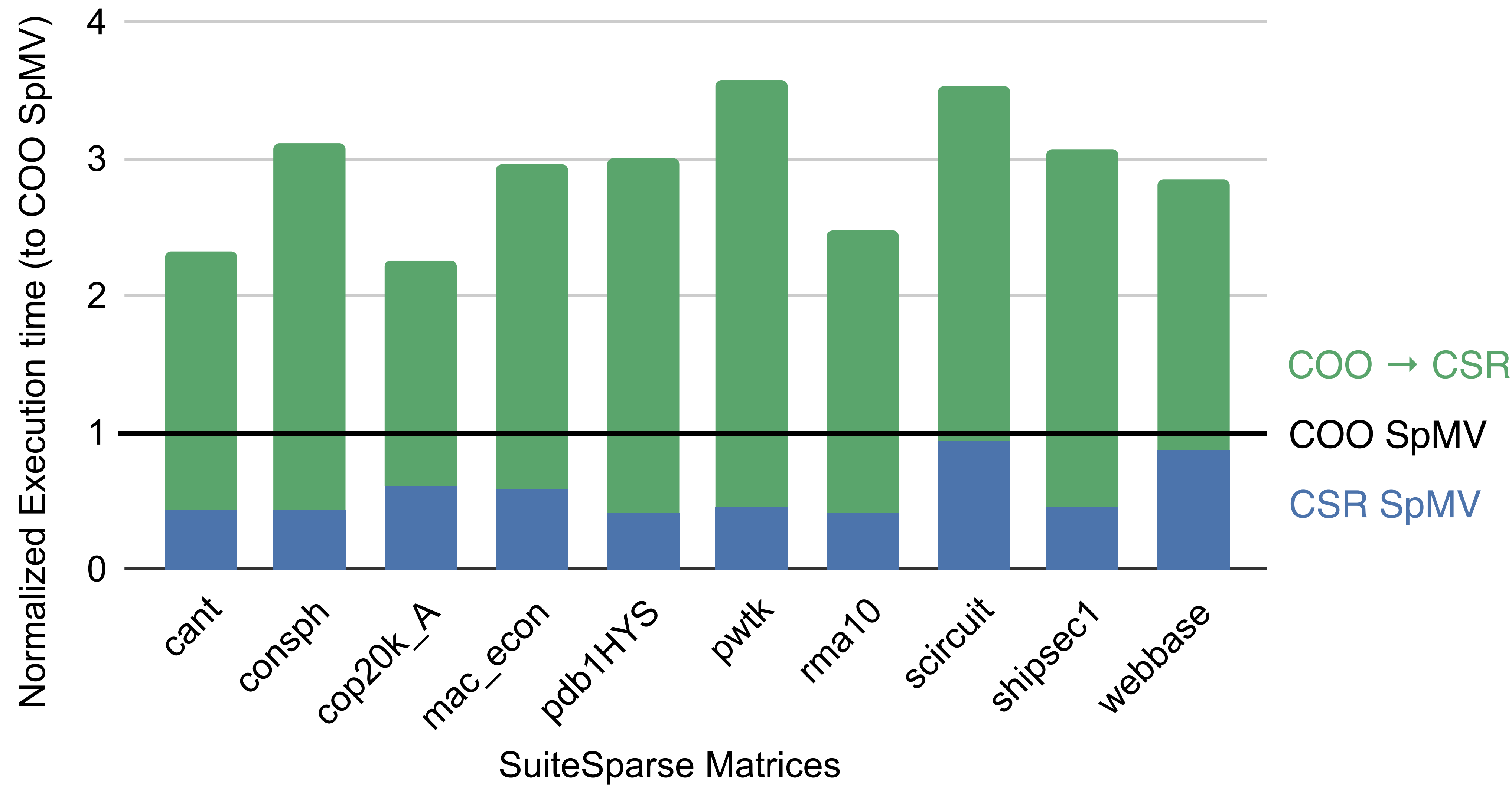
Dense

$$y = Ax$$



Normalized time

CSR vs COO

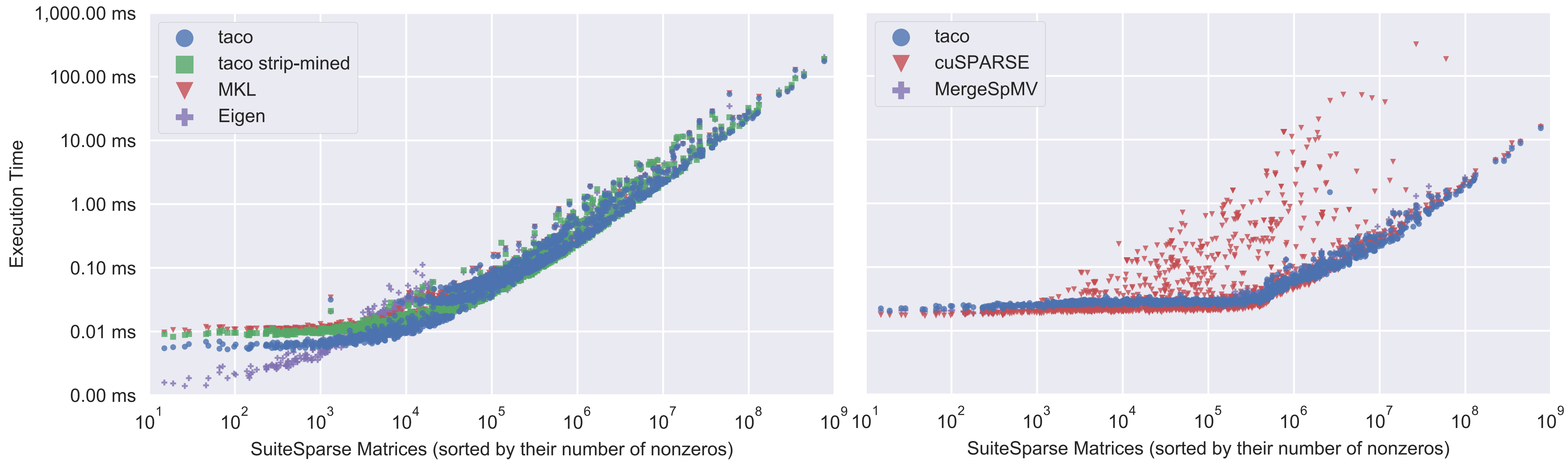


Machines matter for performance

$$a_i = \sum_j B_{ij} c_j \quad (\text{SpMV})$$

CPU

GPU



Sparse data structures in graphs, tensors, and relations encode coordinates in a sparse iteration space

Tensor (nonzeros)

(0,1)
(2,3) (0,5)
(5,5) (7,5)

Relation (rows)

(Harry,CS) (Sally,EE)
(George,CS) (Mary,ME)
(Rita,CS)

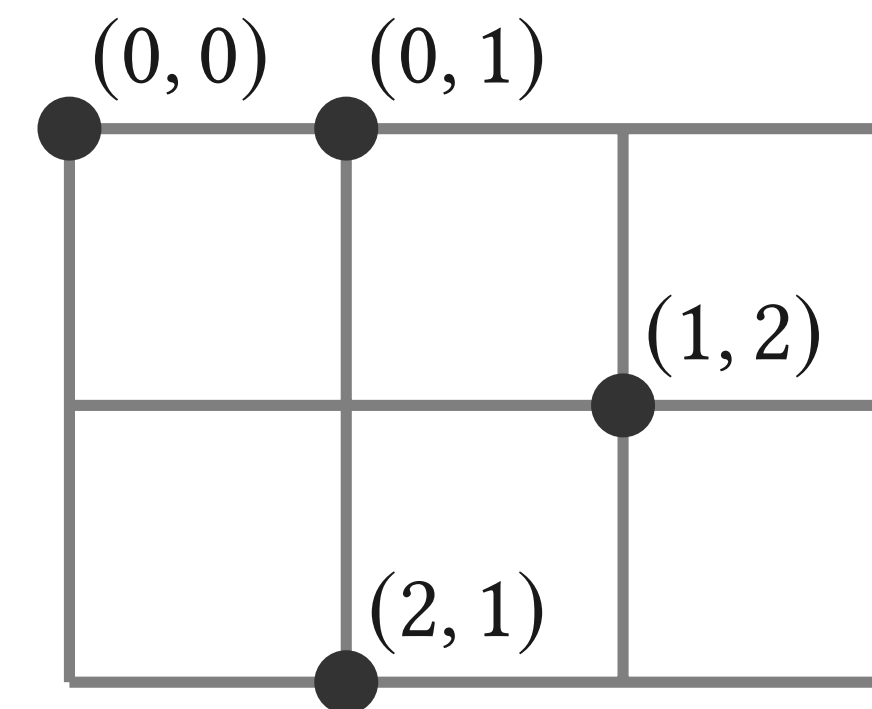
Graph (edges)

(v₁,v₅) (v₄,v₃)
(v₅,v₃)
(v₃,v₅) (v₃,v₁)

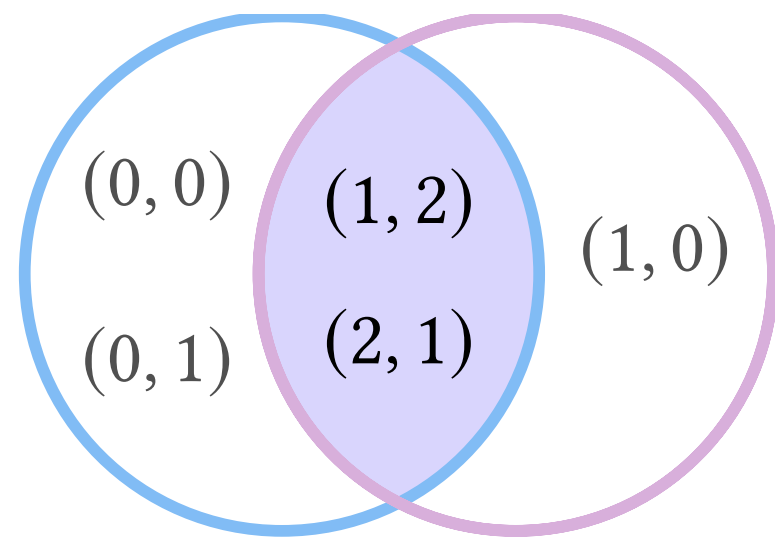
Values may be attached to these coordinates: e.g., nonzero values, edge attributes

Iteration spaces from coordinate relations

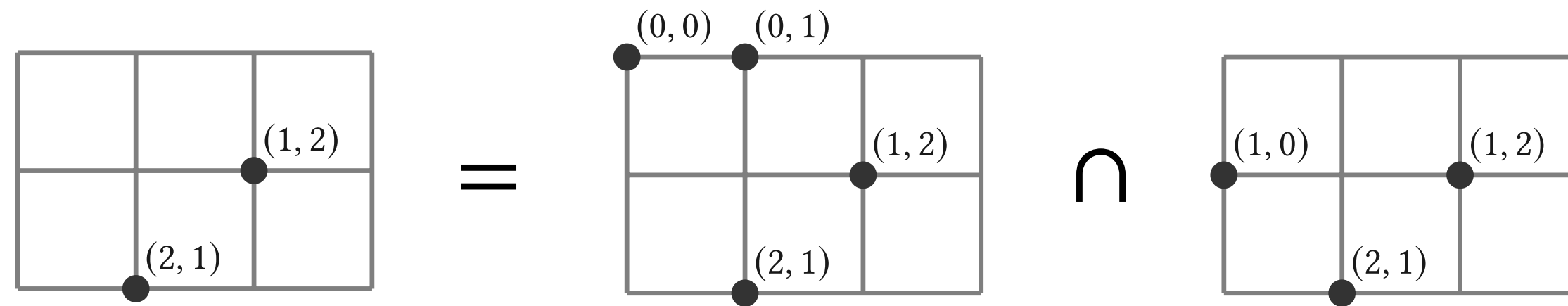
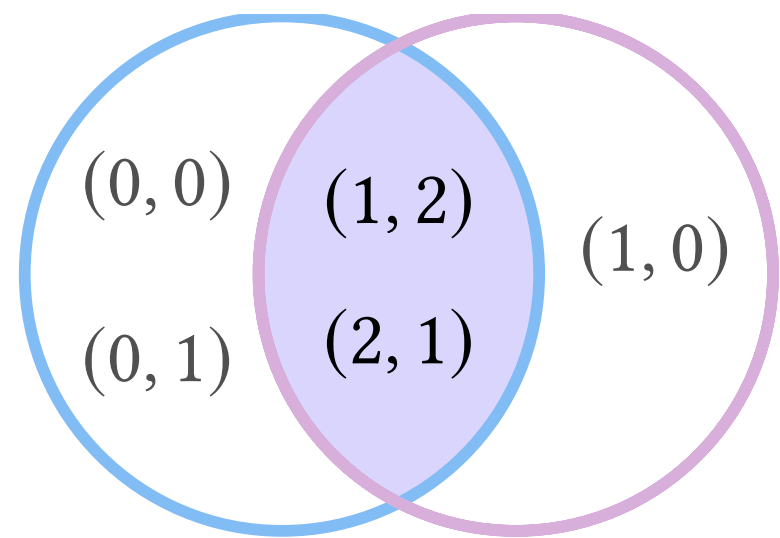
$(0, 0)$
 $(0, 1)$
 $(1, 2)$ $(2, 1)$



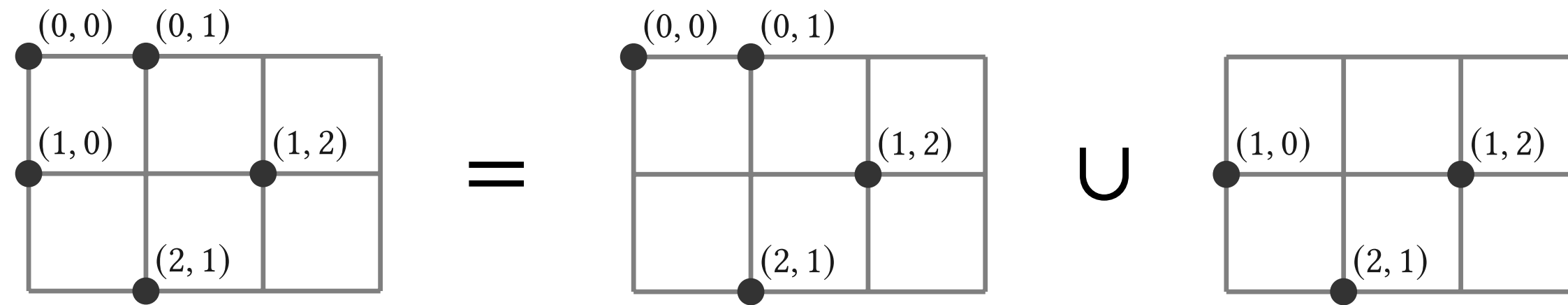
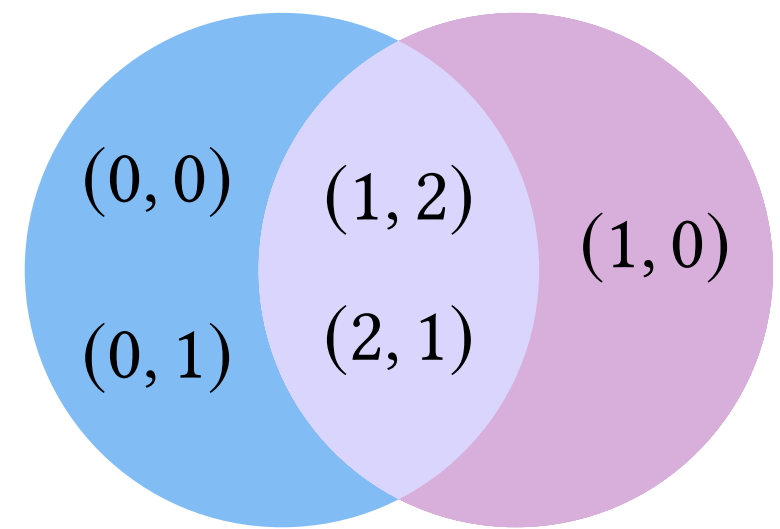
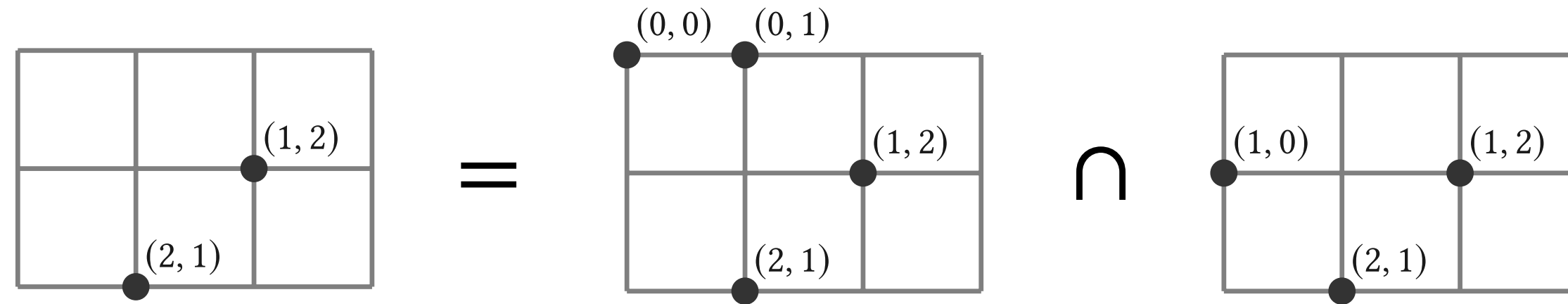
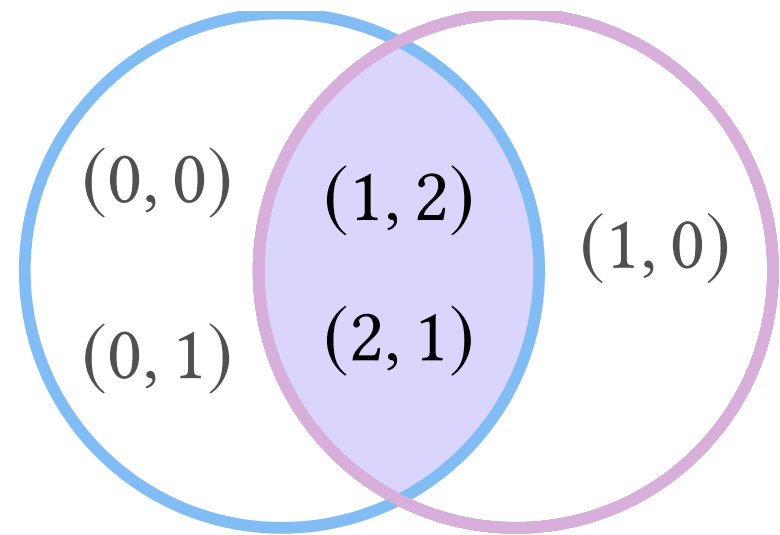
Iteration spaces from set operations



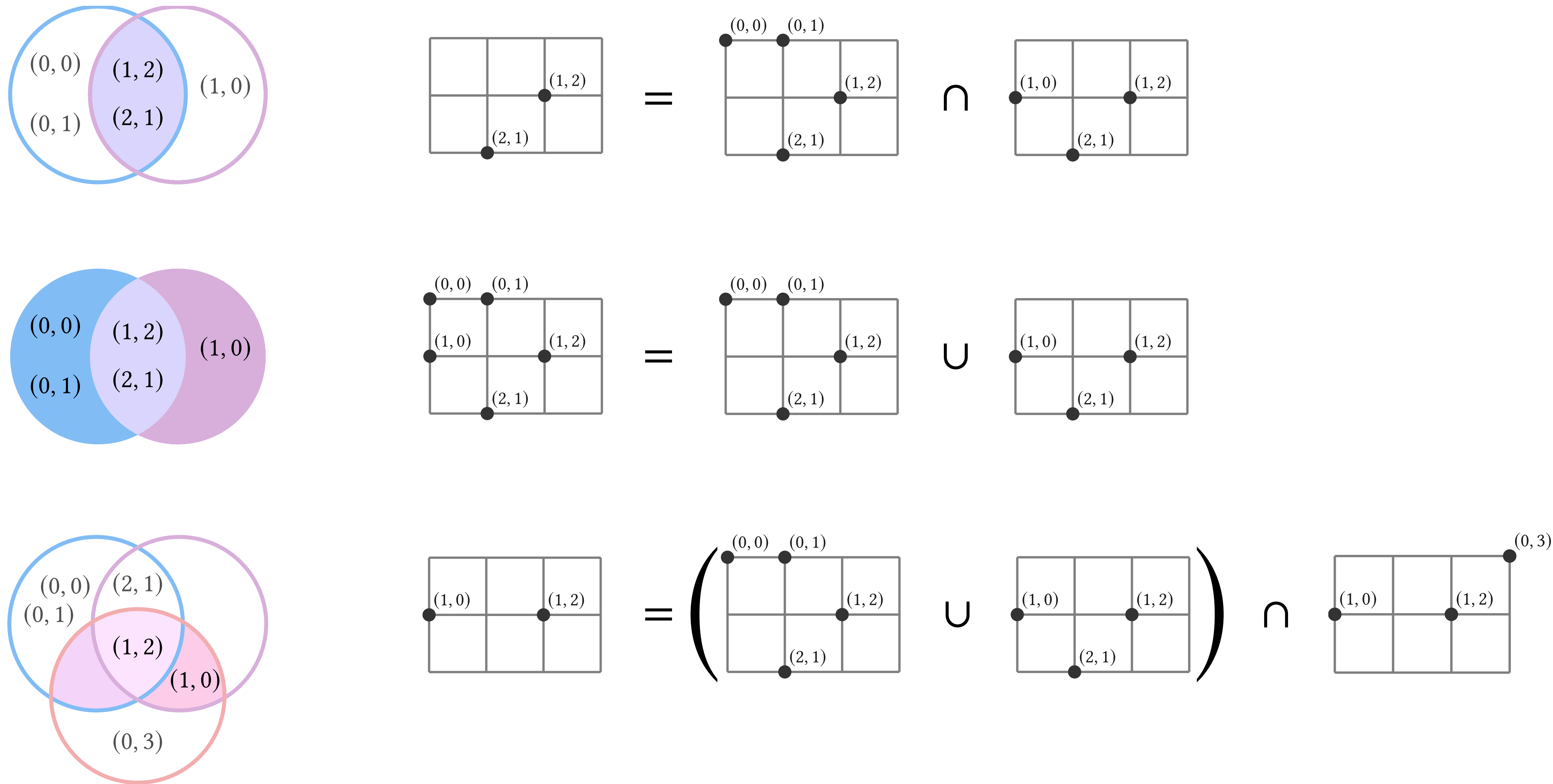
Iteration spaces from set operations



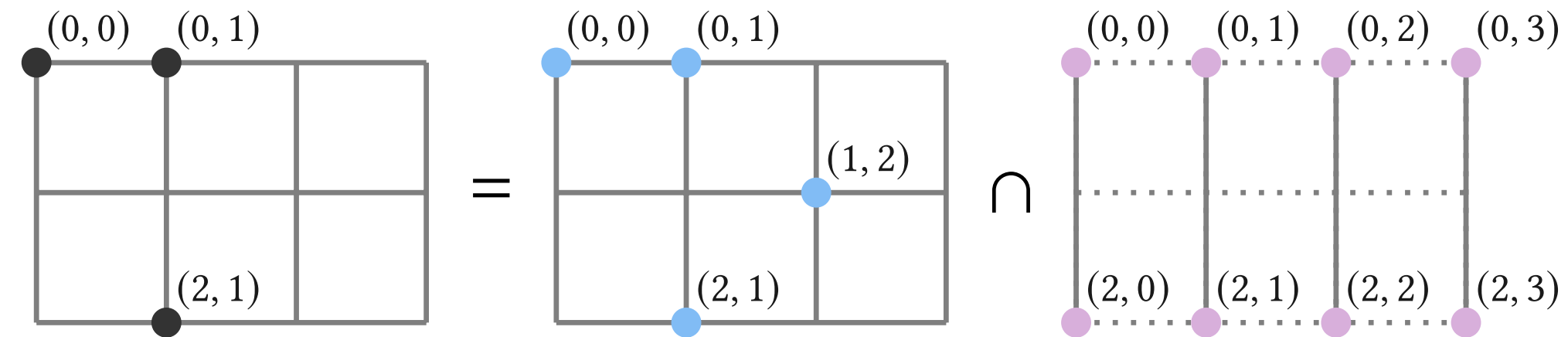
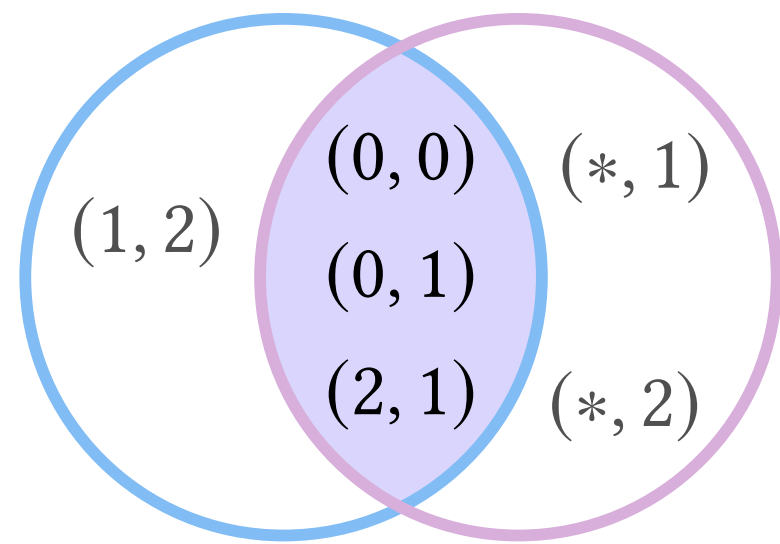
Iteration spaces from set operations



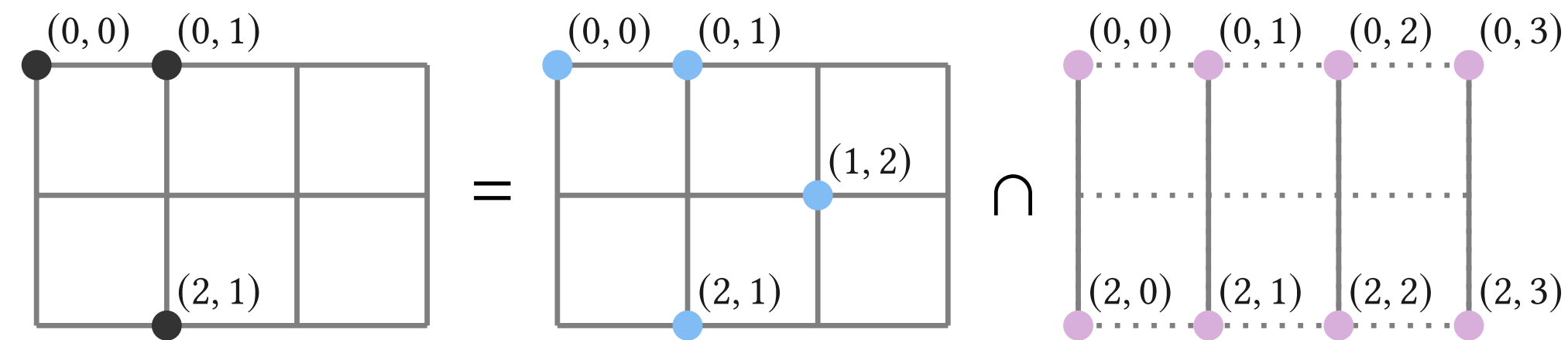
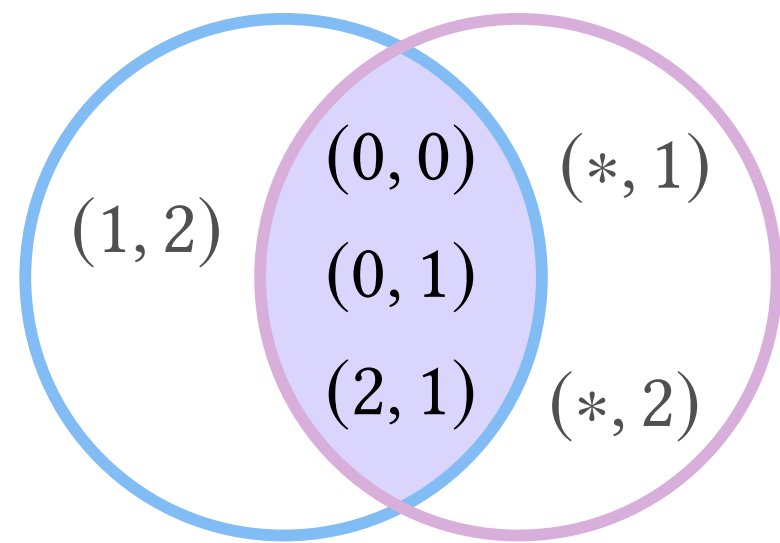
Iteration spaces from set operations



Iteration spaces from broadcast operations

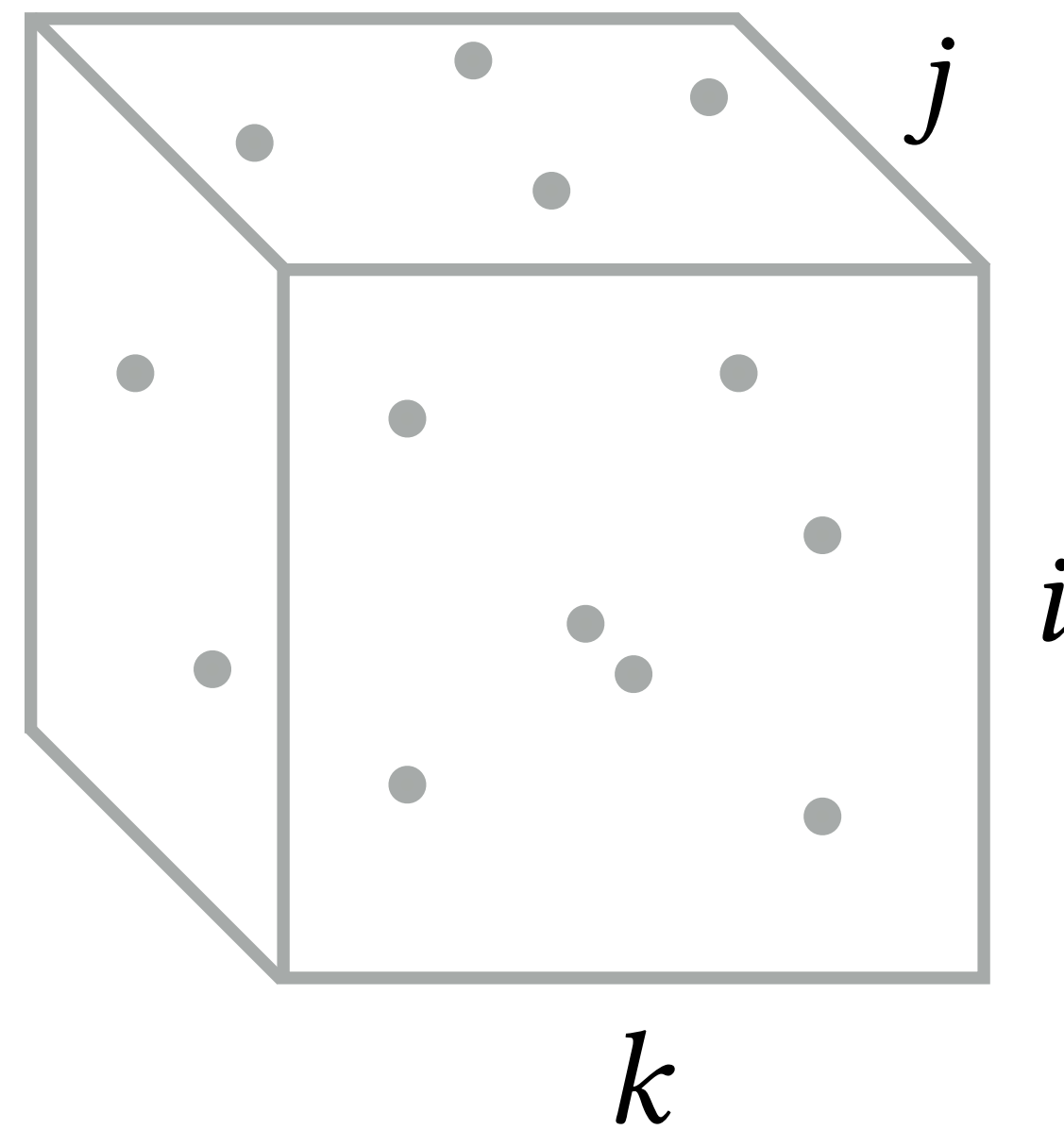


Iteration spaces from broadcast operations

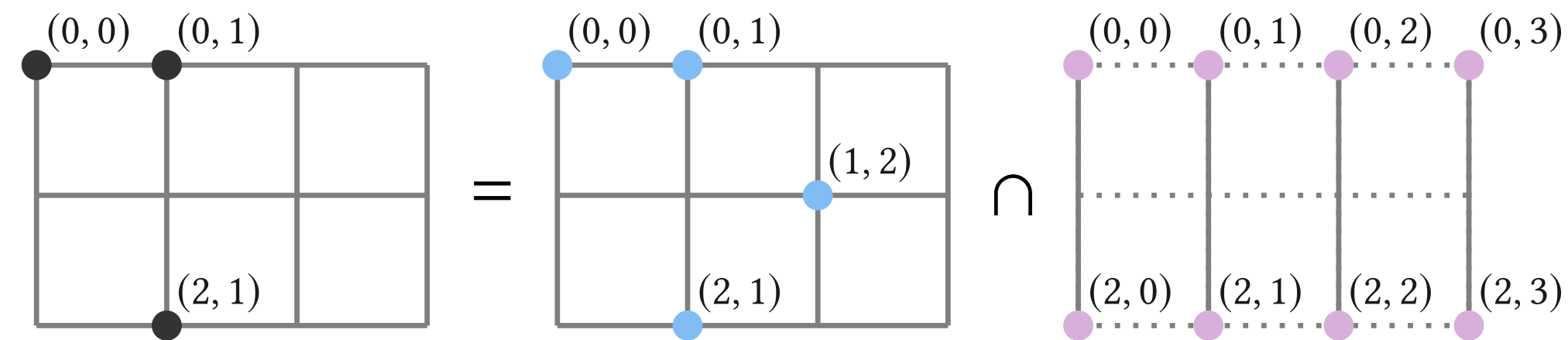
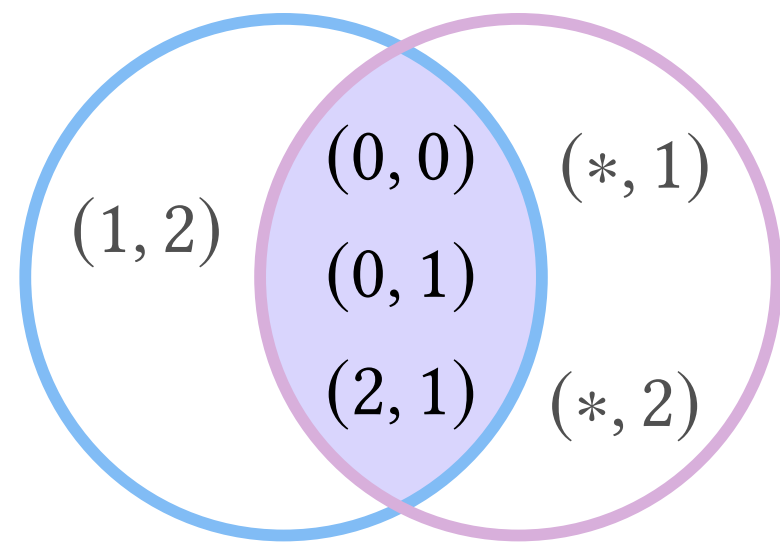


$$A_{ij} = \sum_k B_{ik} C_{kj}$$

$$B_{ik} \cap C_{kj}$$

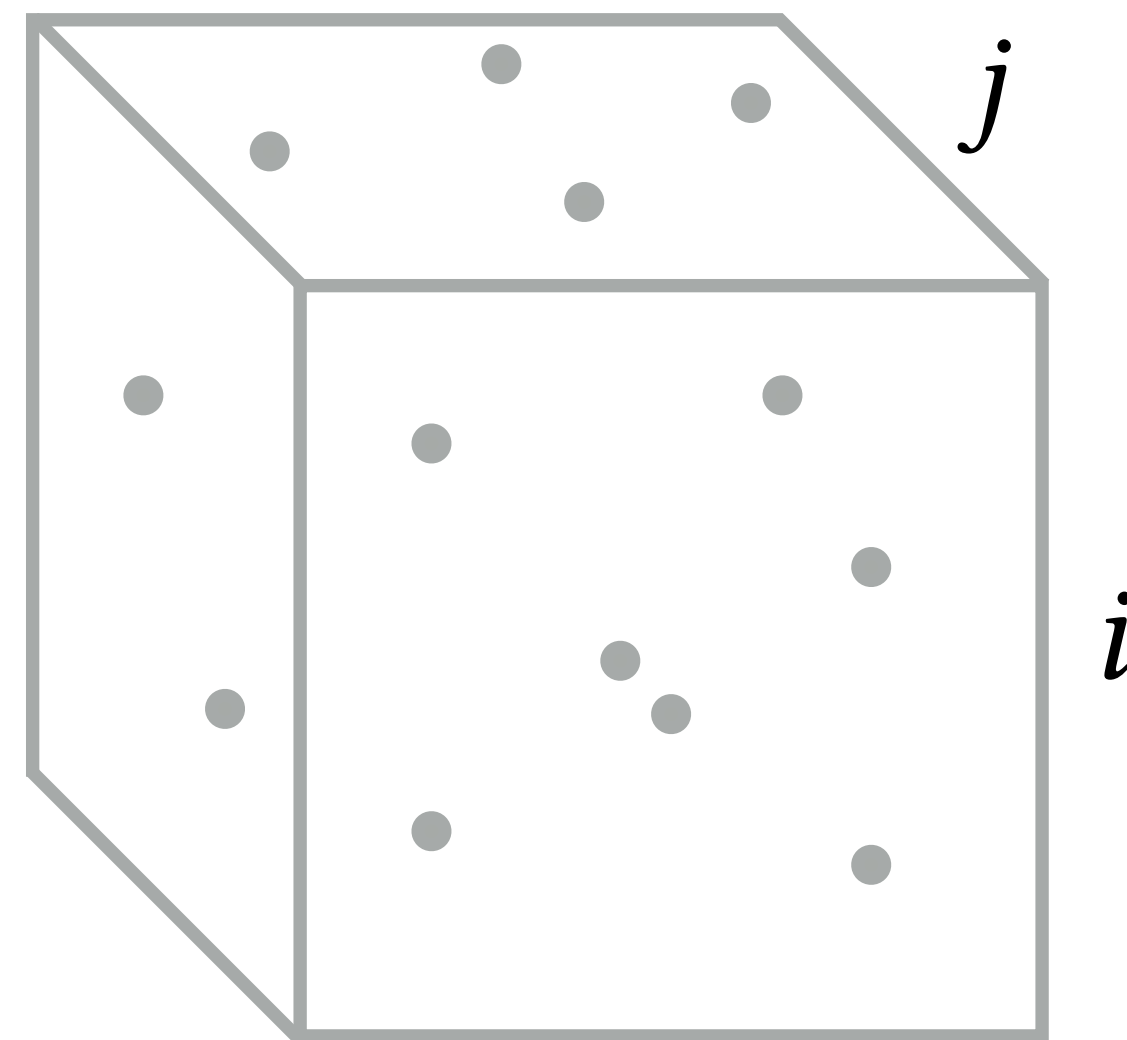


Iteration spaces from broadcast operations



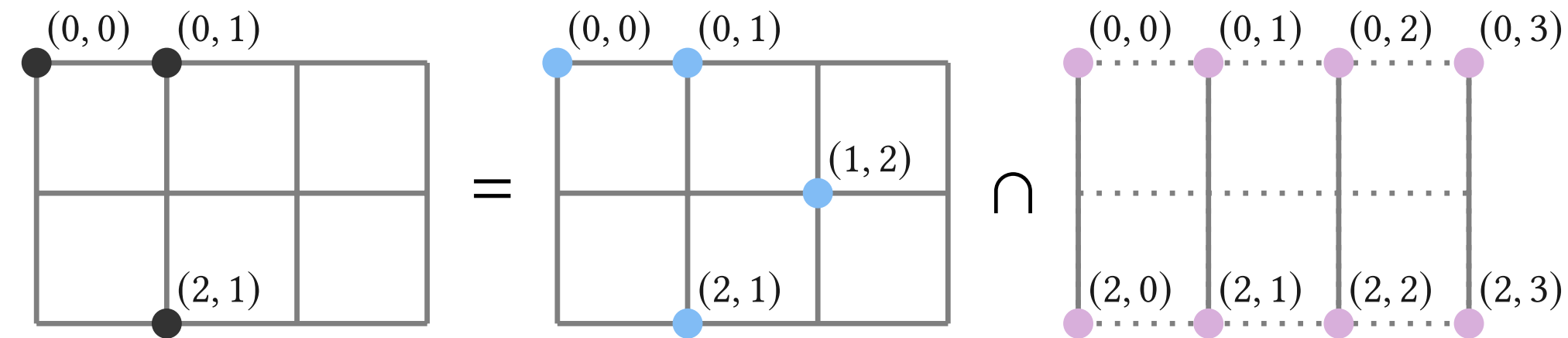
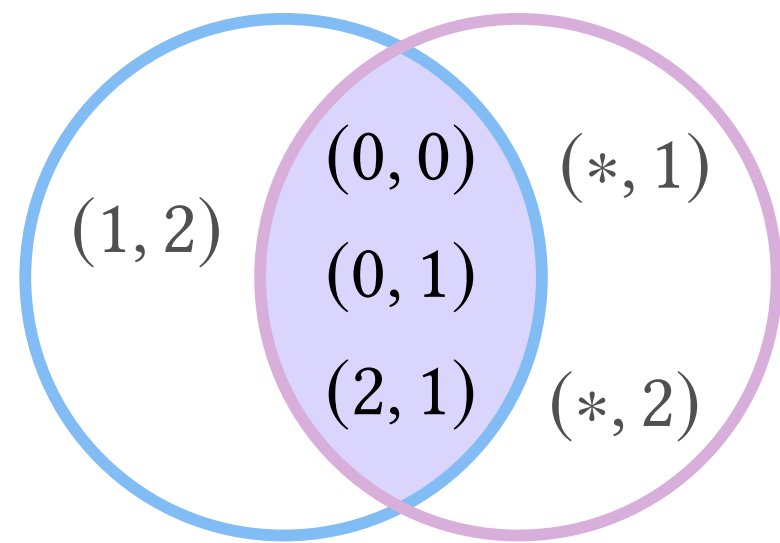
$$A_{ij} = \sum_k B_{ik} C_{kj}$$

$$B_{ik} \cap C_{kj}$$



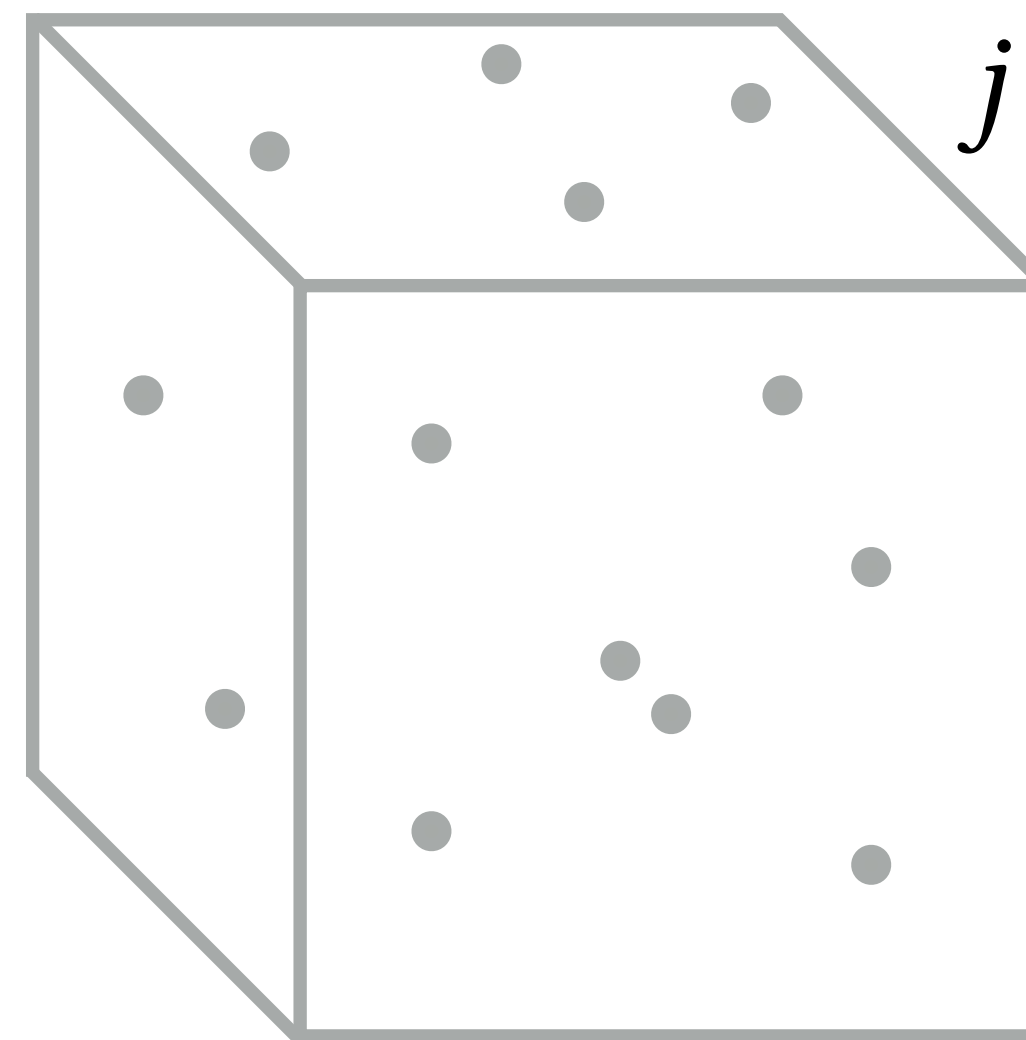
$$k \in B_2 \cap C_1$$

Iteration spaces from broadcast operations



$$A_{ij} = \sum_k B_{ik} C_{kj}$$

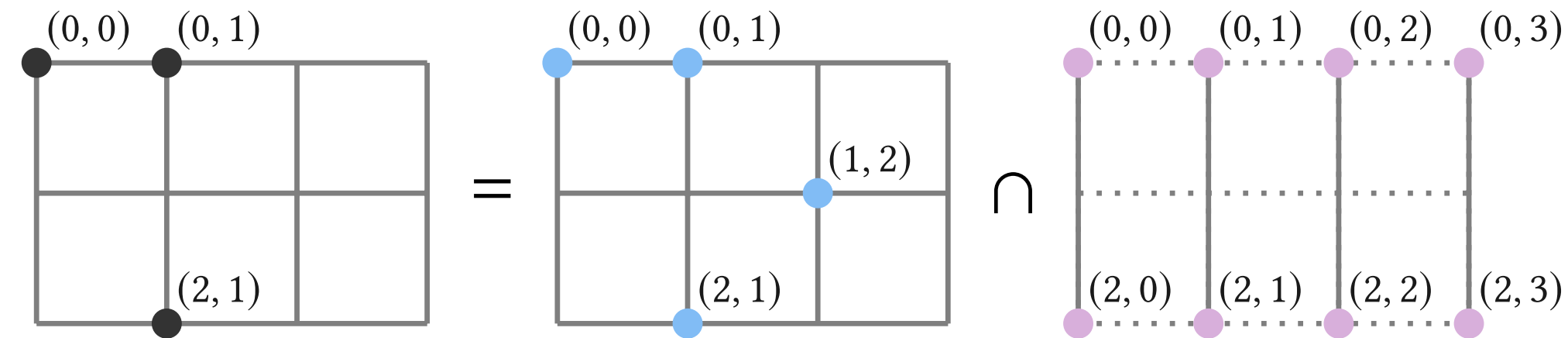
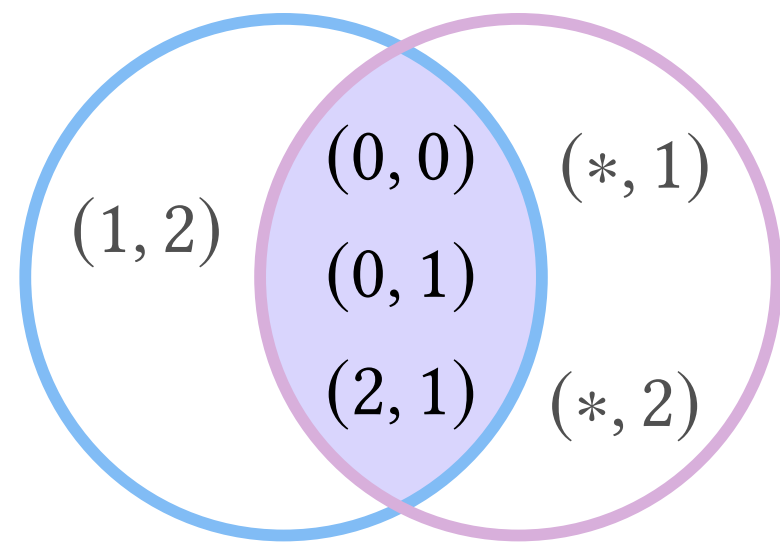
$$B_{ik} \cap C_{kj}$$



$$i \in B_1 \cap \mathbb{U}_i$$

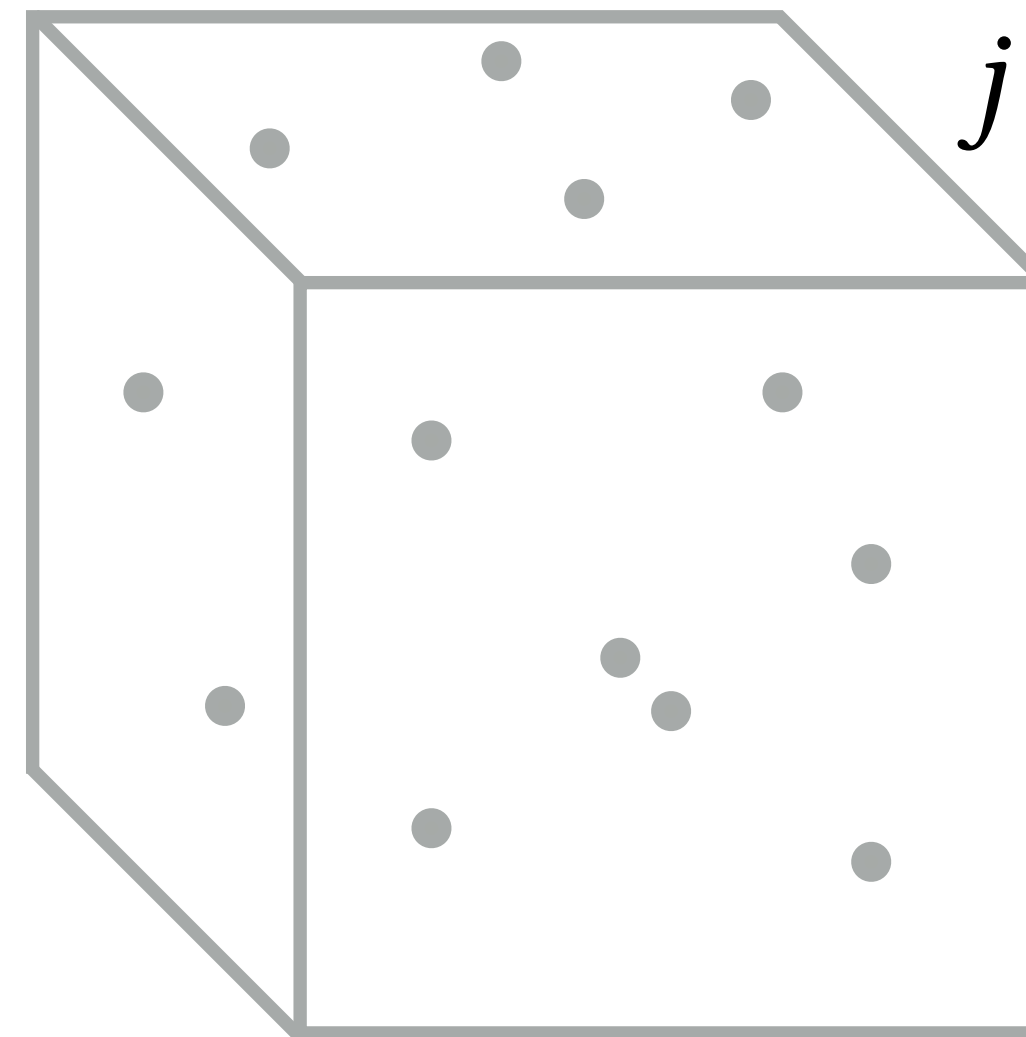
$$k \in B_2 \cap C_1$$

Iteration spaces from broadcast operations



$$A_{ij} = \sum_k B_{ik} C_{kj}$$

$$B_{ik} \cap C_{kj}$$

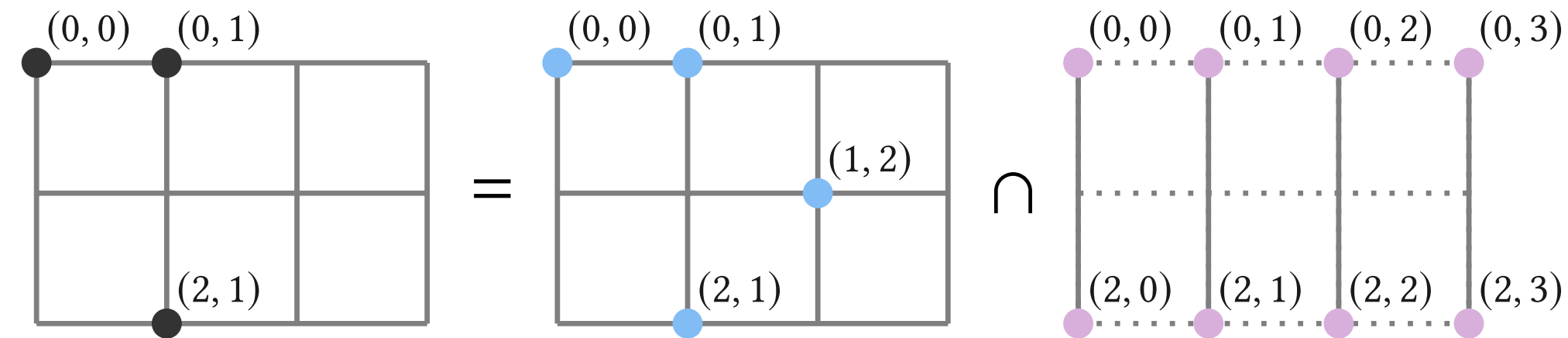
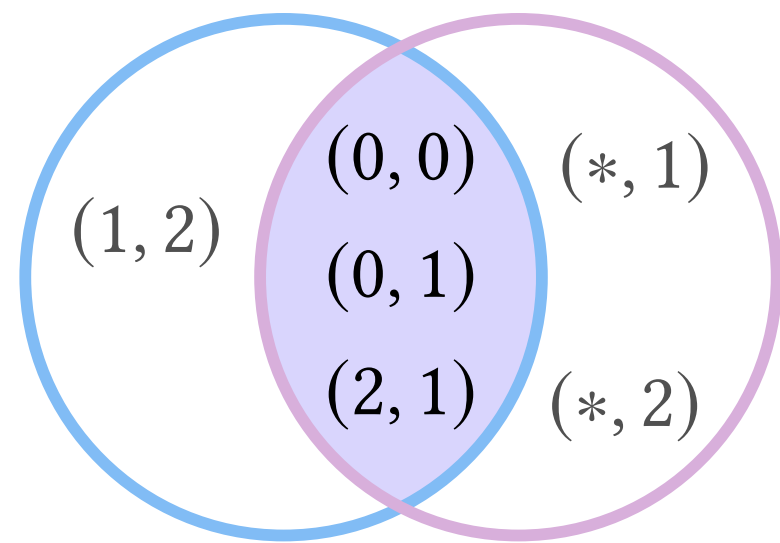


$$i \in B_1 \cap \mathbb{U}_i = B_1$$

$$k \in B_2 \cap C_1$$

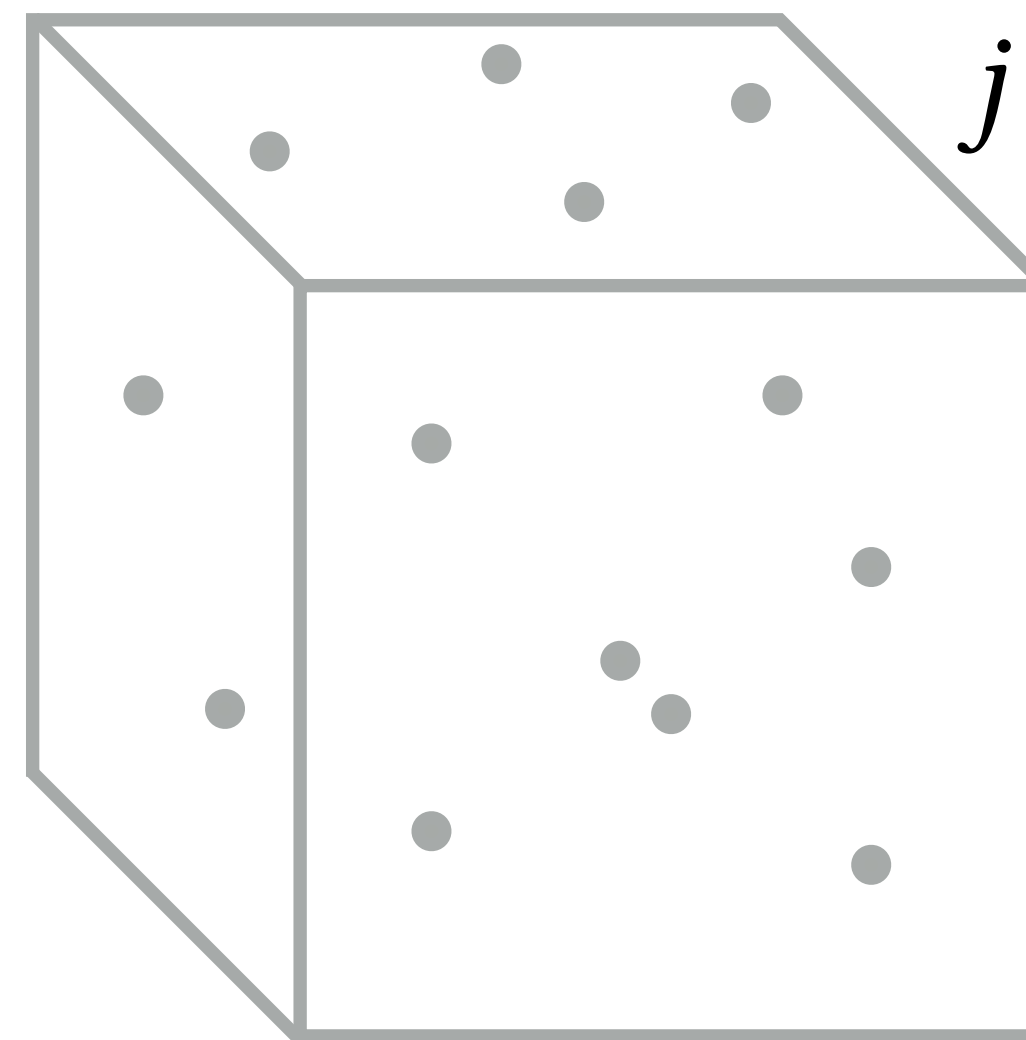
The universe of i consist of all coordinates, of which any data structure stores a subset.

Iteration spaces from broadcast operations



$$A_{ij} = \sum_k B_{ik} C_{kj}$$

$$B_{ik} \cap C_{kj}$$



$$i \in B_1 \cap \mathbb{U}_i = B_1$$

The universe of i consist of all coordinates, of which any data structure stores a subset.

Coordinate relations → coordinate trees (abstractly)

Matrix

	j_1	j_2	j_3
i_1	a	b	
i_2		e	
i_3	g	h	i

Coordinate relations → coordinate trees (abstractly)

Matrix

	j_1	j_2	j_3
i_1	a	b	
i_2		e	
i_3	g	h	i



Coordinate Relation

$$\begin{array}{ll} (i_1, j_1) \rightarrow a & (i_1, j_2) \rightarrow b \\ (i_3, j_3) \rightarrow i & (i_2, j_2) \rightarrow e \\ & (i_3, j_1) \rightarrow g \\ (i_3, j_2) \rightarrow h & \end{array}$$

Coordinate relations $\rightarrow$ coordinate trees (abstractly)

Matrix

	j_1	j_2	j_3
i_1	a	b	
i_2		e	
i_3	g	h	i

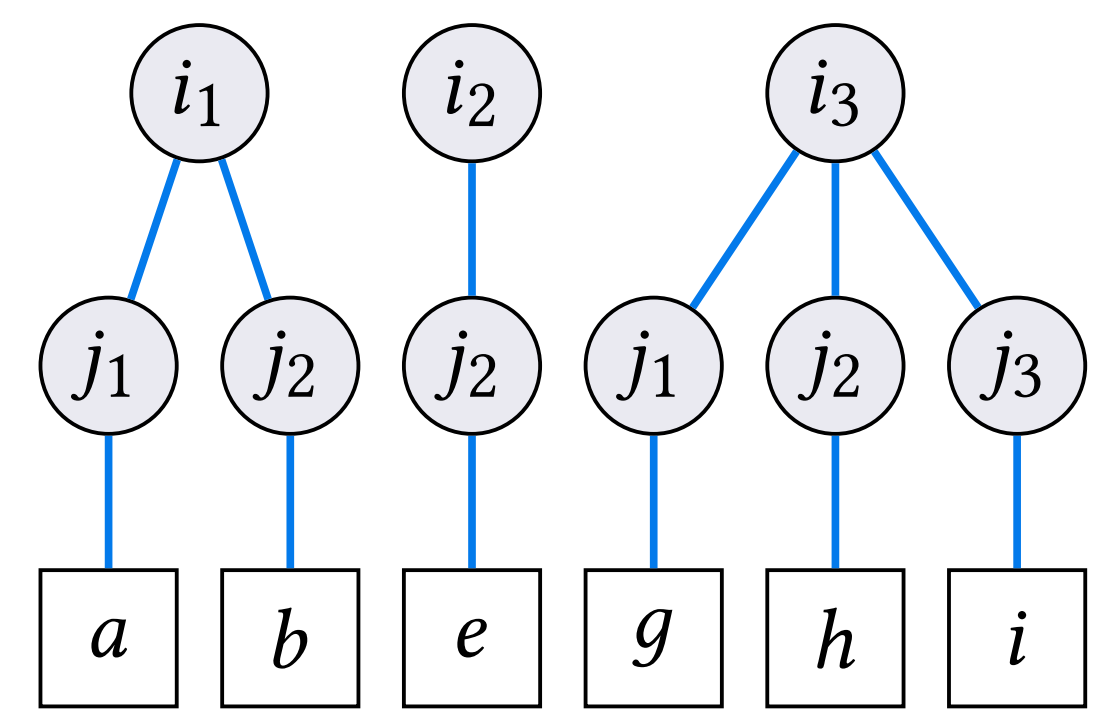


Coordinate Relation

$$\begin{array}{ll} (i_1, j_1) \rightarrow a & (i_1, j_2) \rightarrow b \\ (i_3, j_3) \rightarrow i & (i_2, j_2) \rightarrow e \\ & (i_3, j_1) \rightarrow g \\ (i_3, j_2) \rightarrow h & \end{array}$$



Coordinate Tree



Coordinate relations → coordinate trees (concretely)

	0	1	2	3
0	A		B	
1		C	D	E
2				F

A		B			C	D	E				F
0	1	2	3	4	5	6	7	8	9	10	11

Coordinate relations → coordinate trees (concretely)

	0	1	2	3
0	A		B	
1		C	D	E
2				F

A	B	C	D	E	F
0	1	2	3	4	5

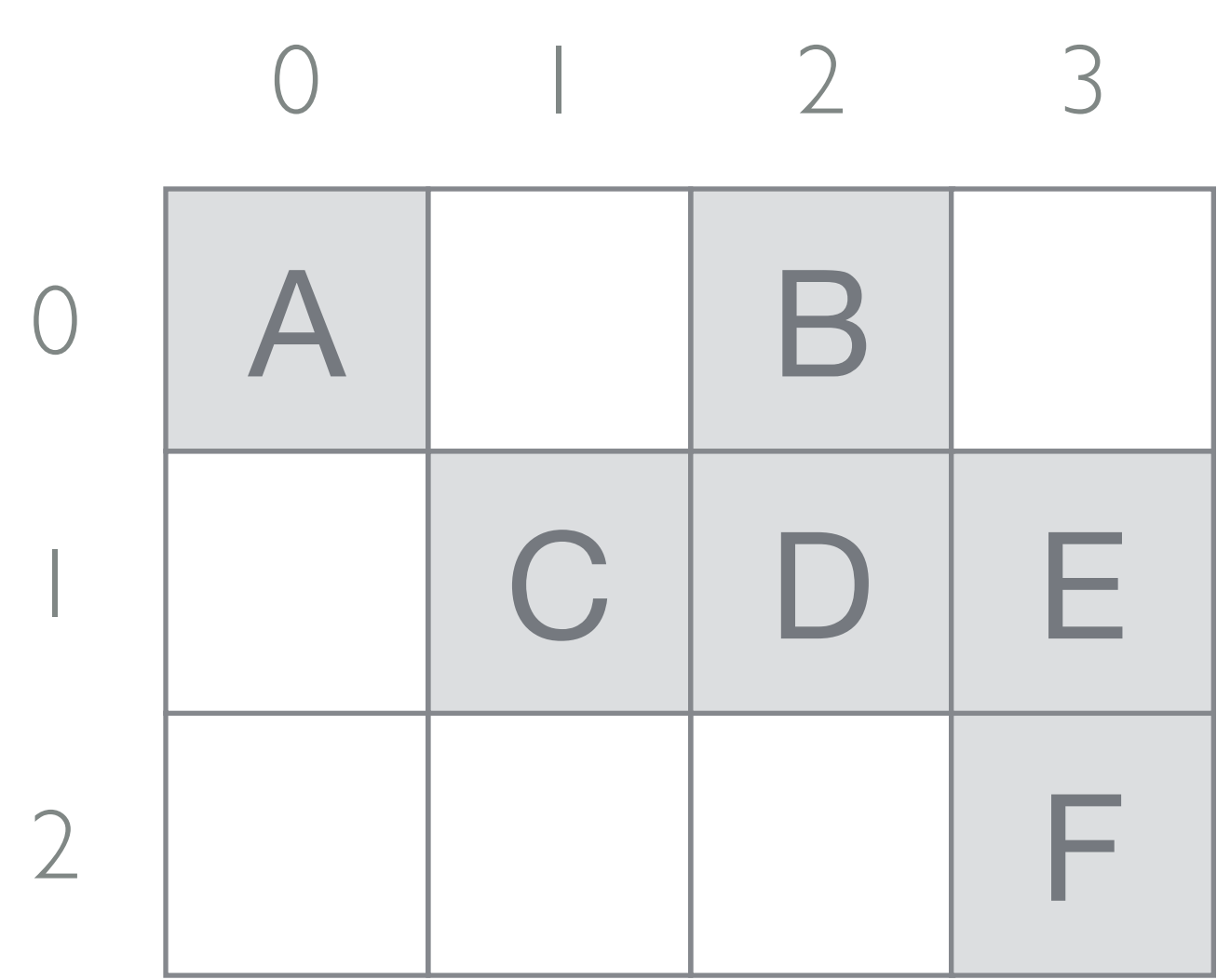
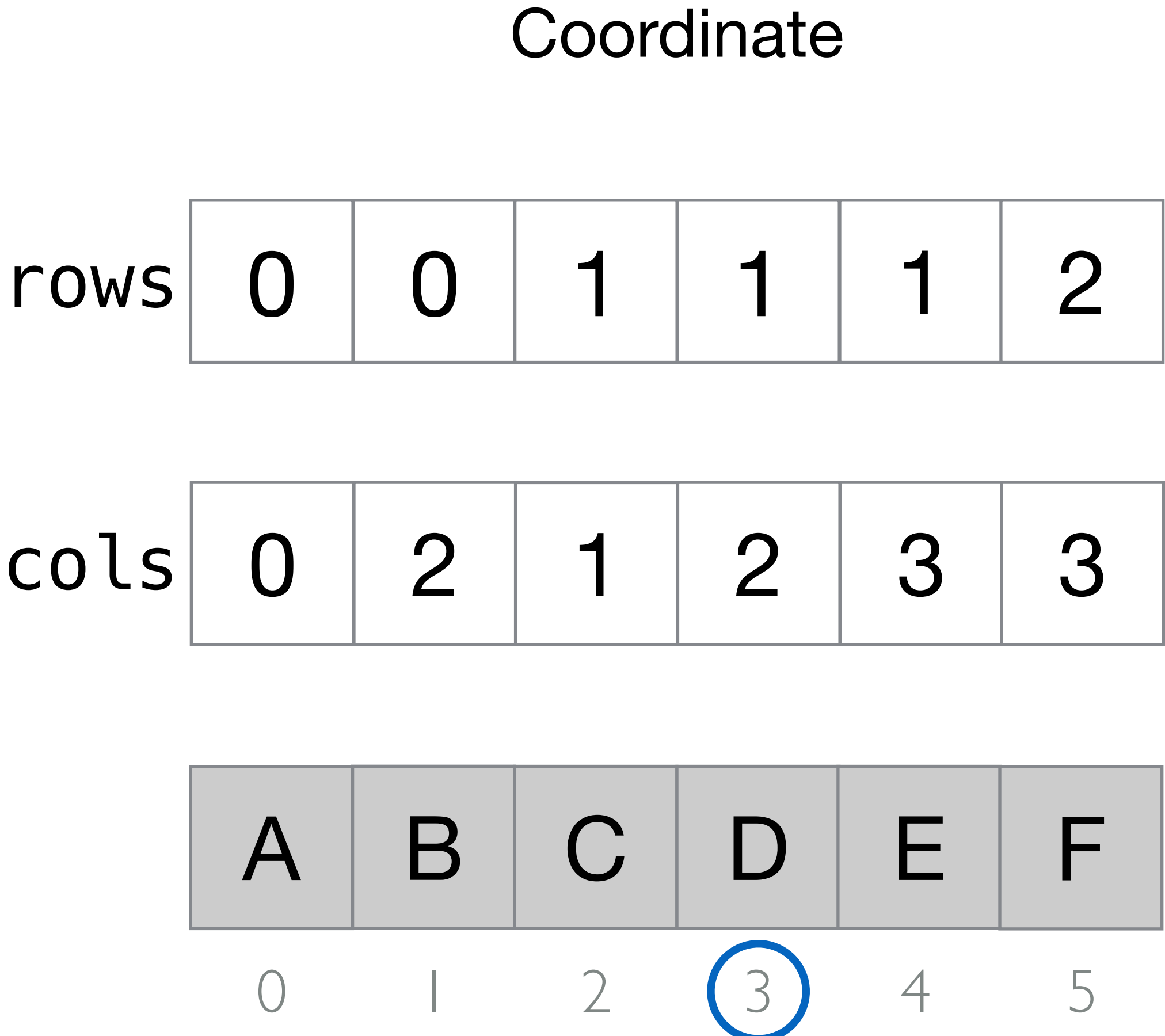
Coordinate relations → coordinate trees (concretely)

row(3) = ???
col(3) = ???

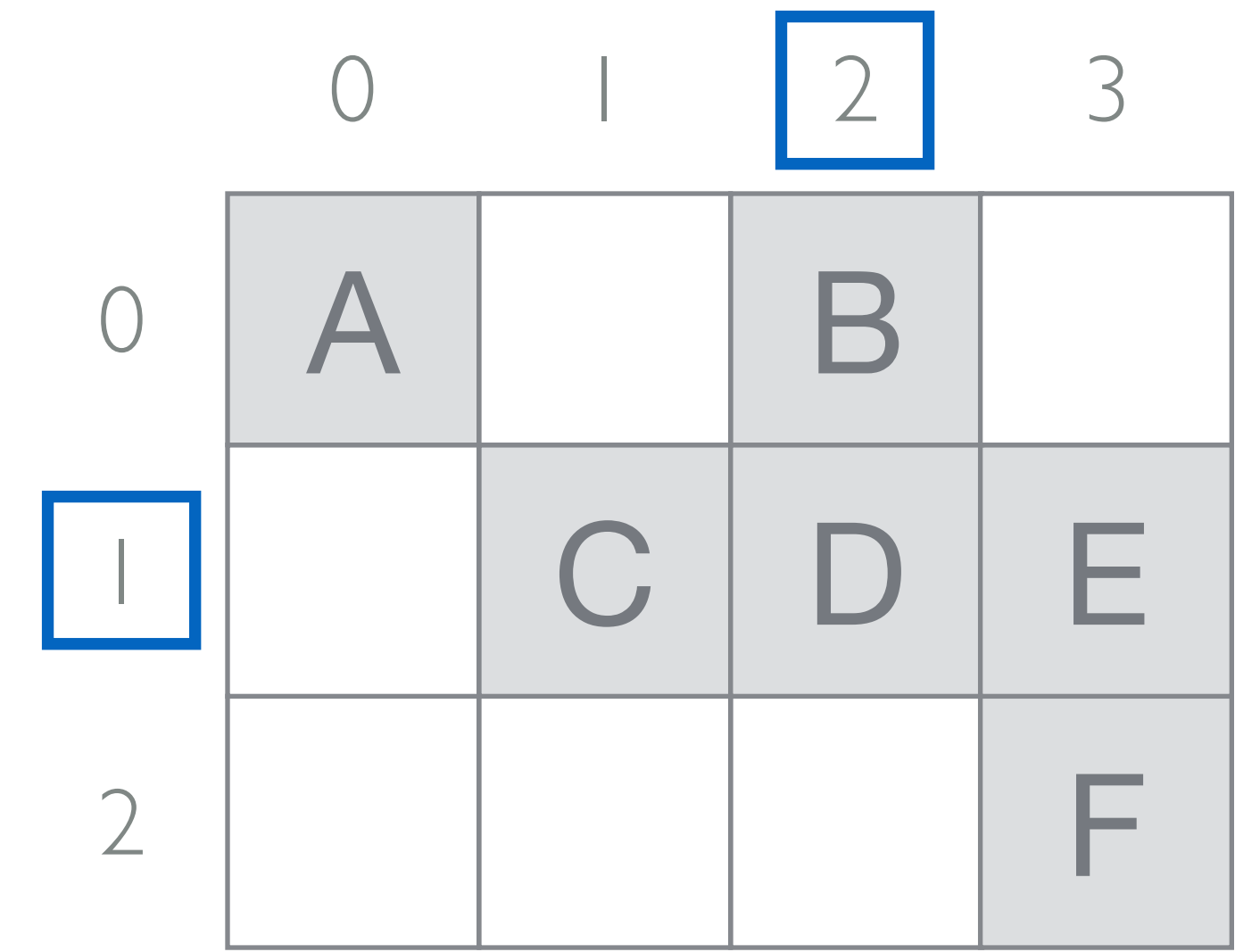
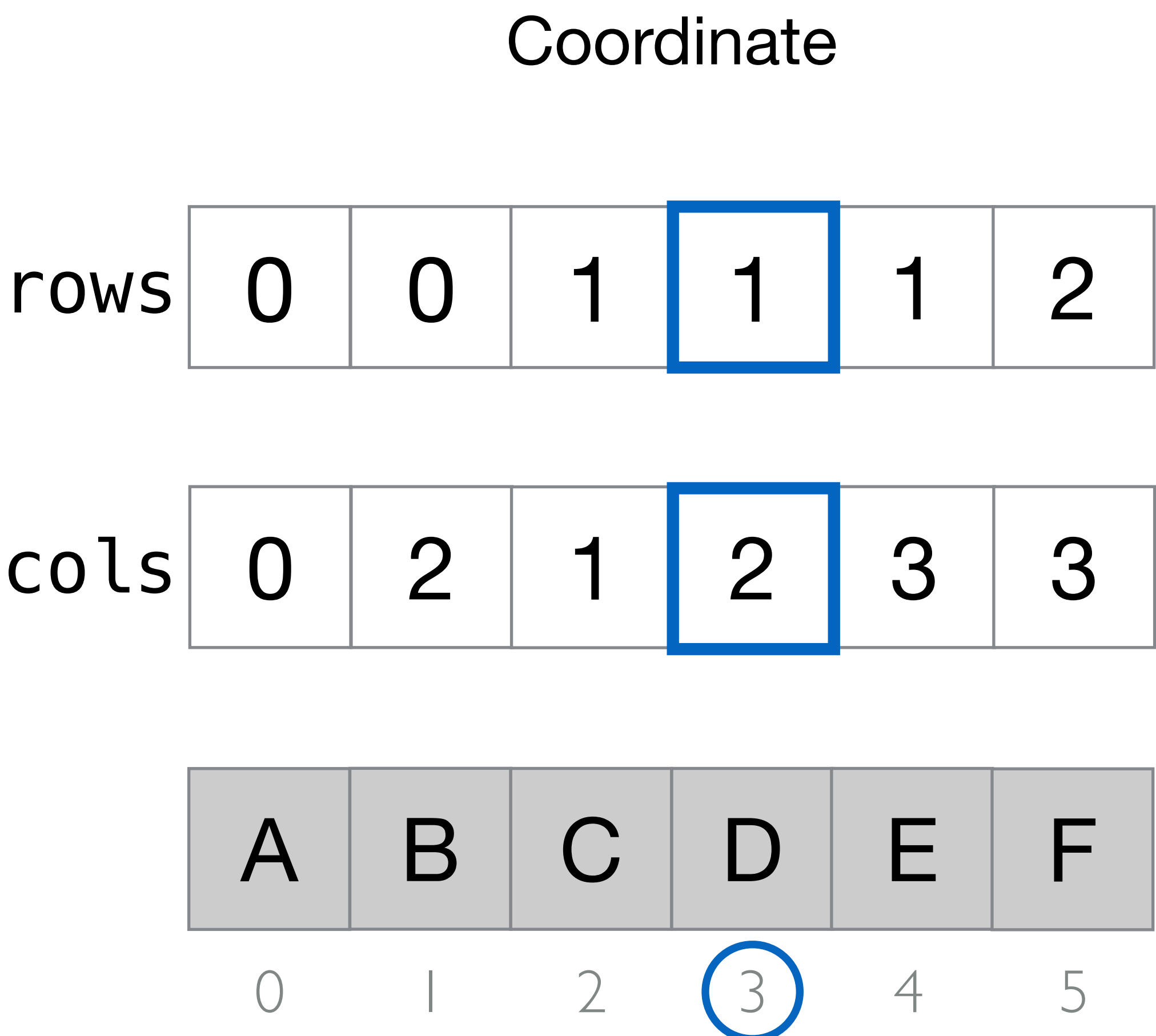
	0	1	2	3
0	A		B	
1		C	D	E
2				F

A	B	C	D	E	F
0	1	2	3	4	5

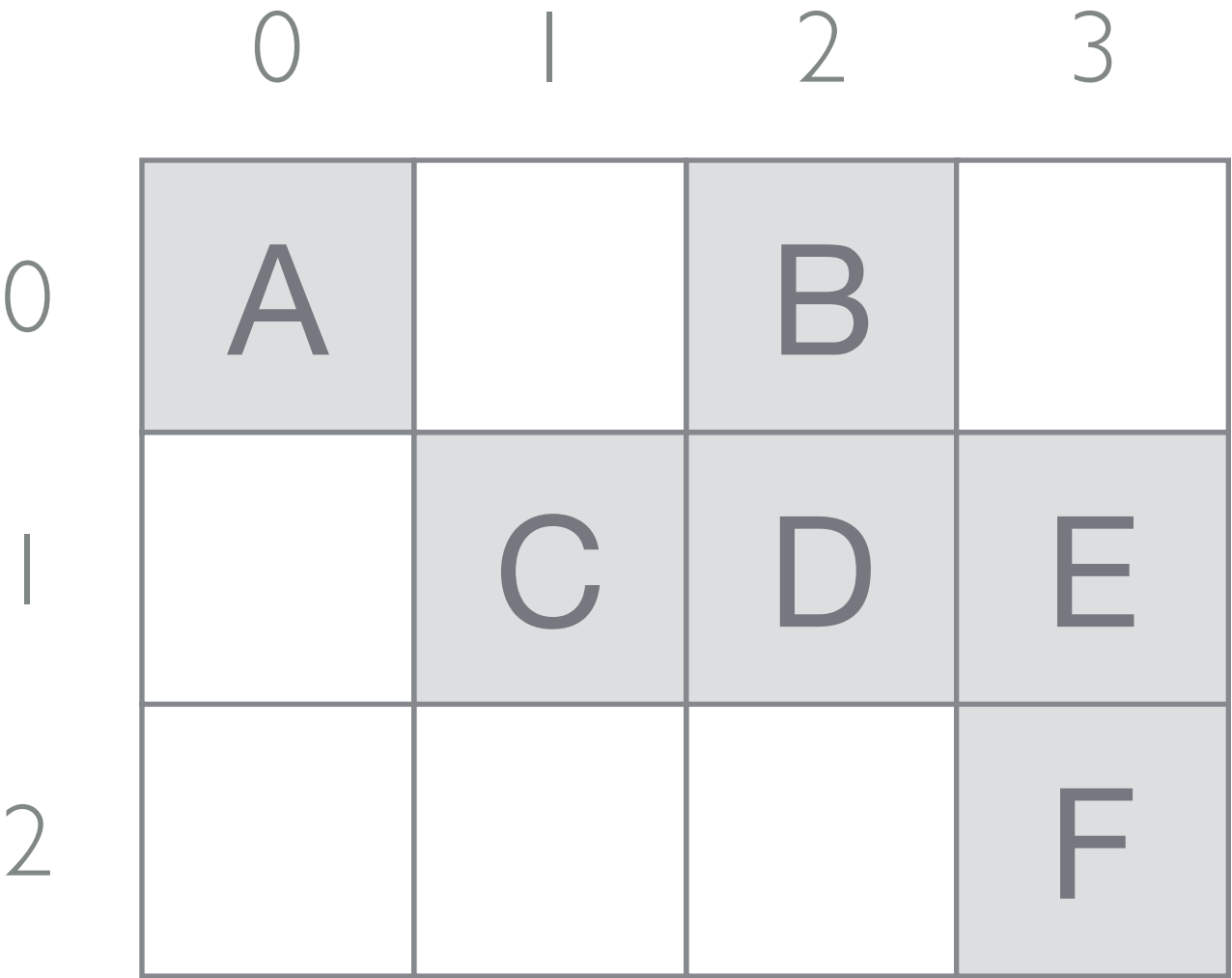
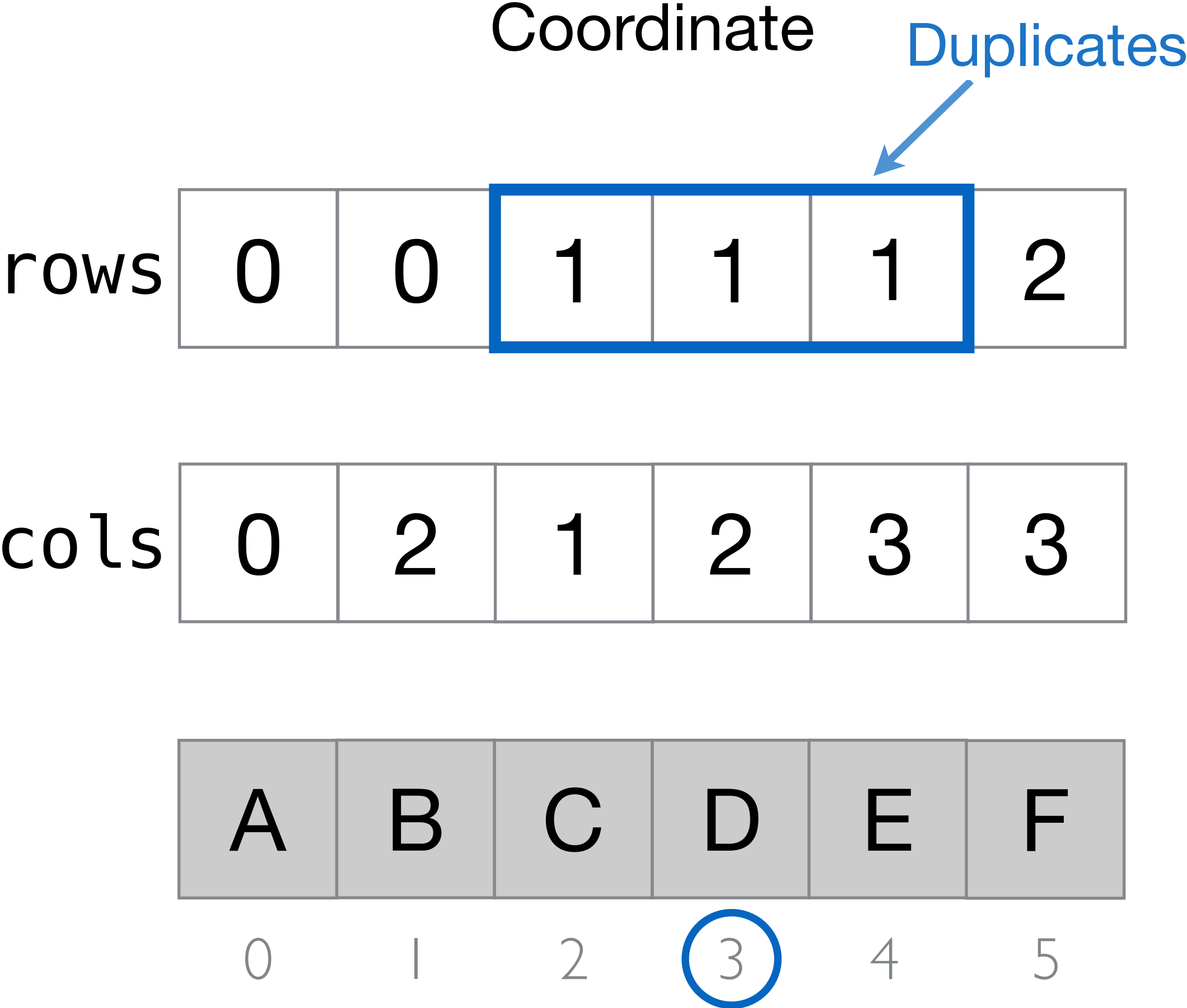
Coordinate relations → coordinate trees (concretely)



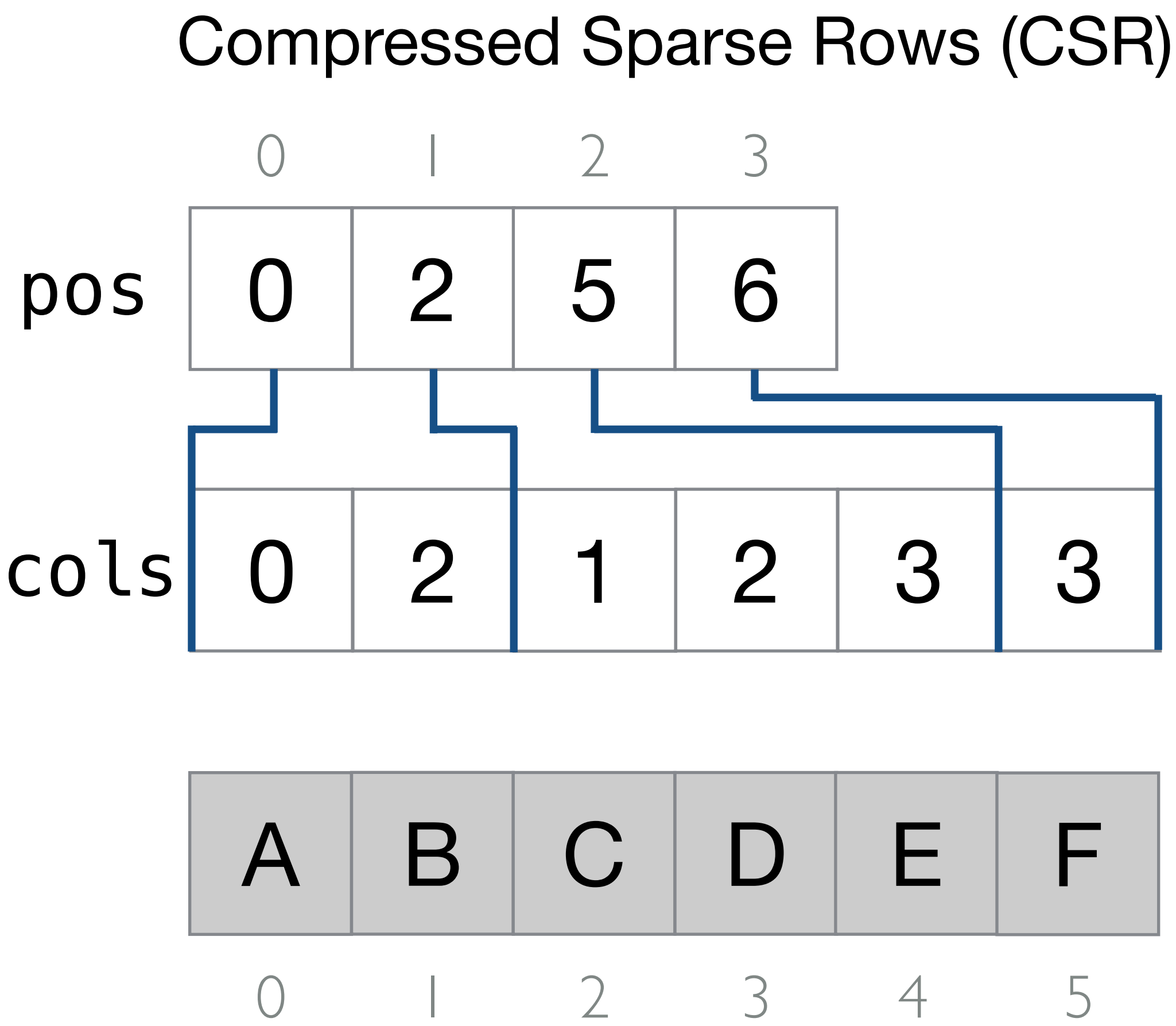
Coordinate relations → coordinate trees (concretely)



Coordinate relations → coordinate trees (concretely)

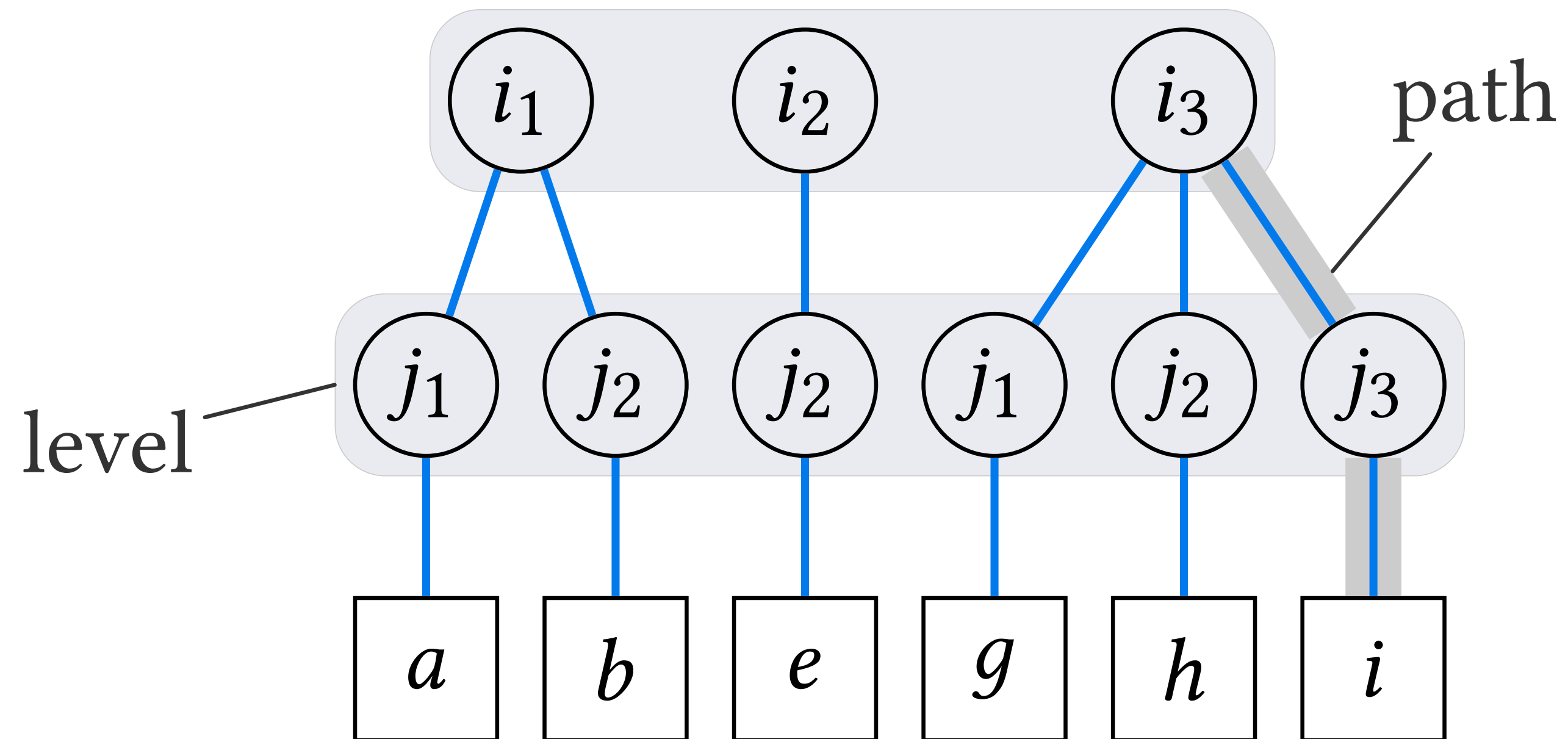


Coordinate relations → coordinate trees (concretely)



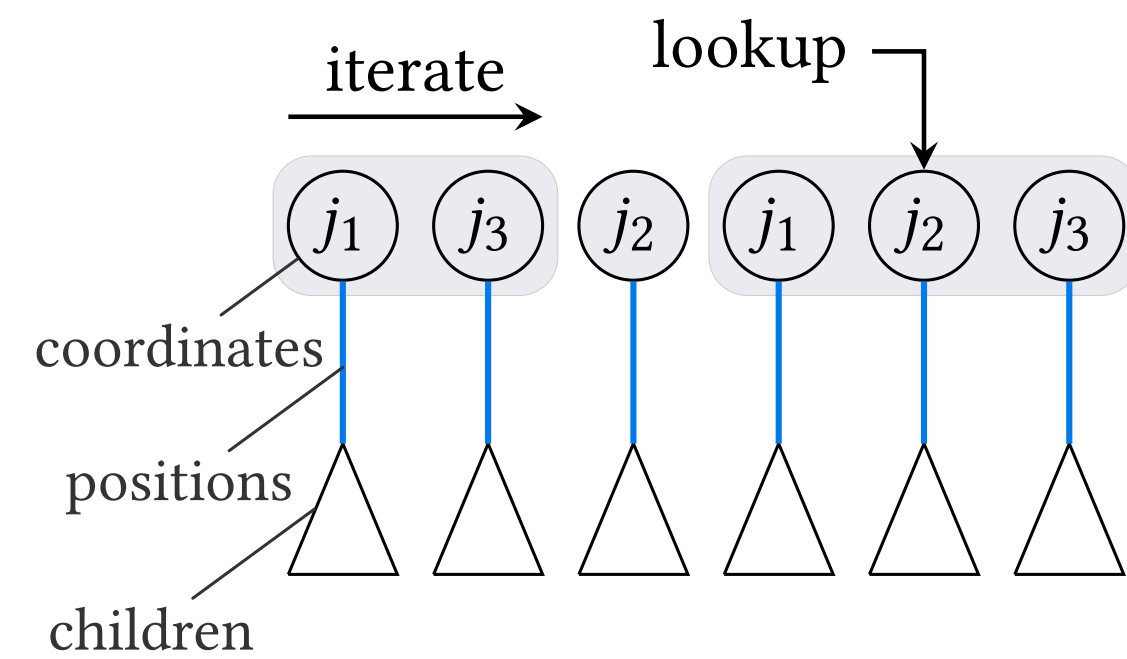
	0	1	2	3
0	A		B	
1		C	D	E
2				F

Level-based representation: compiler abstraction

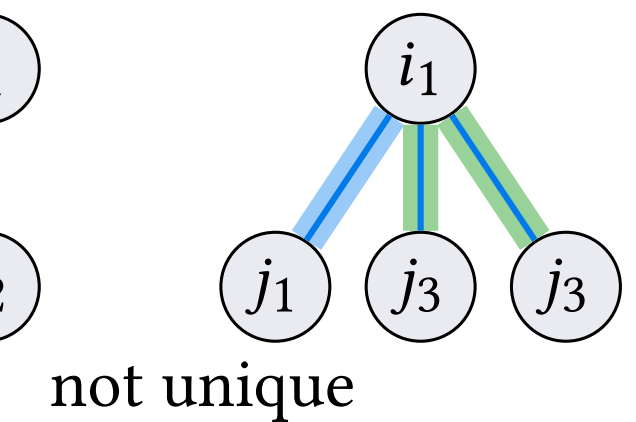
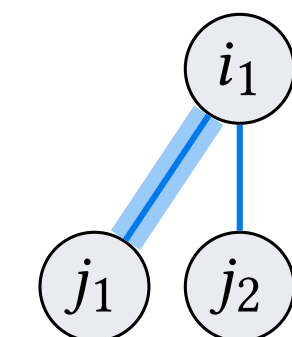
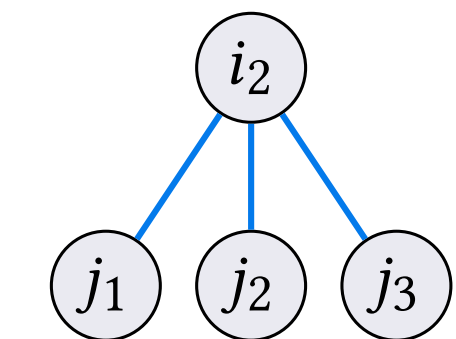
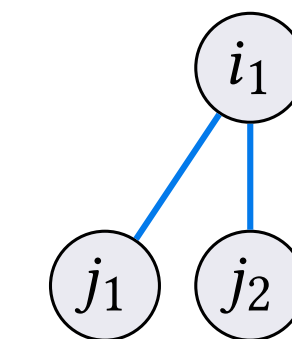
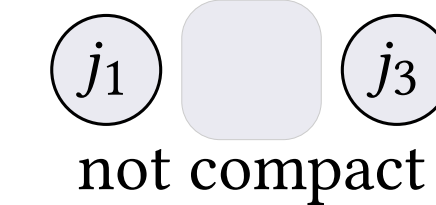
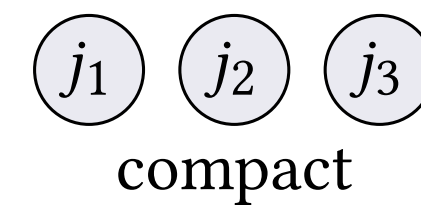
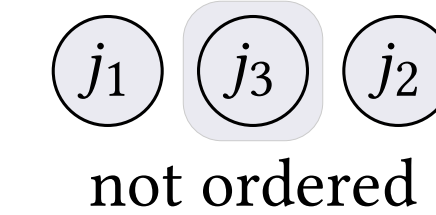
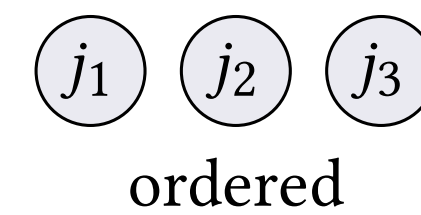
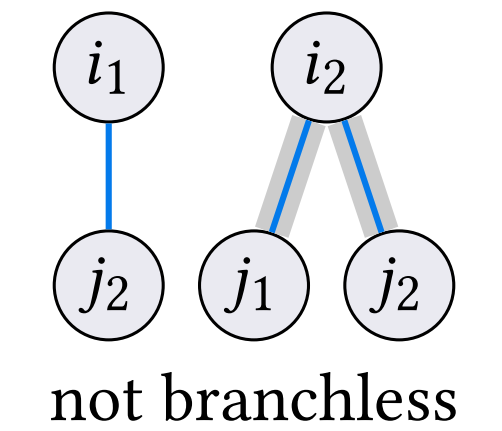
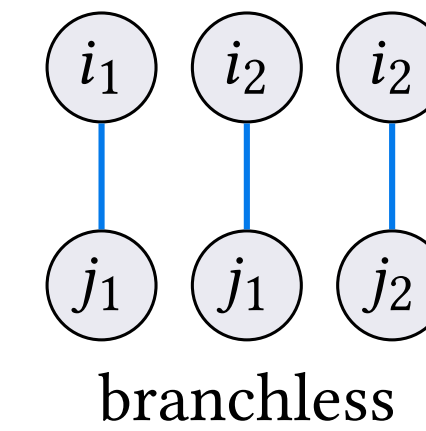
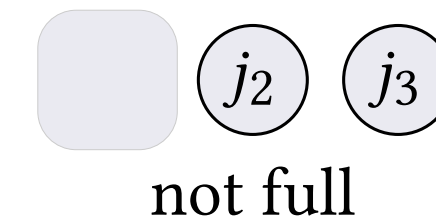
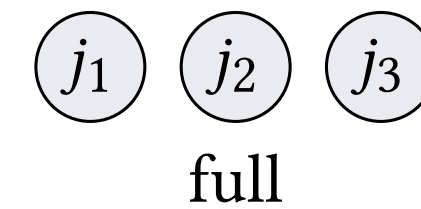


Level abstraction: capabilities and properties

Capabilities



Properties



The code generator sees only the level abstraction and not specific level types

Level types: dense and compressed

Dense locate capability:

```
locate( $p_{k-1}$ ,  $i_1$ , ...,  $i_k$ ):  
    return  $\langle p_{k-1} * N_k + i_k, \text{true} \rangle$ 
```


Level types: dense and compressed

Dense locate capability:

```
locate( $p_{k-1}$ ,  $i_1$ , ...,  $i_k$ ):  
    return  $\langle p_{k-1} * N_k + i_k, \text{true} \rangle$ 
```

Compressed iterate capability

```
pos_bounds( $p_{k-1}$ ):  
    return  $\langle \text{pos}[p_{k-1}], \text{pos}[p_{k-1} + 1] \rangle$ 
```

```
pos_access( $p_k$ ,  $i_1$ , ...,  $i_{k-1}$ ):  
    return  $\langle \text{crd}[p_k], \text{true} \rangle$ 
```

Level types: dense and compressed

Dense locate capability:

```
locate( $p_{k-1}$ ,  $i_1$ , ...,  $i_k$ ):  
    return  $\langle p_{k-1} * N_k + i_k, \text{true} \rangle$ 
```

Compressed iterate capability

```
pos_bounds( $p_{k-1}$ ):  
    return  $\langle \text{pos}[p_{k-1}], \text{pos}[p_{k-1} + 1] \rangle$ 
```

```
pos_access( $p_k$ ,  $i_1$ , ...,  $i_{k-1}$ ):  
    return  $\langle \text{crd}[p_k], \text{true} \rangle$ 
```

$$y = Ax$$

```
for (int i = 0; i < m; i++) {  
    for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {  
        int j = A_crd[pA];  
        y[i] += A[pA] * x[j];  
    }  
}
```

Level types: dense and compressed

Dense locate capability:

```
locate( $p_{k-1}$ ,  $i_1$ , ...,  $i_k$ ):  
    return  $\langle p_{k-1} * N_k + i_k, \text{true} \rangle$ 
```

Compressed iterate capability

```
pos_bounds( $p_{k-1}$ ):  
    return  $\langle \text{pos}[p_{k-1}], \text{pos}[p_{k-1} + 1] \rangle$ 
```

```
pos_access( $p_k$ ,  $i_1$ , ...,  $i_{k-1}$ ):  
    return  $\langle \text{crd}[p_k], \text{true} \rangle$ 
```

$$y = Ax$$

```
for (int i = 0; i < m; i++) {  
    for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {  
        int j = A_crd[pA];  
        y[i] += A[pA] * x[j];  
    }  
}
```

Compressed iterate



Level types: dense and compressed

Dense locate capability:

```
locate( $p_{k-1}$ ,  $i_1$ , ...,  $i_k$ ):  
    return  $\langle p_{k-1} * N_k + i_k, \text{true} \rangle$ 
```

Compressed iterate capability

```
pos_bounds( $p_{k-1}$ ):  
    return  $\langle \text{pos}[p_{k-1}], \text{pos}[p_{k-1} + 1] \rangle$ 
```

```
pos_access( $p_k$ ,  $i_1$ , ...,  $i_{k-1}$ ):  
    return  $\langle \text{crd}[p_k], \text{true} \rangle$ 
```

$$y = Ax$$

```
for (int i = 0; i < m; i++) {  
    for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {  
        int j = A_crd[pA];  
        y[i] += A[pA] * x[j];  
    }  
}
```

Compressed iterate

Level types: dense and compressed

Dense locate capability:

```
locate( $p_{k-1}$ ,  $i_1$ , ...,  $i_k$ ):  
    return  $\langle p_{k-1} * N_k + i_k, \text{true} \rangle$ 
```

Compressed iterate capability

```
pos_bounds( $p_{k-1}$ ):  
    return  $\langle \text{pos}[p_{k-1}], \text{pos}[p_{k-1} + 1] \rangle$ 
```

```
pos_access( $p_k$ ,  $i_1$ , ...,  $i_{k-1}$ ):  
    return  $\langle \text{crd}[p_k], \text{true} \rangle$ 
```

$$y = Ax$$

```
for (int i = 0; i < m; i++) {  
    for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {  
        int j = A_crd[pA];  
        y[i] += A[pA] * x[j];  
    }  
}
```

Compressed iterate

Level types: dense and compressed

Dense locate capability:

```
locate( $p_{k-1}$ ,  $i_1$ , ...,  $i_k$ ):  
    return  $\langle p_{k-1} * N_k + i_k, \text{true} \rangle$ 
```

Compressed iterate capability

```
pos_bounds( $p_{k-1}$ ):  
    return  $\langle \text{pos}[p_{k-1}], \text{pos}[p_{k-1} + 1] \rangle$ 
```

```
pos_access( $p_k$ ,  $i_1$ , ...,  $i_{k-1}$ ):  
    return  $\langle \text{crd}[p_k], \text{true} \rangle$ 
```

$$y = Ax$$

```
for (int i = 0; i < m; i++) {  
    for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {  
        int j = A_crd[pA];  
        y[i] += A[pA] * x[j];  
    }  
}
```

Dense locate

Compressed iterate

Level types: dense and compressed

Dense locate capability:

```
locate( $p_{k-1}$ ,  $i_1$ , ...,  $i_k$ ):  
    return < $p_{k-1} * N_k + i_k$ , true>
```

Compressed iterate capability

```
pos_bounds( $p_{k-1}$ ):  
    return <pos[ $p_{k-1}$ ], pos[ $p_{k-1} + 1$ ]>
```

```
pos_access( $p_k$ ,  $i_1$ , ...,  $i_{k-1}$ ):  
    return <crd[ $p_k$ ], true>
```

$$y = Ax$$

```
for (int i = 0; i < m; i++) {  
    for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {  
        int j = A_crd[pA];  
        y[i] += A[pA] * x[j];  
    }  
}
```

Dense locate

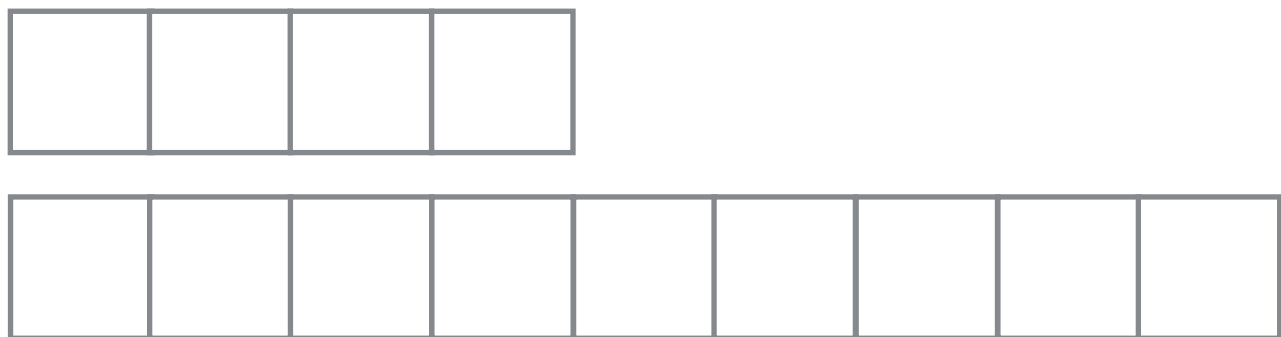
Compressed iterate

Level types can be composed in many ways

Dense



Compressed



Singleton

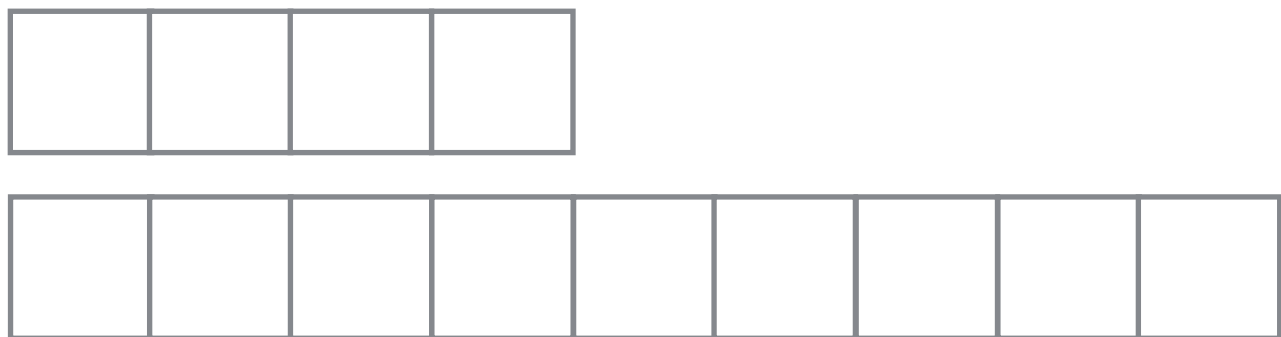


Level types can be composed in many ways

Dense



Compressed



Singleton



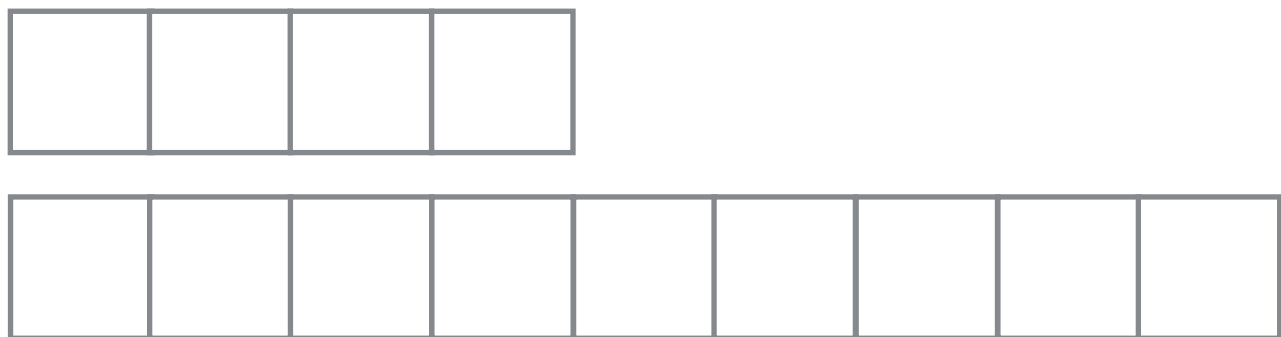
	0	1	2	3
0	A		B	
1		C	D	E
2				F

Level types can be composed in many ways

Dense



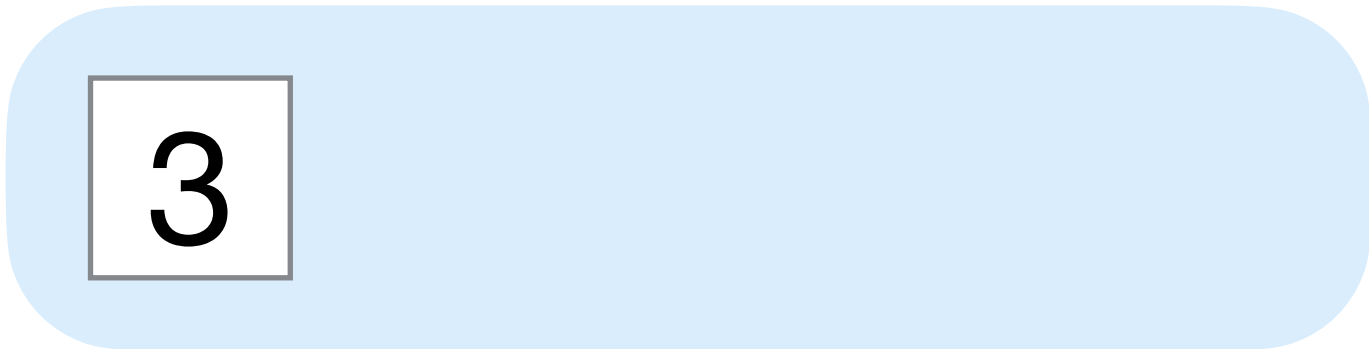
Compressed



Singleton



Dense



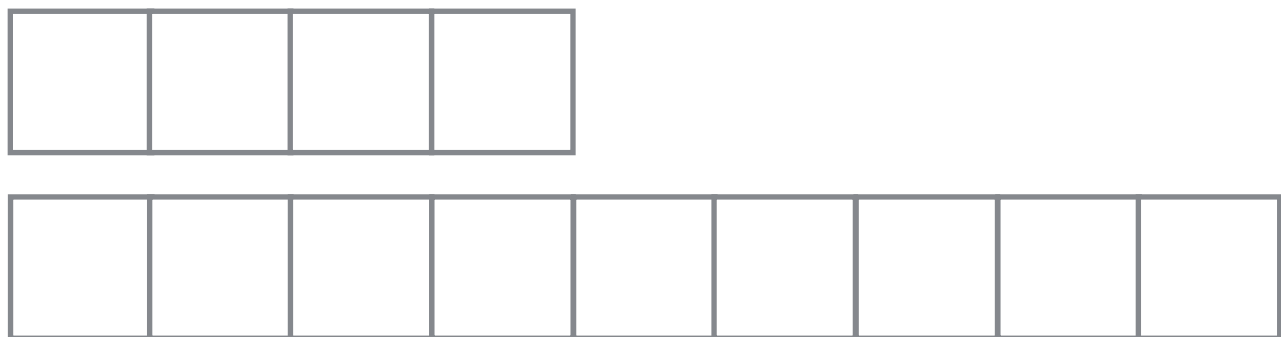
	0	1	2	3
0	A		B	
1		C	D	E
2				F

Level types can be composed in many ways

Dense



Compressed



Singleton



Dense



Compressed



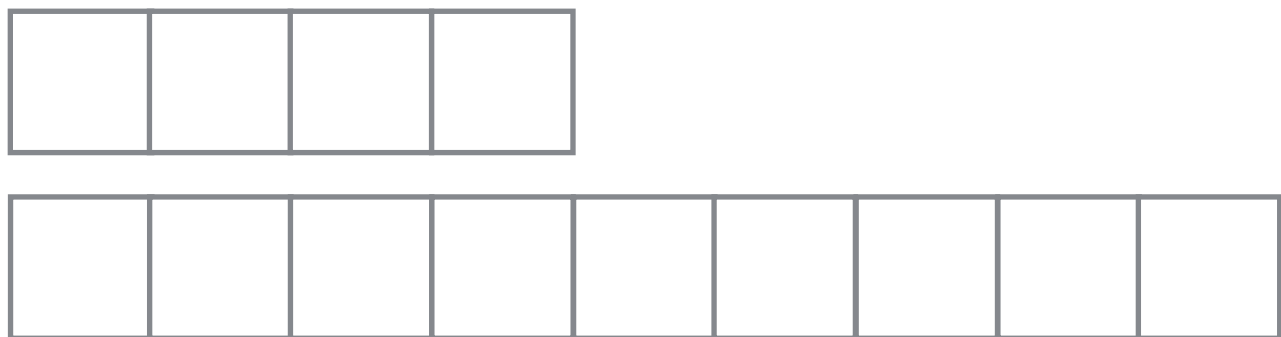
	0	1	2	3
0	A		B	
1		C	D	E
2				F

Level types can be composed in many ways

Dense



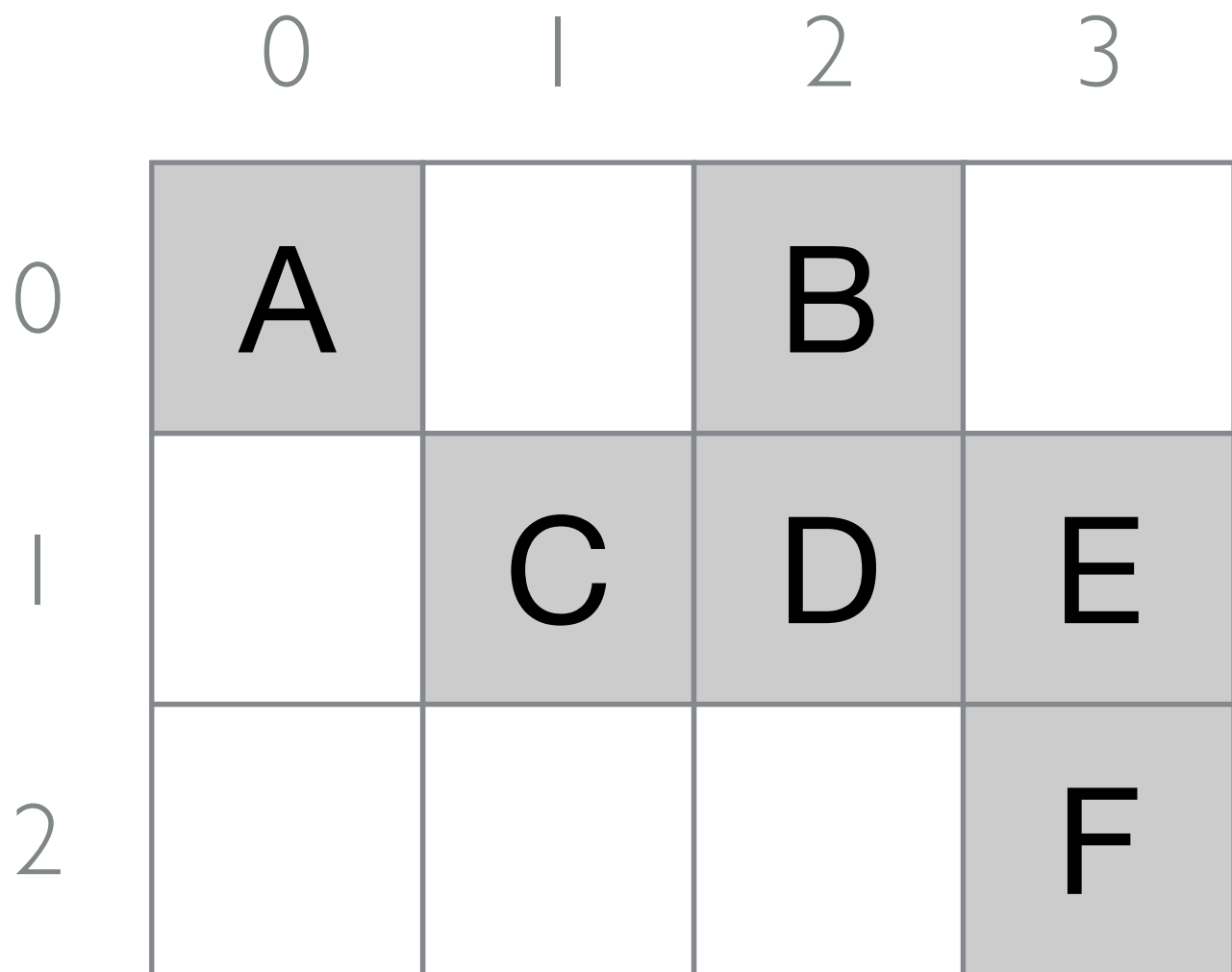
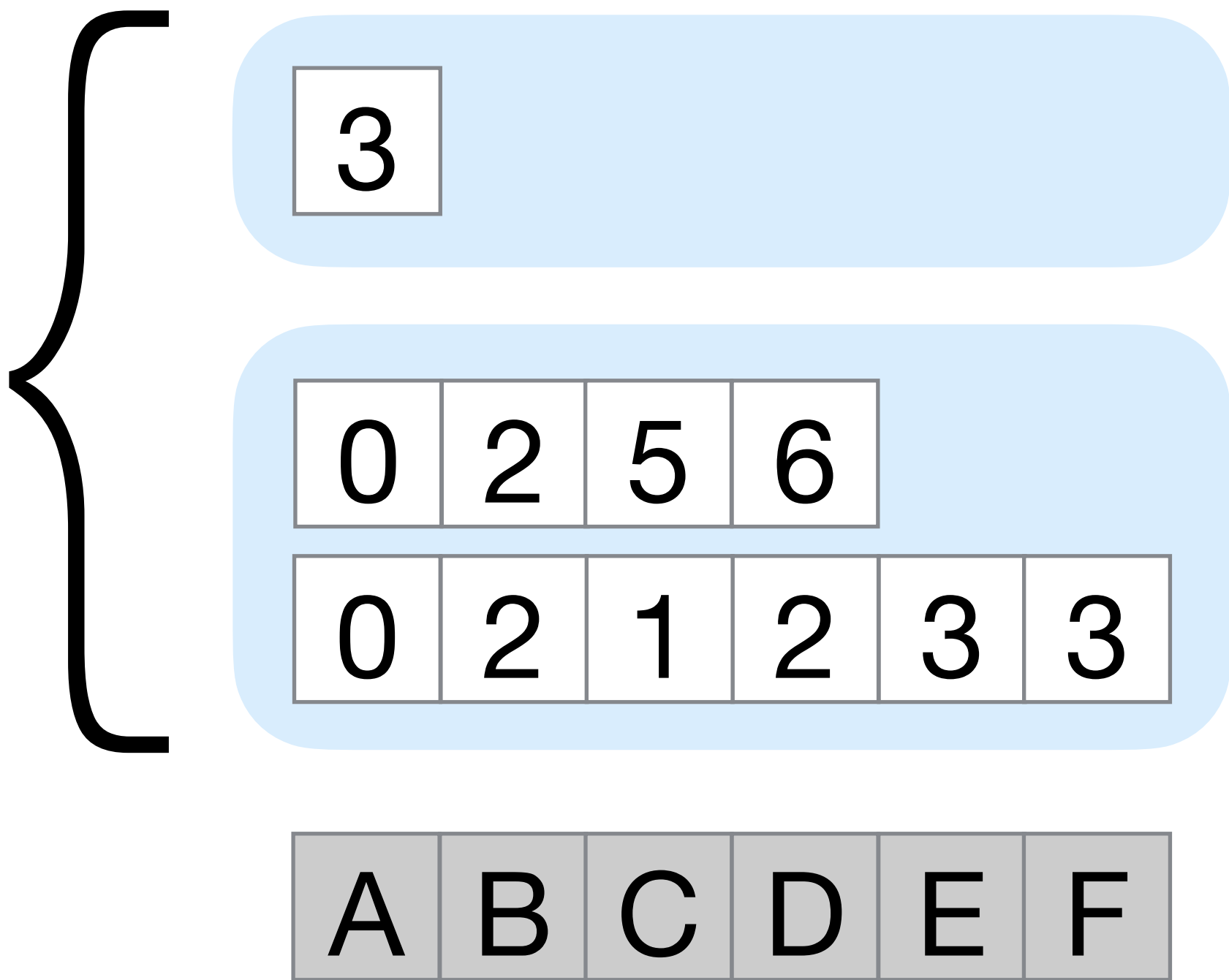
Compressed



Singleton



CSR

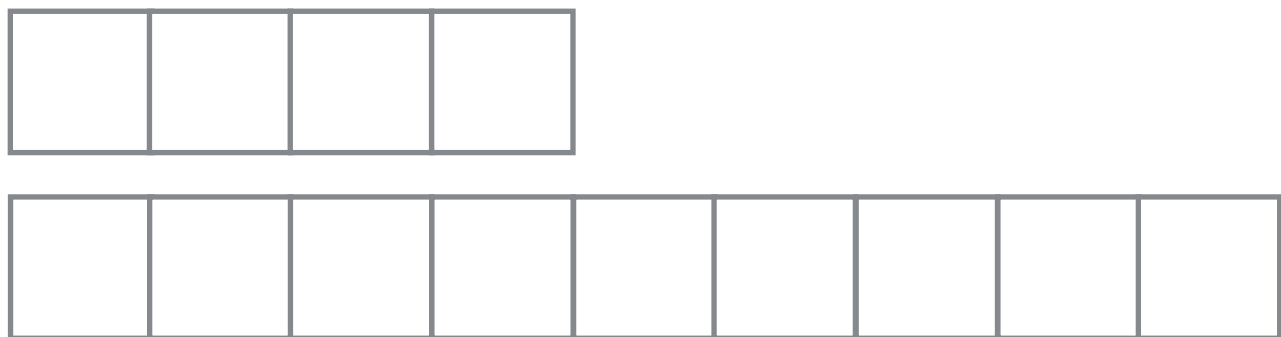


Level types can be composed in many ways

Dense



Compressed



Singleton



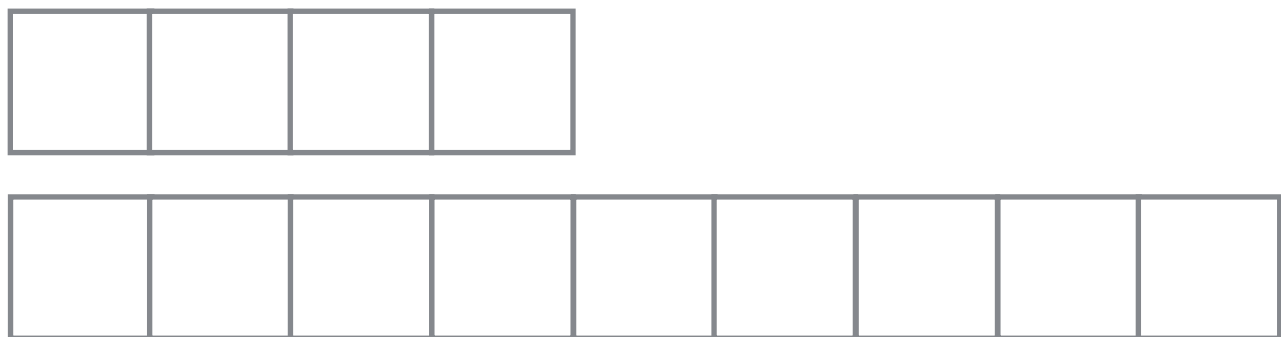
	0	1	2	3
0	A		B	
1		C	D	E
2				F

Level types can be composed in many ways

Dense



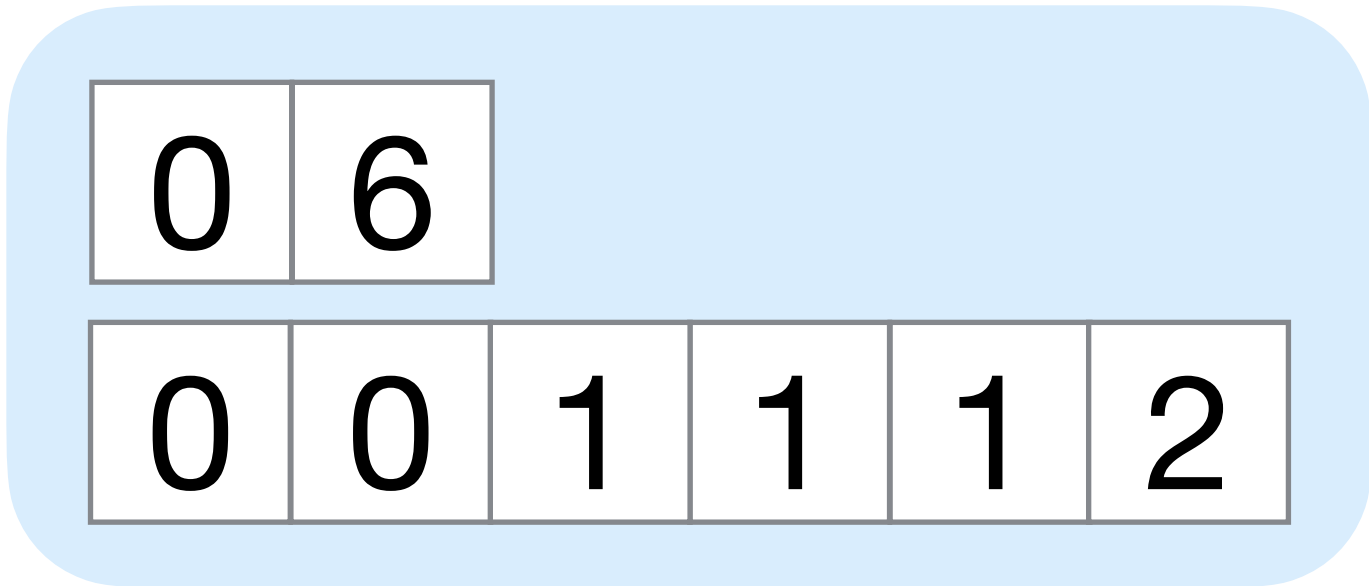
Compressed



Singleton



Compressed



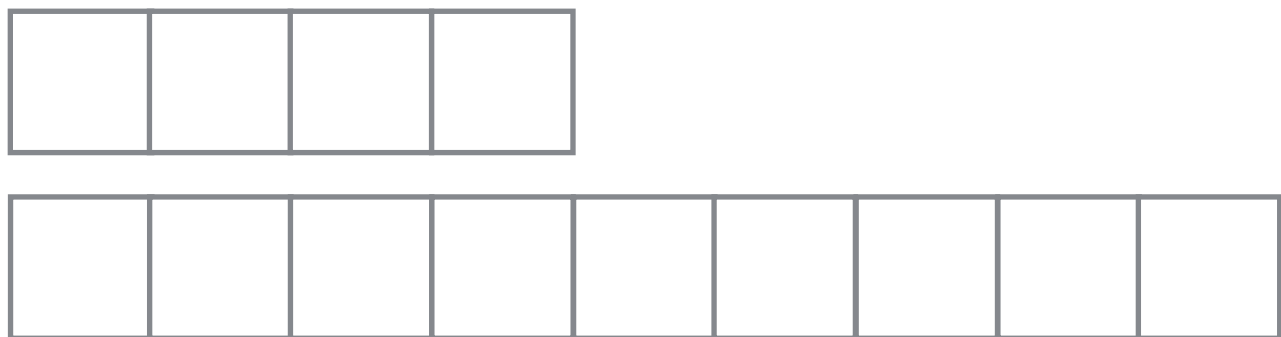
	0	1	2	3
0	A		B	
1		C	D	E
2				F

Level types can be composed in many ways

Dense



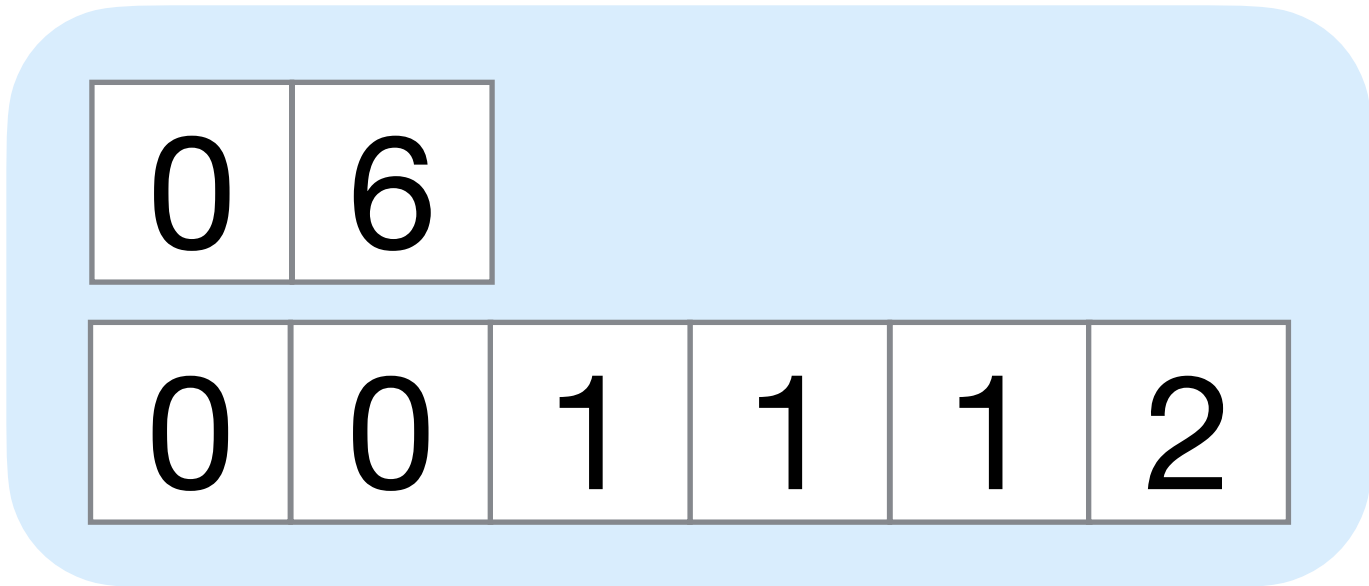
Compressed



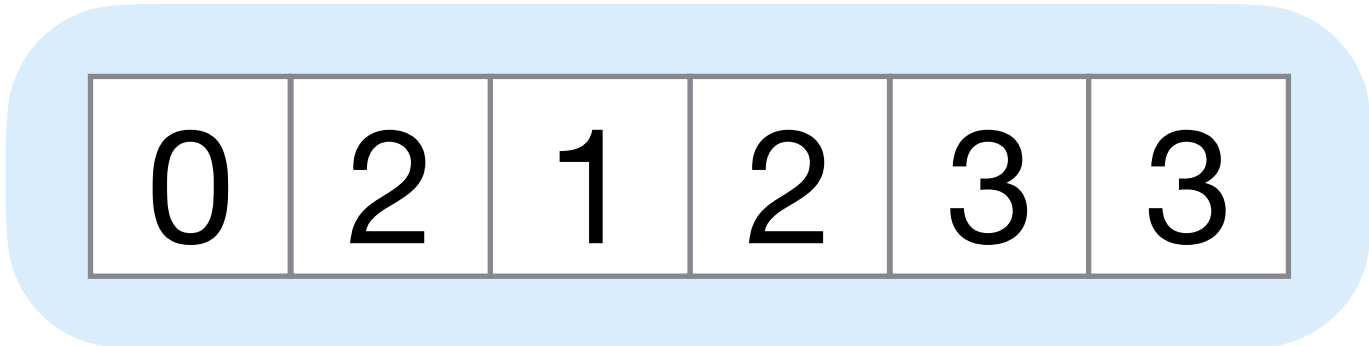
Singleton



Compressed



Singleton



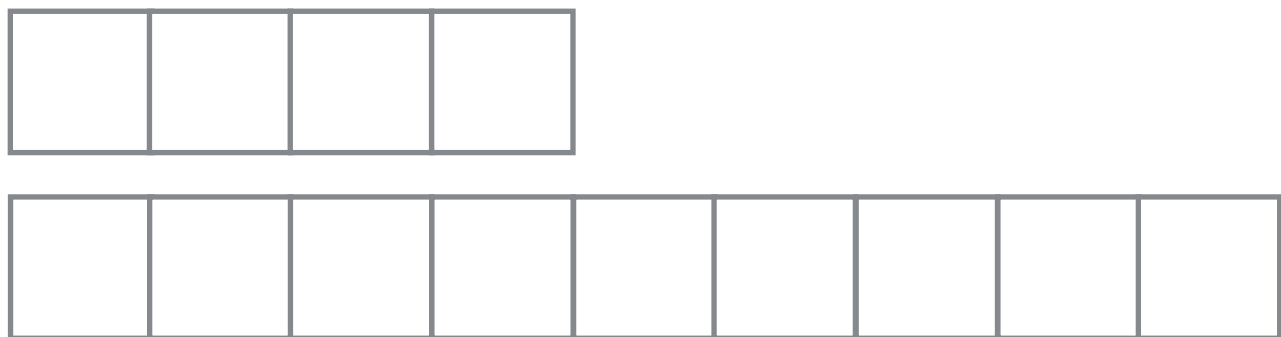
	0	1	2	3
0	A		B	
1		C	D	E
2				F

Level types can be composed in many ways

Dense



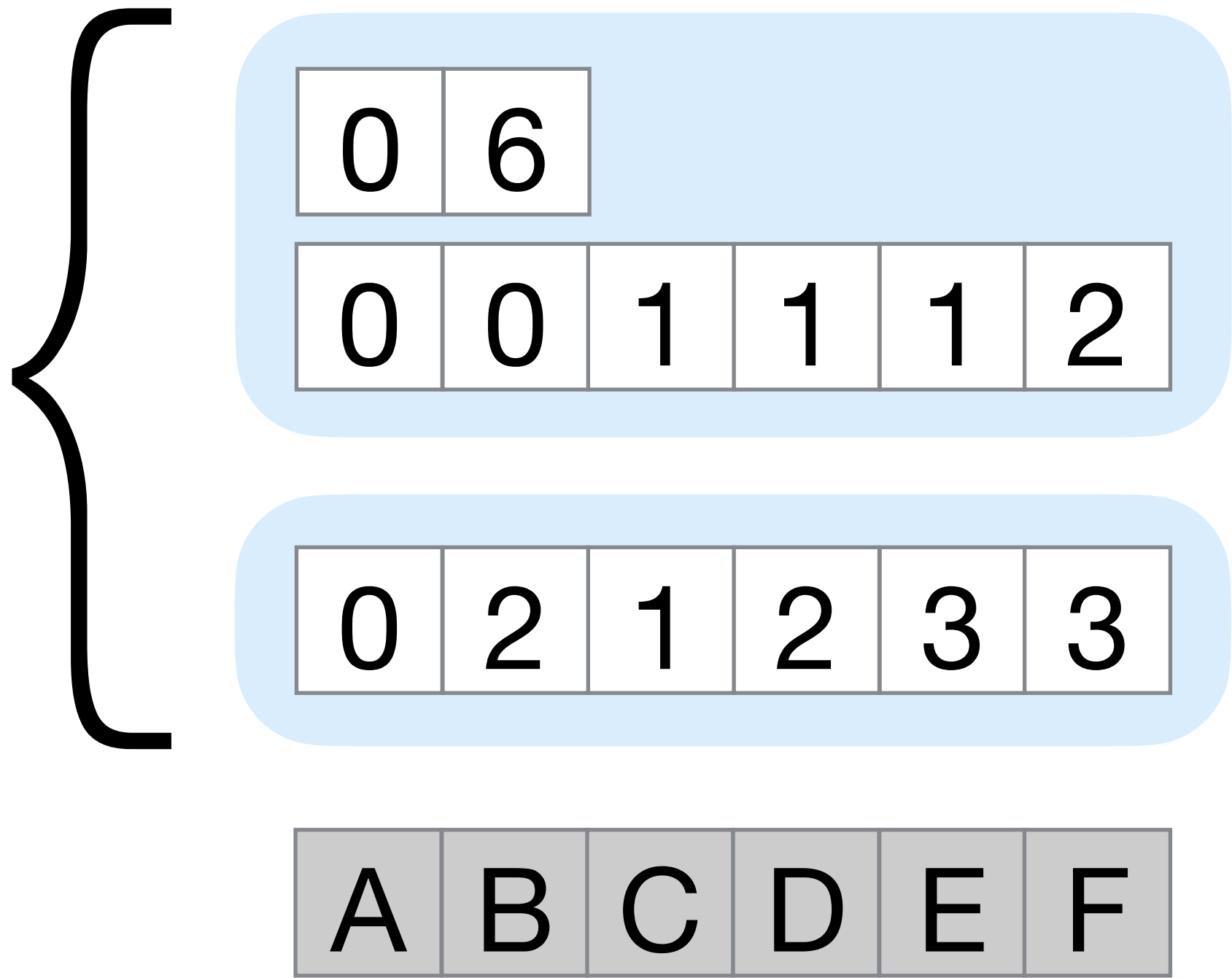
Compressed



Singleton



Coordinates



	0	1	2	3
0	A		B	
1		C	D	E
2				F

Level types can be composed in many ways

Dense

Compressed

Singleton

Level types can be composed in many ways

Level
formats

Dense Compressed Singleton

Coordinate matrix
Compressed
Singleton

CSR
Dense
Compressed

[Tinney and Walker, 1967]

Tensor
formats

Level types can be composed in many ways

Level
formats

Dense Compressed Singleton

Coordinate matrix
Compressed
Singleton

CSR
Dense
Compressed

[Tinney and Walker, 1967]

Dense array tensor
Dense
Dense
Dense

Coordinate tensor
Compressed
Singleton
Singleton

Tensor
formats

Mode-generic tensor
Compressed
Singleton
Dense
Dense

[Baskaran et al. 2012]

Level types can be composed in many ways

Level formats Dense Compressed Singleton

Tensor formats

Coordinate matrix
Compressed
Singleton

[Baskaran et al. 2012]

CSR
Dense
Compressed

[Tinney and Walker, 1967]

Dense array tensor
Dense
Dense
Dense

[Buluç et al. 2009]

Coordinate tensor
Compressed
Singleton
Singleton

Mode-generic tensor
Compressed
Singleton
Dense
Dense

BCSR
Dense
Compressed
Dense
Dense

[Im and Yelick 1998]

CSB
Dense
Dense
Compressed
Singleton

ELLPACK
Dense
Dense
Singleton

[Kincaid et al. 1989]

Level types can be composed in many ways

Level formats	Dense Hashed	Compressed Range	Singleton Offset
---------------	--------------	------------------	------------------

Tensor formats

Coordinate matrix
Compressed
Singleton

[Baskaran et al. 2012]

CSR
Dense
Compressed

[Tinney and Walker, 1967]

Dense array tensor
Dense
Dense
Dense

[Buluç et al. 2009]

Coordinate tensor
Compressed
Singleton
Singleton

Mode-generic tensor
Compressed
Singleton
Dense
Dense

BCSR
Dense
Compressed
Dense
Dense

[Im and Yelick 1998]

CSB
Dense
Dense
Compressed
Singleton

ELLPACK
Dense
Dense
Singleton

[Kincaid et al. 1989]

Level types can be composed in many ways

Level formats	Dense Hashed	Compressed Range	Singleton Offset
---------------	--------------	------------------	------------------

Tensor formats

Coordinate matrix
Compressed
Singleton

[Baskaran et al. 2012]

Mode-generic tensor
Compressed
Singleton
Dense
Dense

Hash map vector
Hashed

[Patwary et al. 2015]

CSR
Dense
Compressed

[Tinney and Walker, 1967]

BCSR
Dense
Compressed
Dense
Dense

[Im and Yelick 1998]

Hash map matrix
Hashed
Hashed

Dense array tensor
Dense
Dense
Dense

CSB
Dense
Dense
Compressed
Singleton

[Buluç et al. 2009]

DIA
Dense
Range
Offset

[Saad 2003]

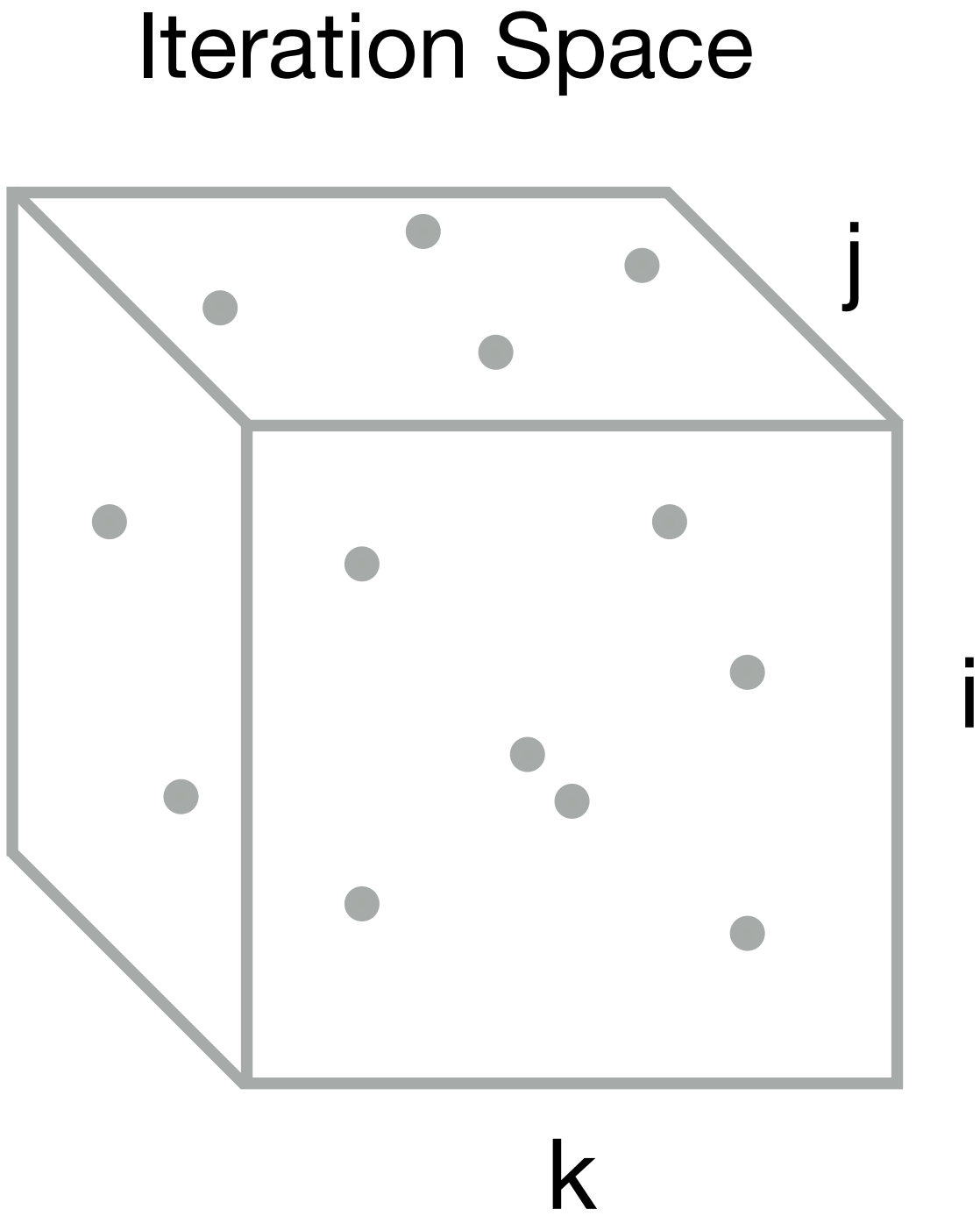
Coordinate tensor
Compressed
Singleton
Singleton

ELLPACK
Dense
Dense
Singleton

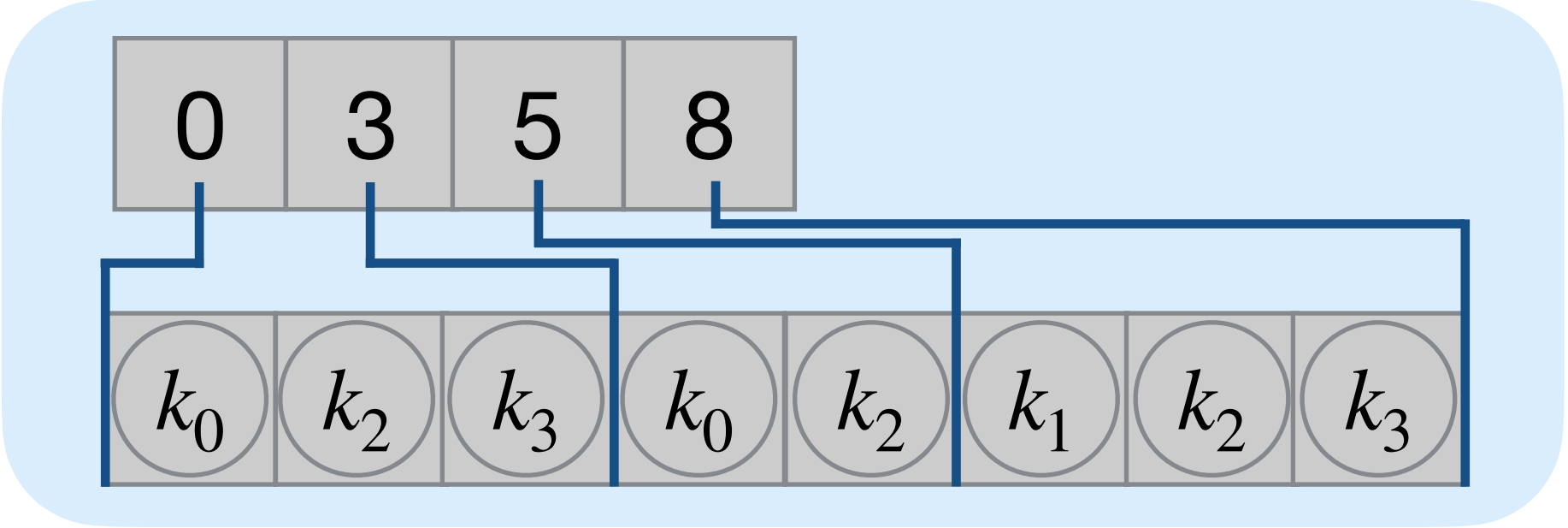
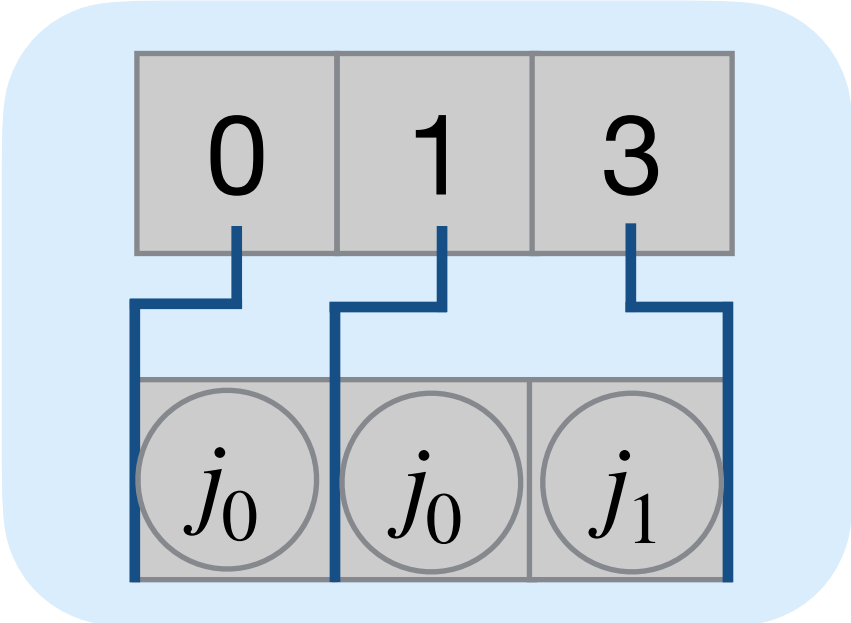
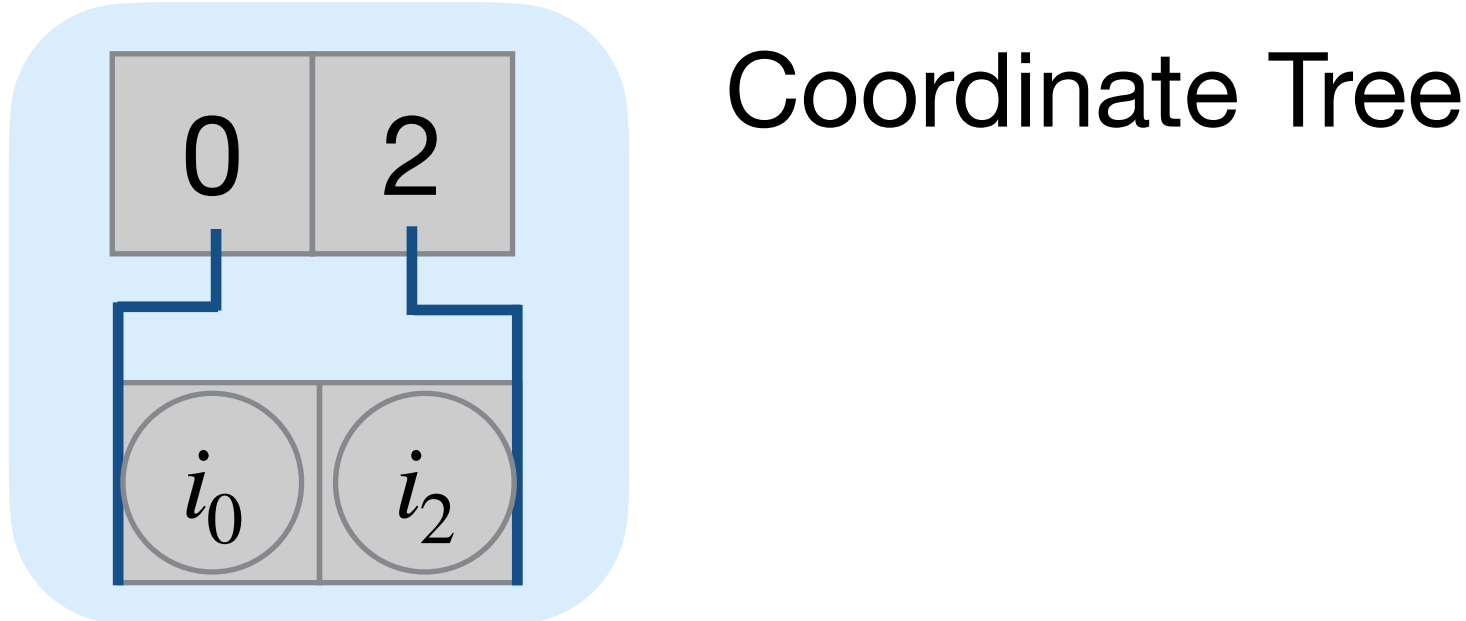
[Kincaid et al. 1989]

Block DIA
Dense
Range
Offset
Dense
Dense

Iteration graphs express iteration spaces and data structure ordering

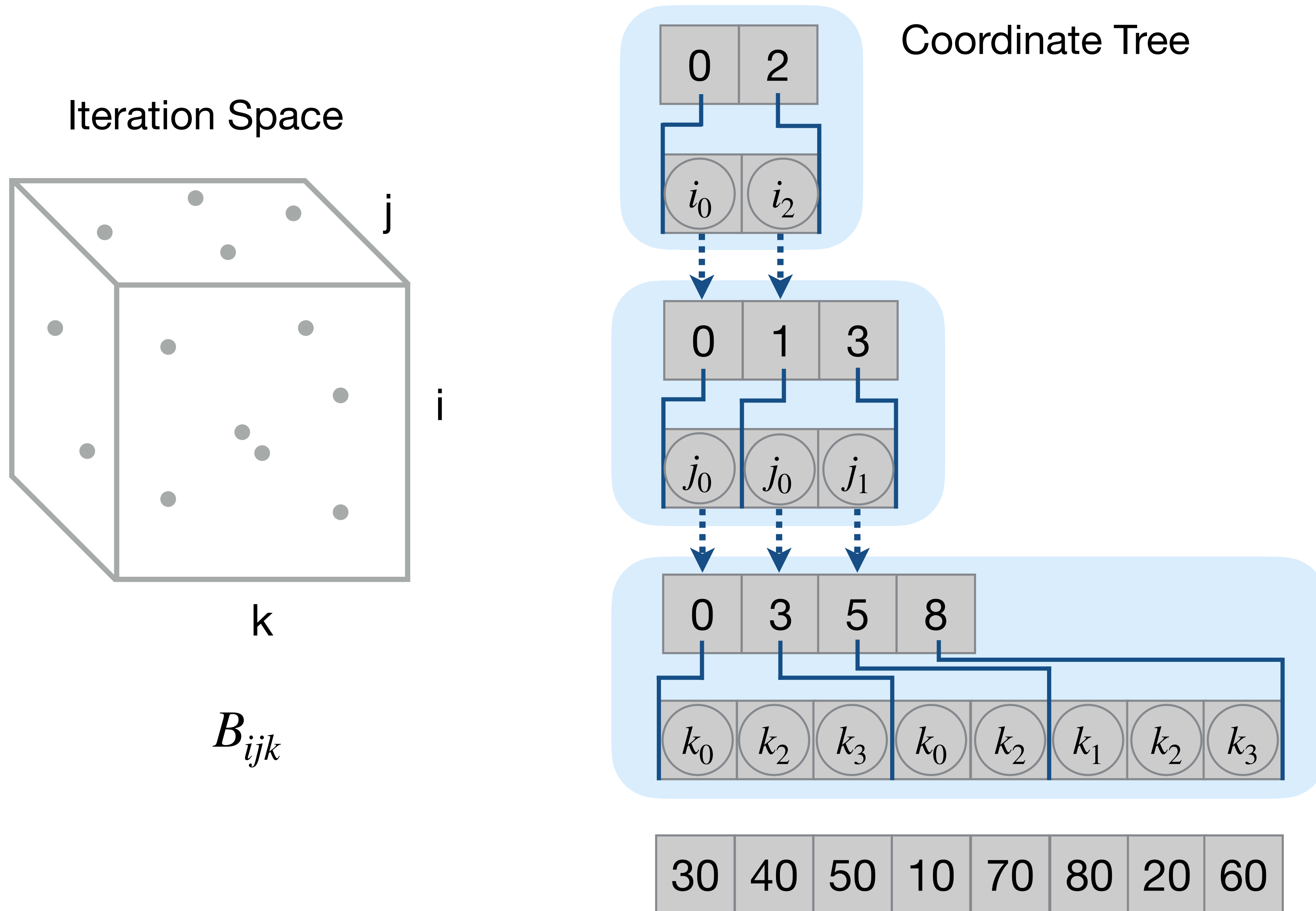


B_{ijk}

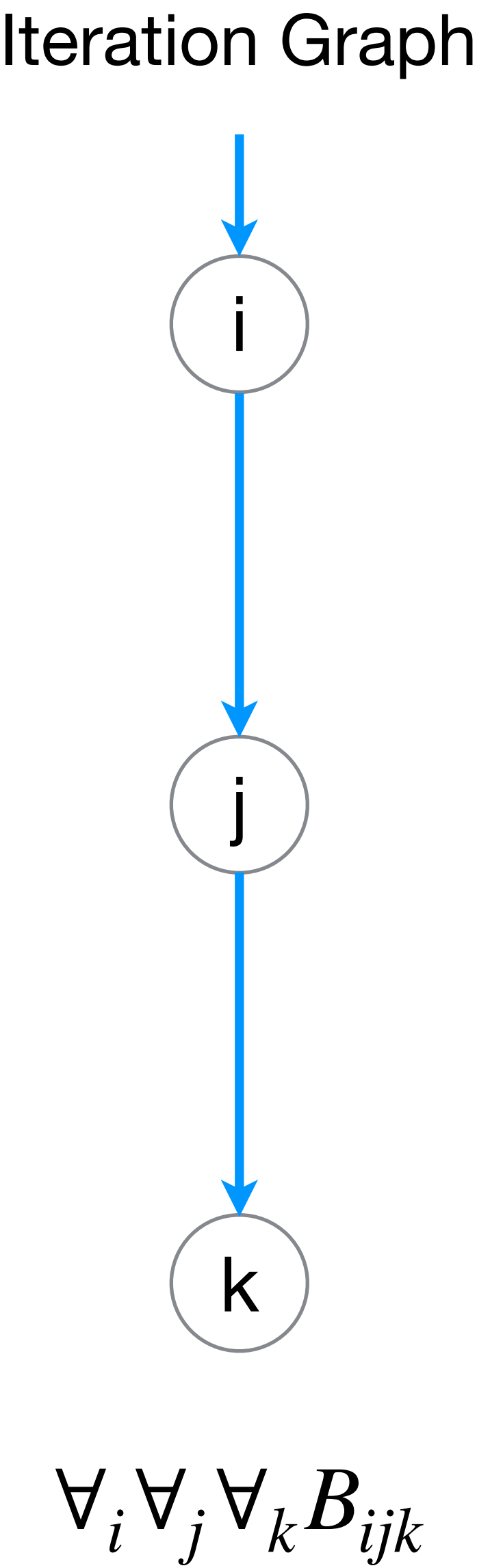
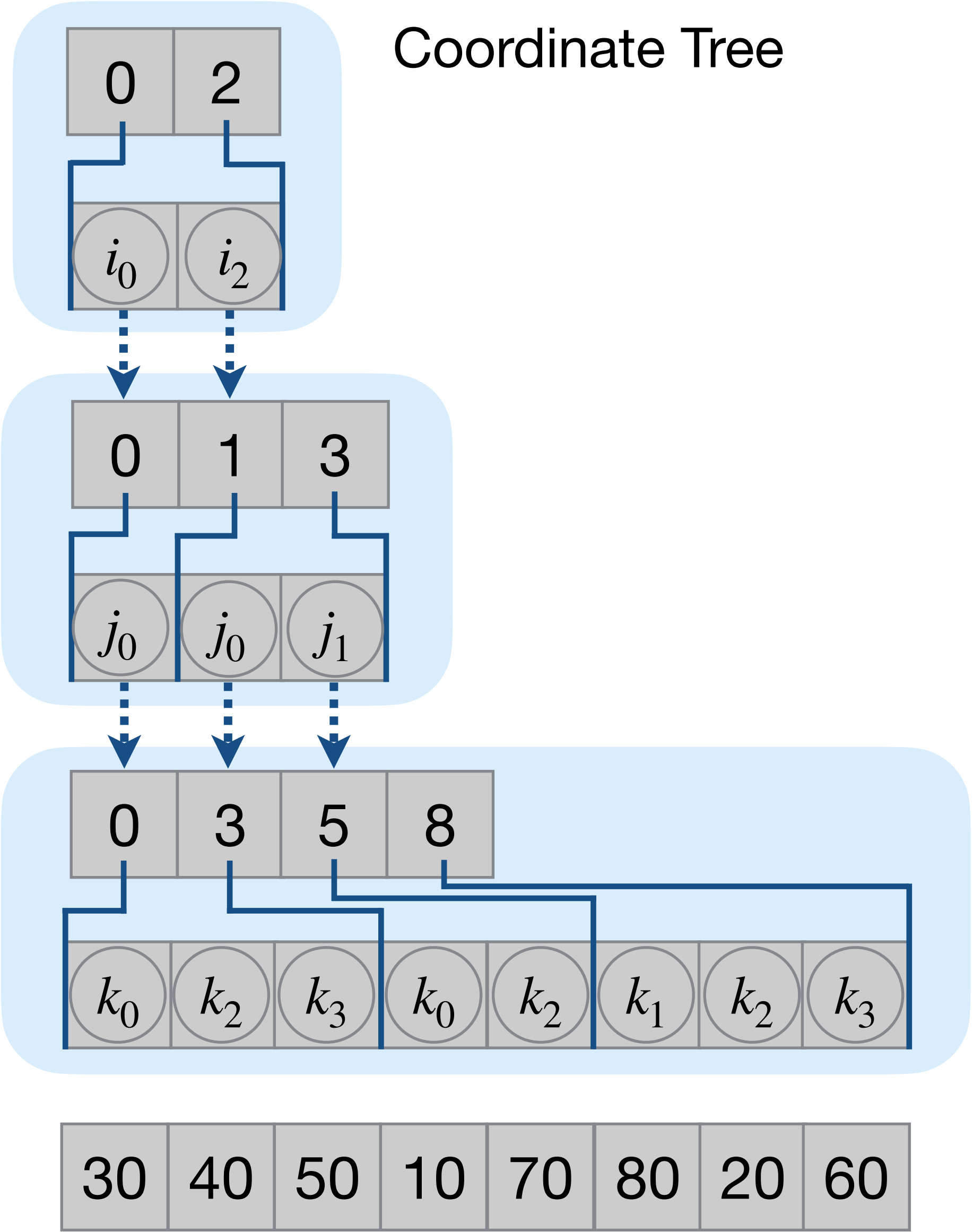
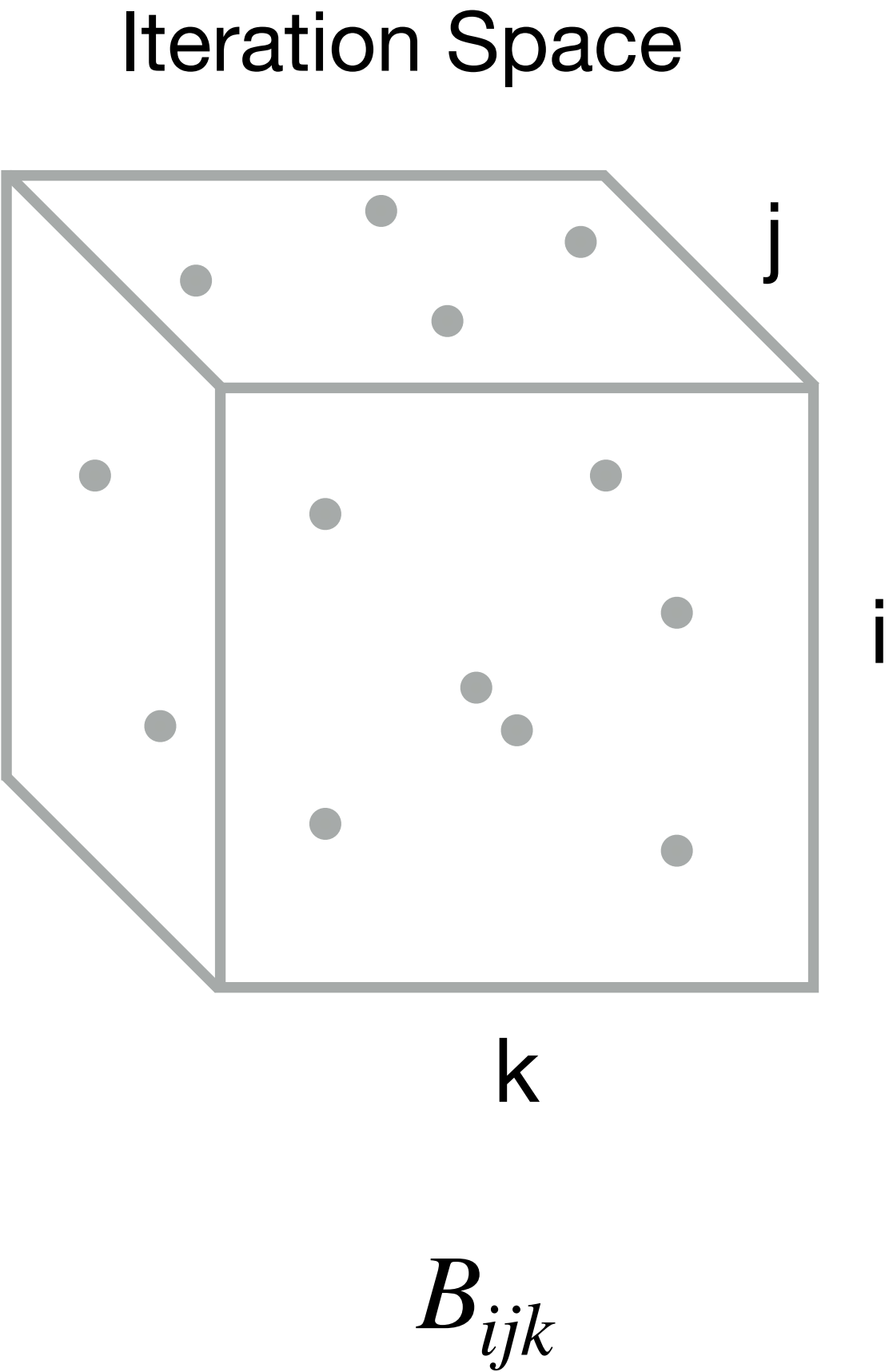


30	40	50	10	70	80	20	60
----	----	----	----	----	----	----	----

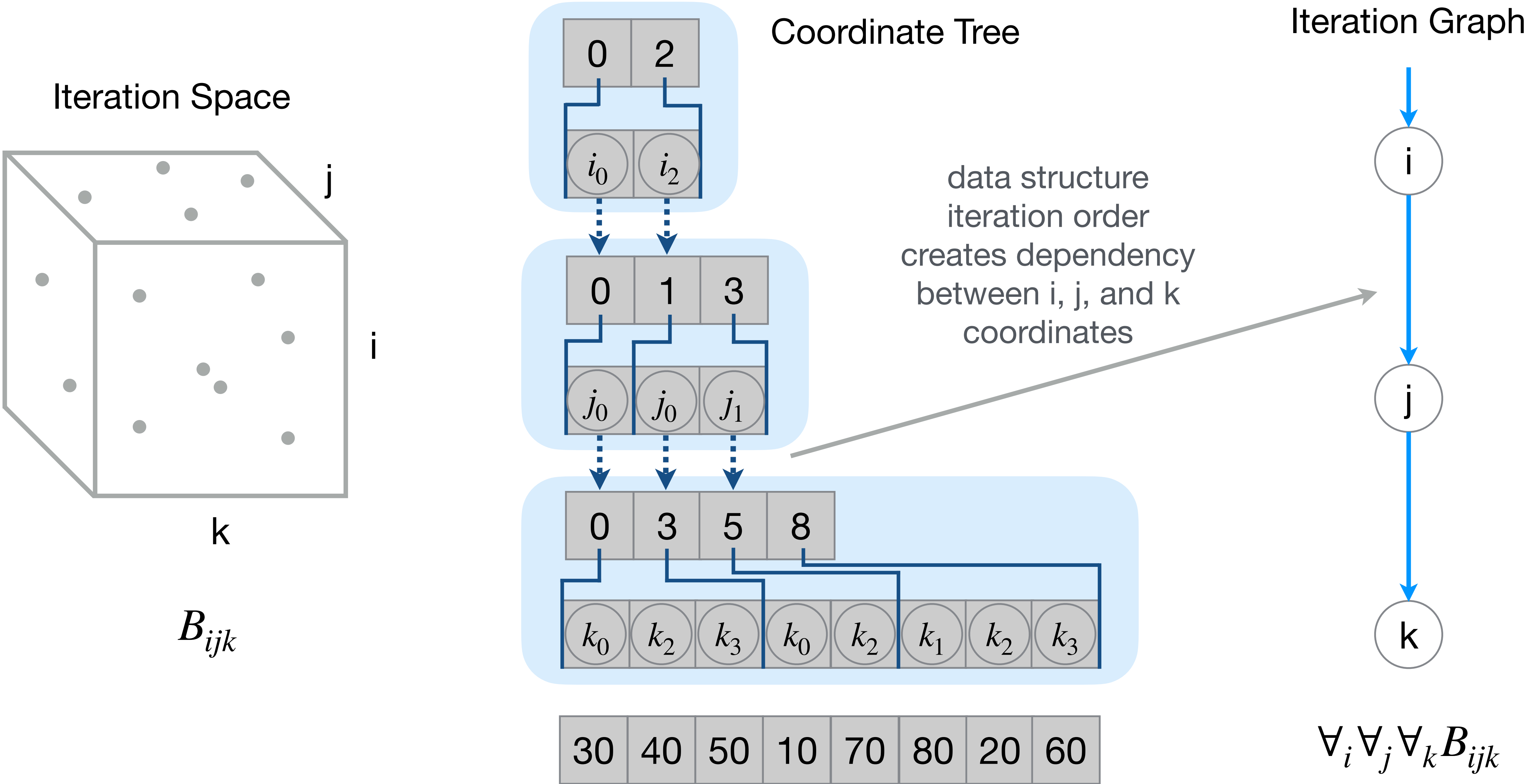
Iteration graphs express iteration spaces and data structure ordering



Iteration graphs express iteration spaces and data structure ordering



Iteration graphs express iteration spaces and data structure ordering



Sparse iteration spaces and Iteration Graphs

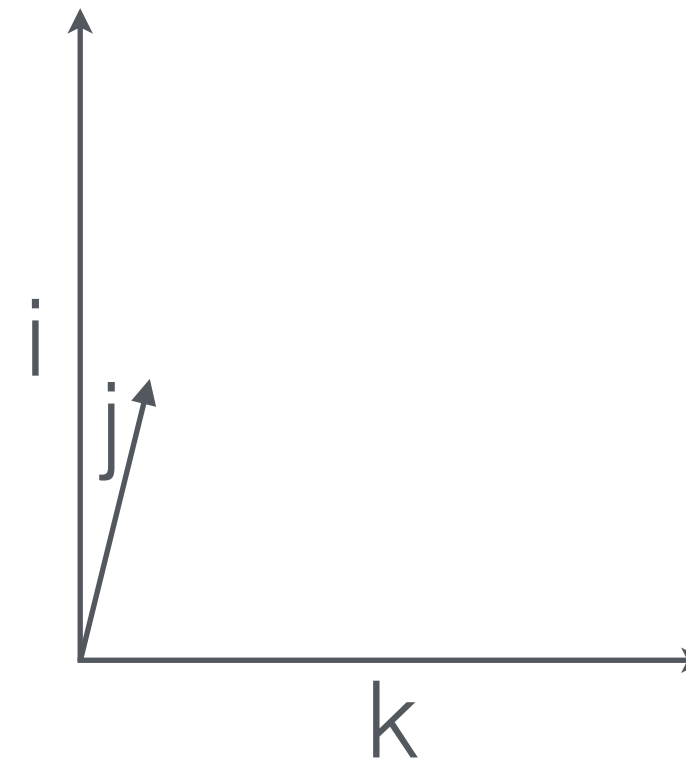
$$A_{ij} = \sum_k B_{ijk} * c_k$$

Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$

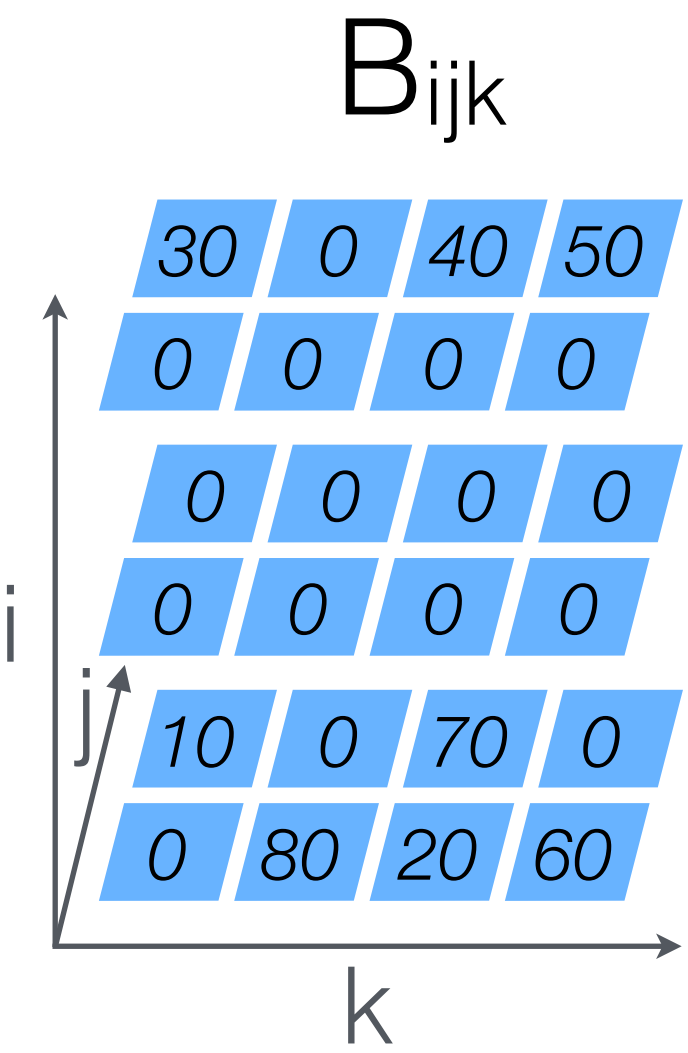

Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$



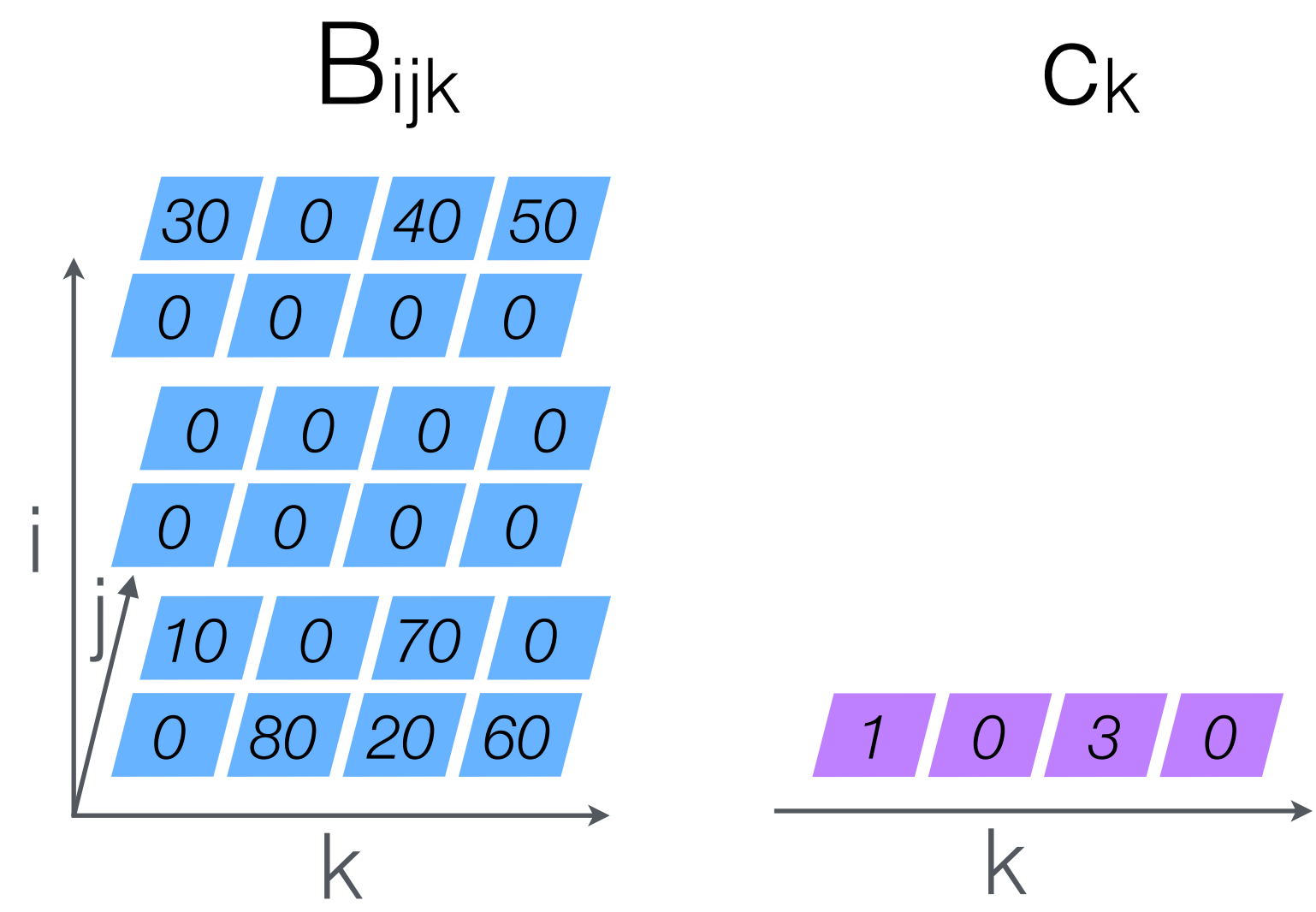
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$



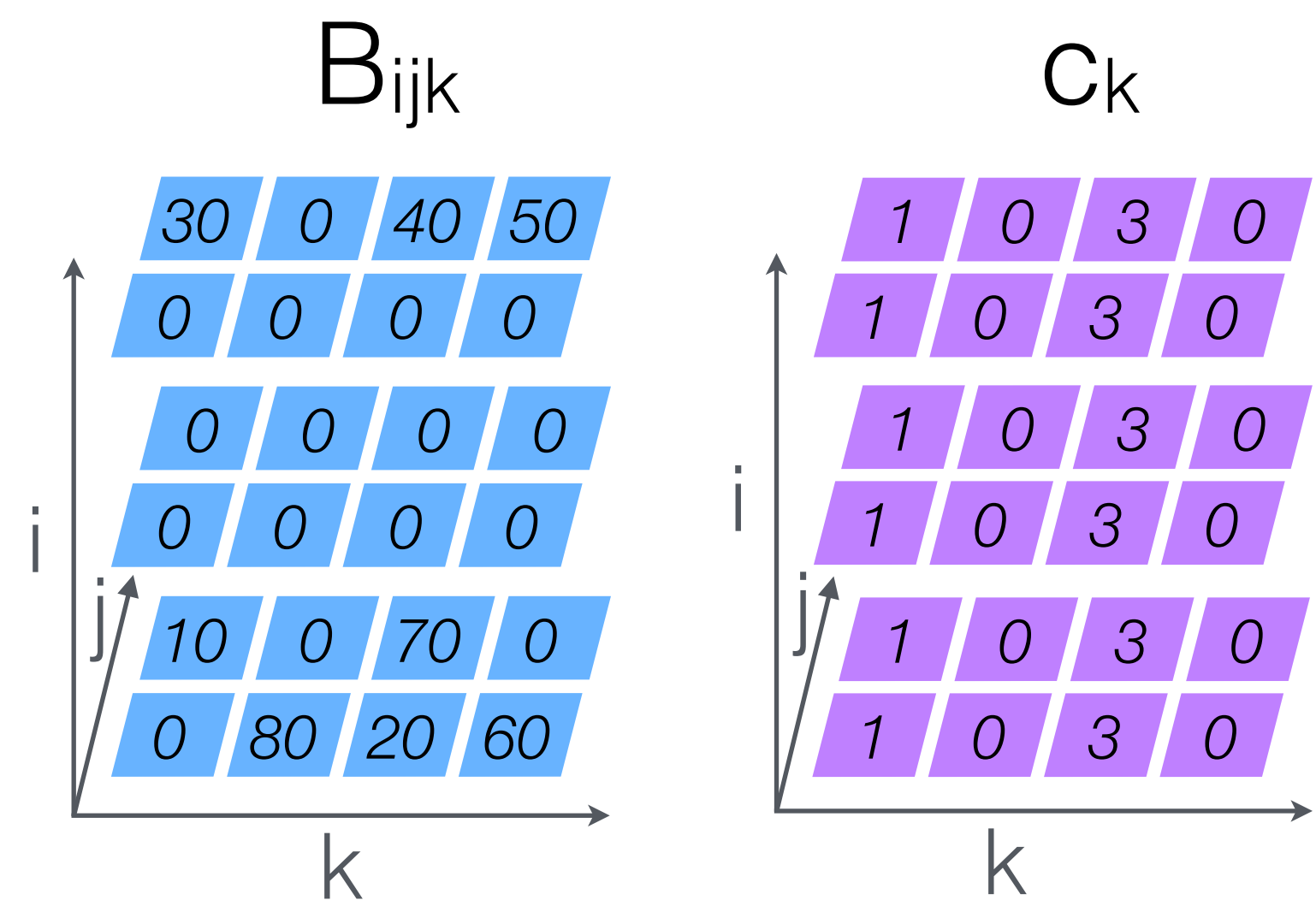
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$



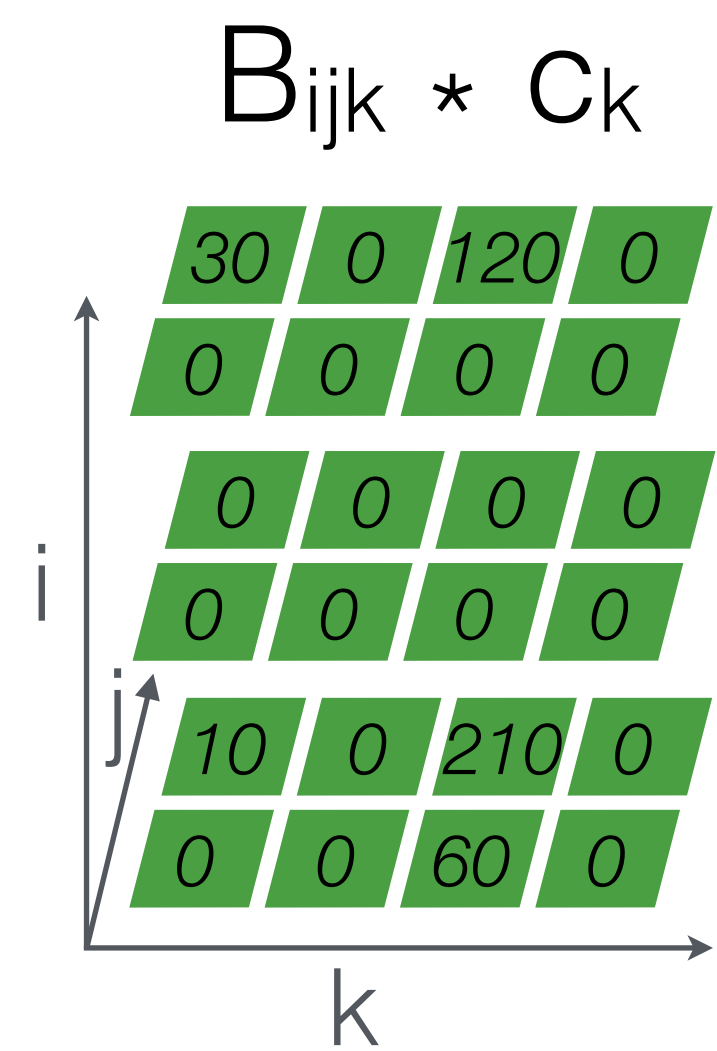
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$



Sparse iteration spaces and Iteration Graphs

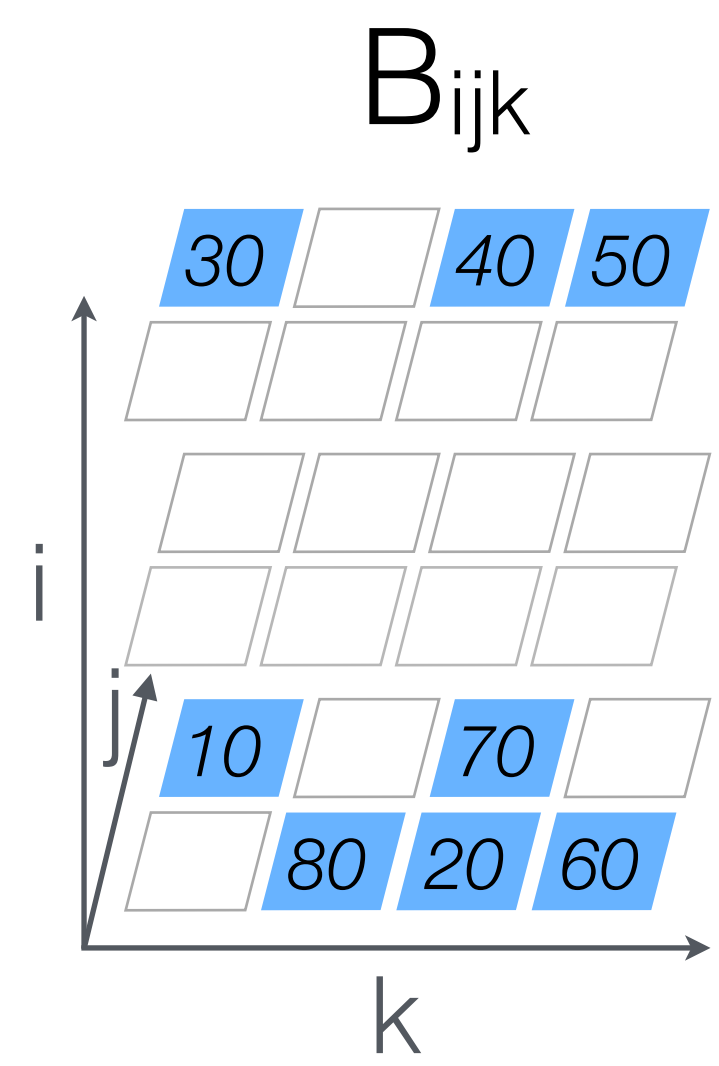
$$A_{ij} = \sum_k B_{ijk} * c_k$$



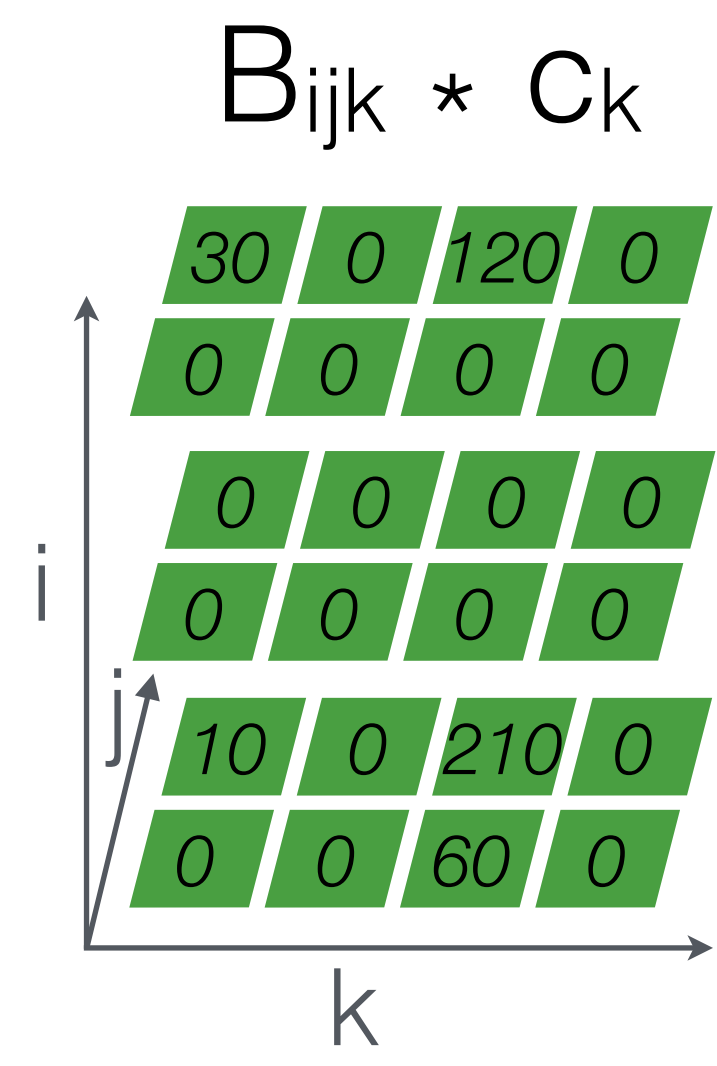
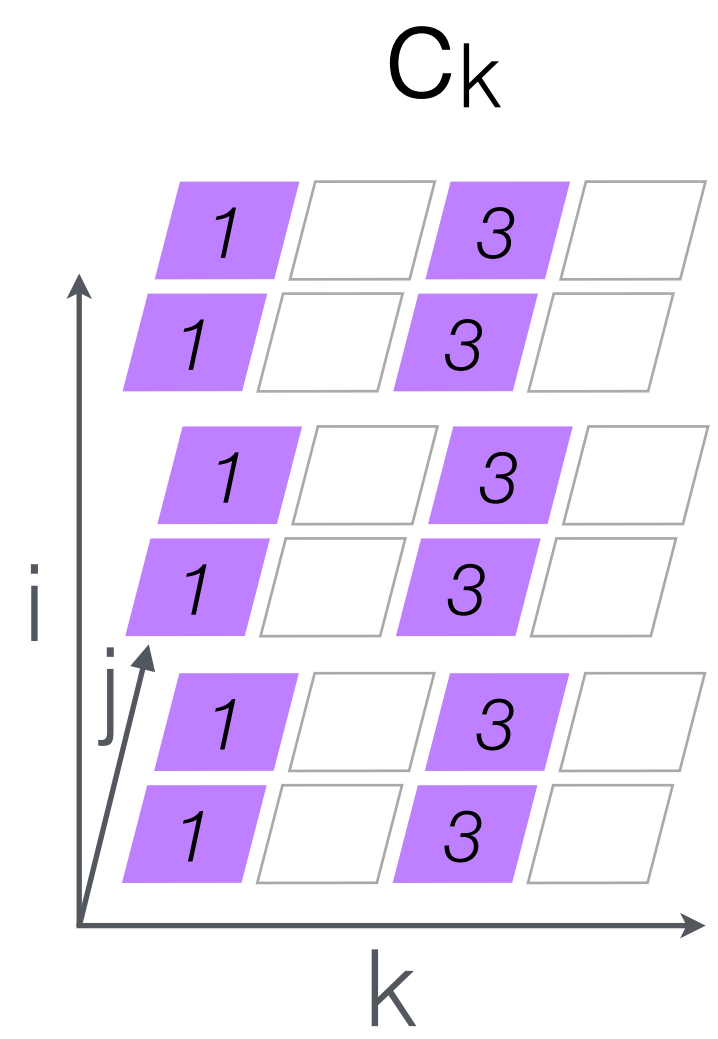
Dense

Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



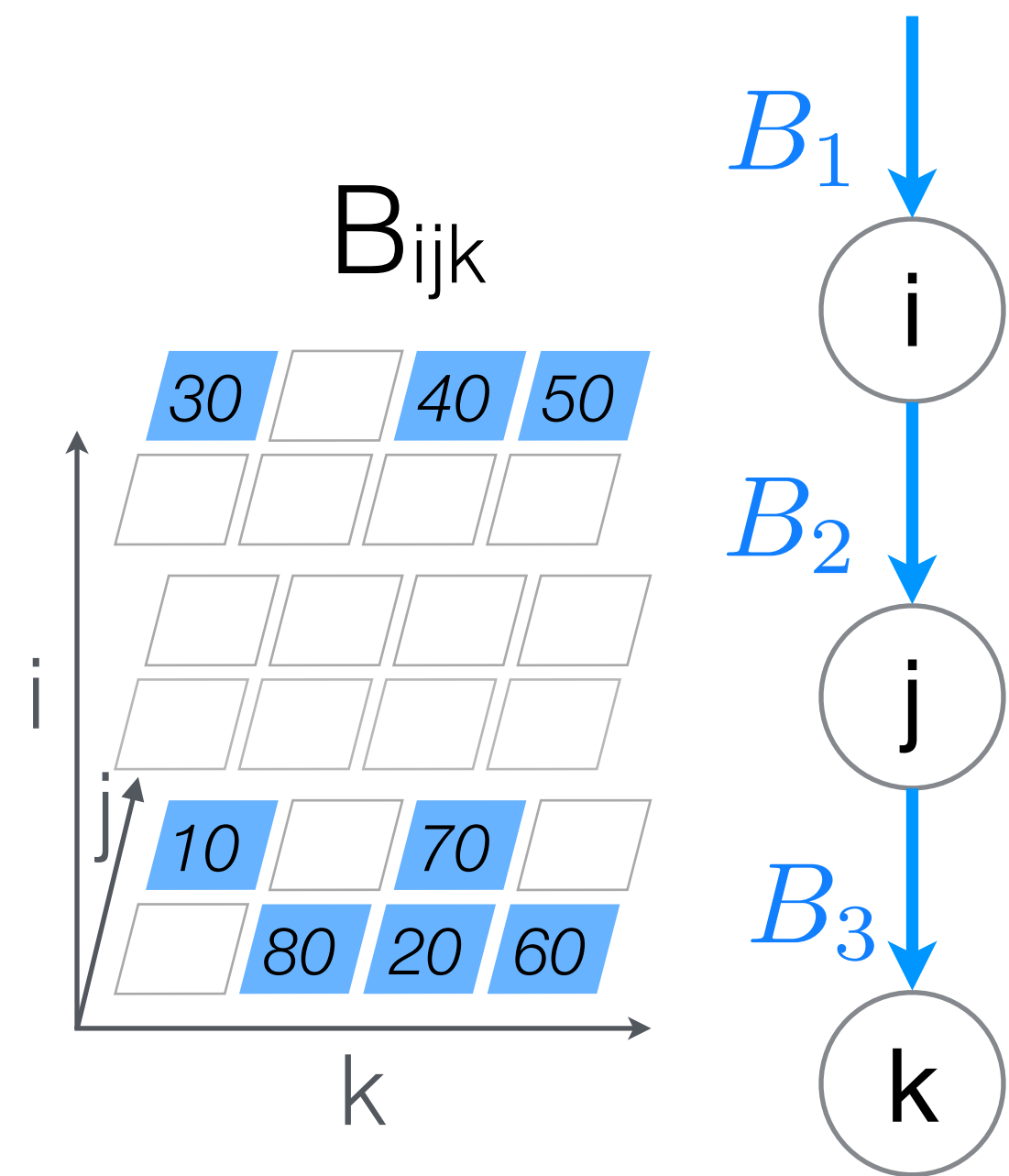
Sparse



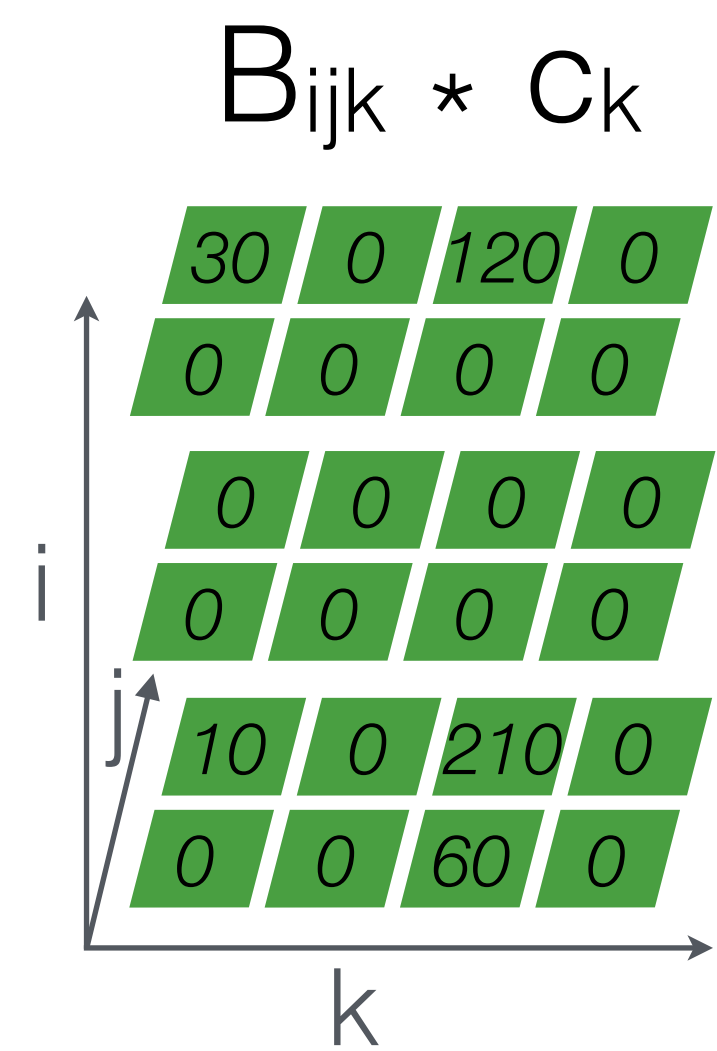
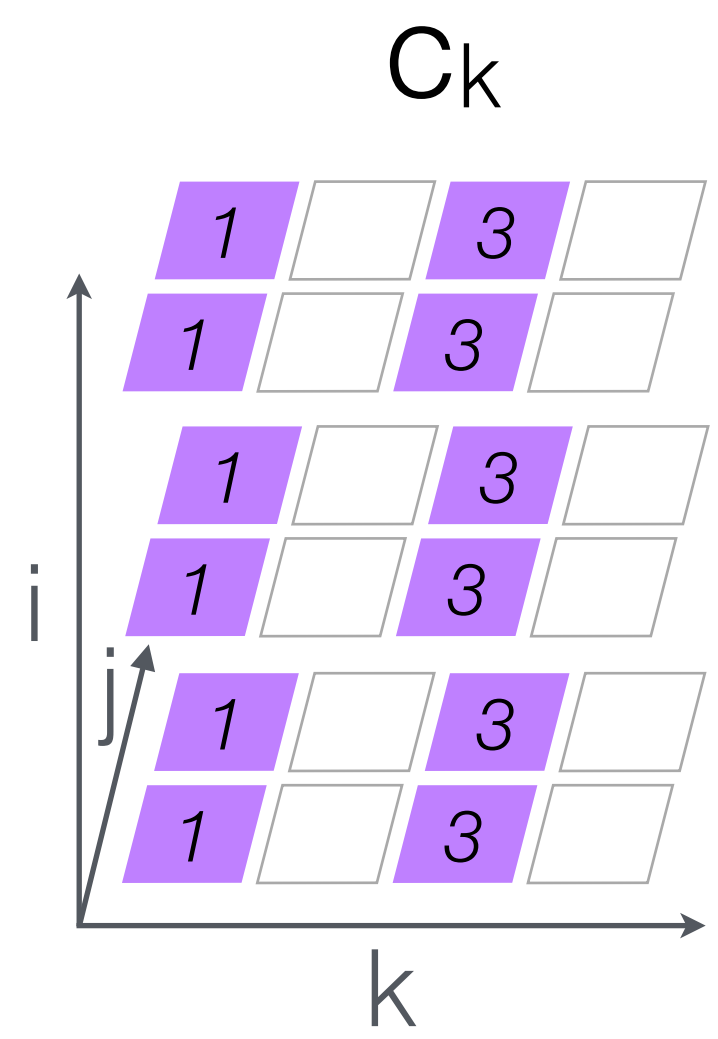
Dense

Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



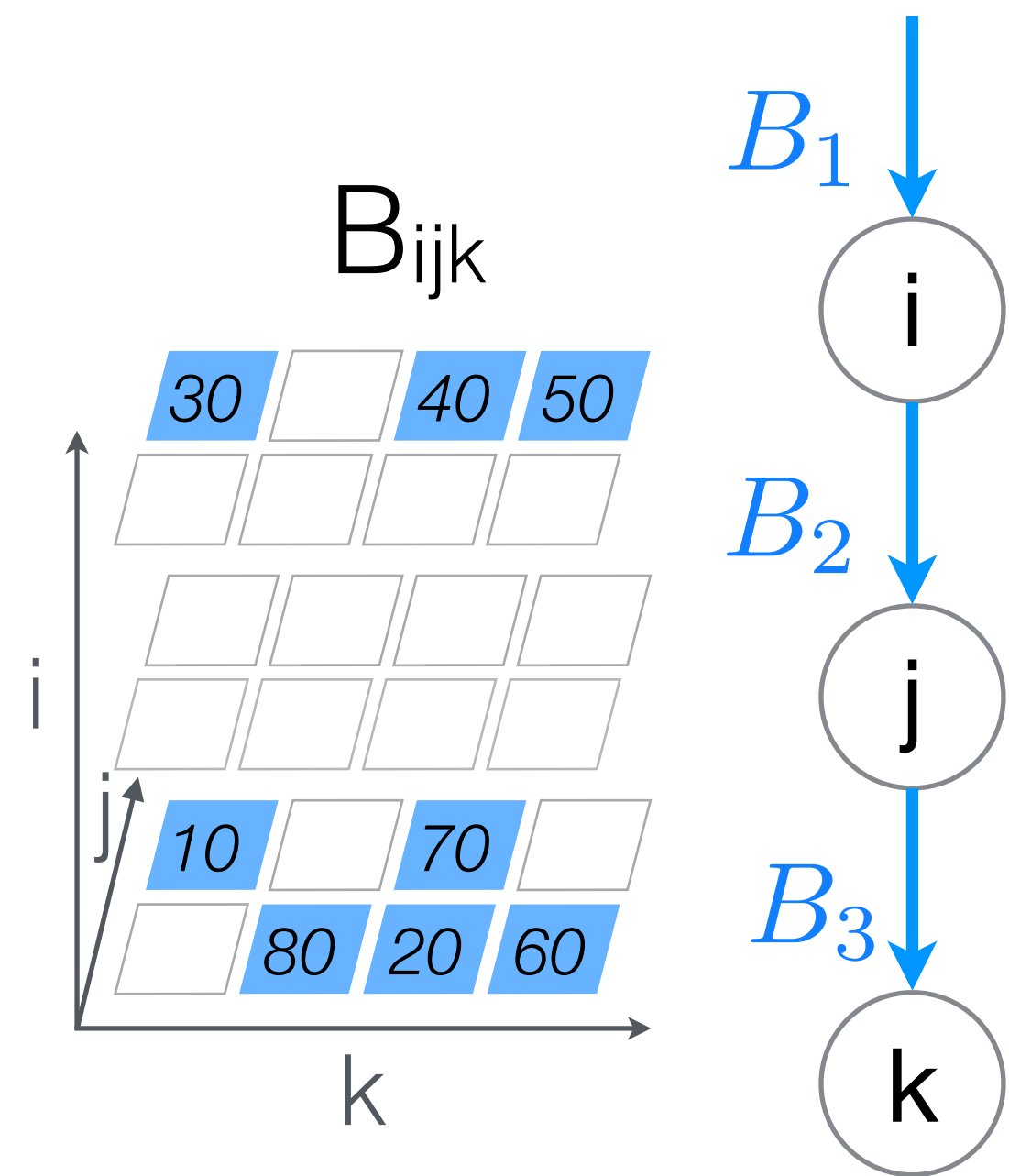
Sparse



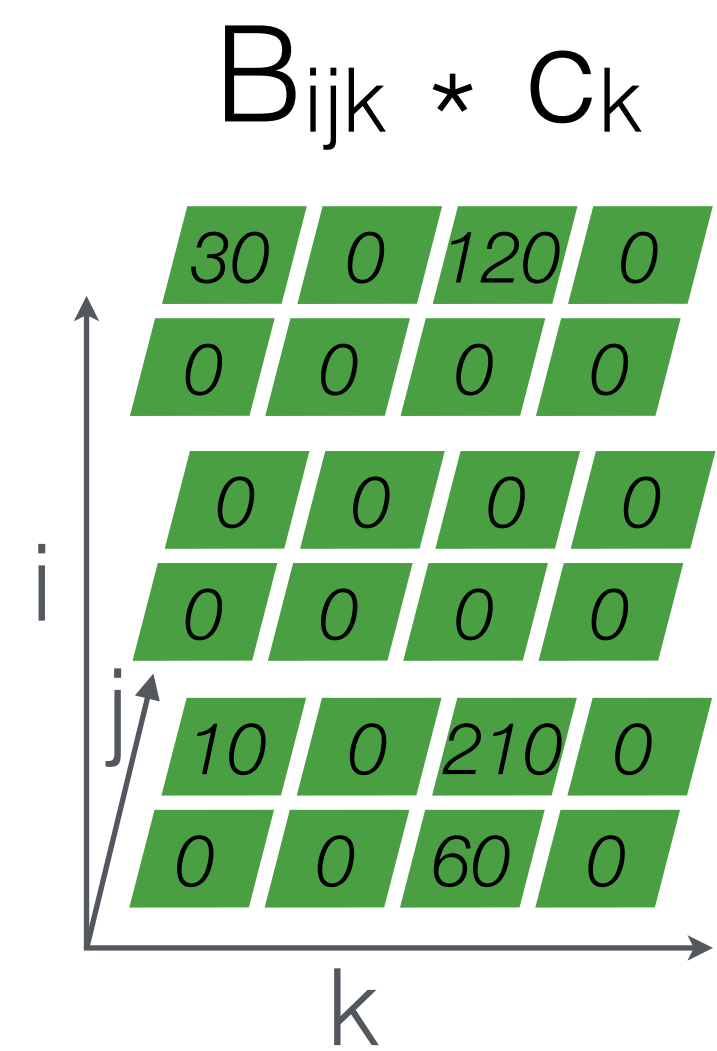
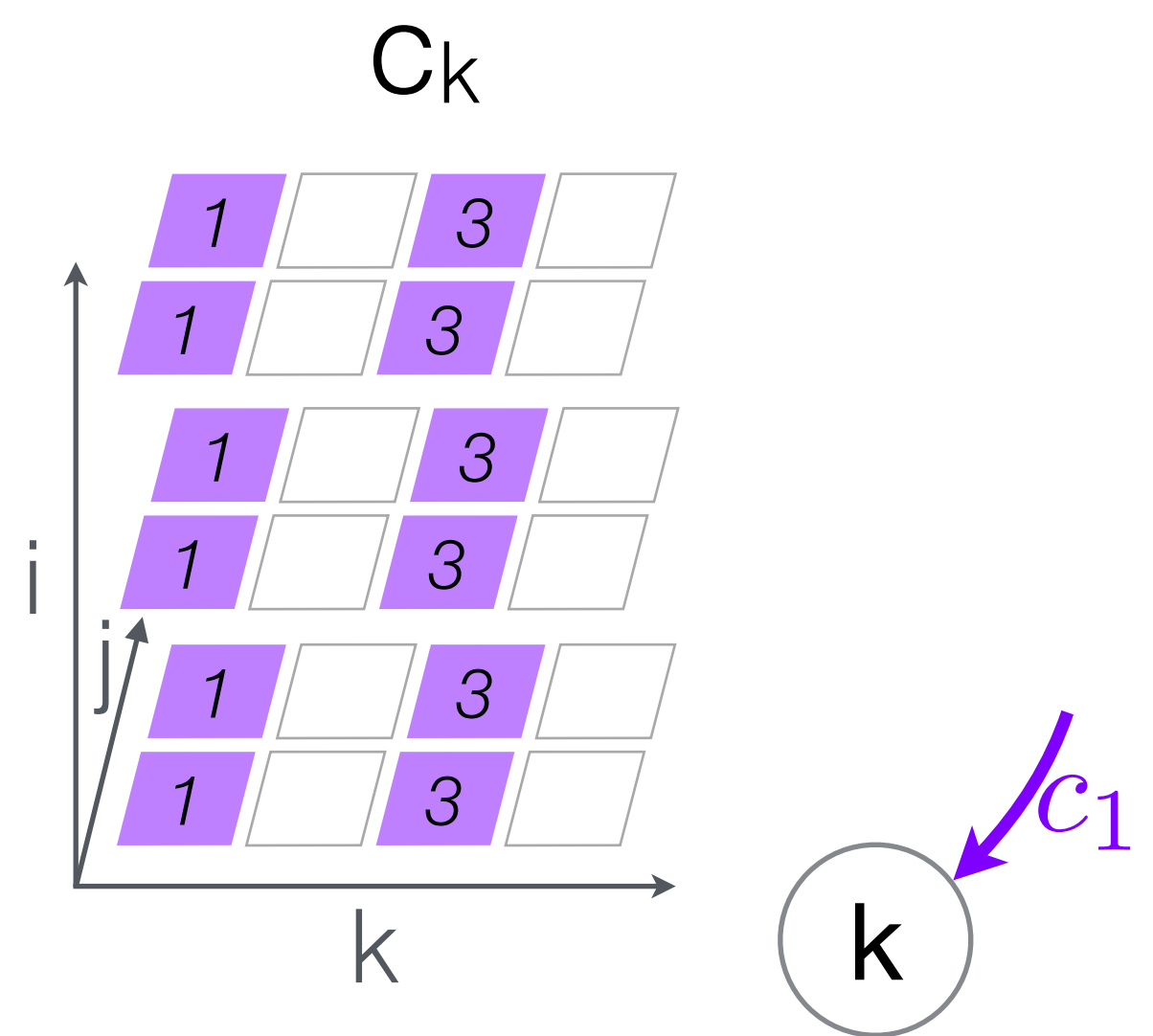
Dense

Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$



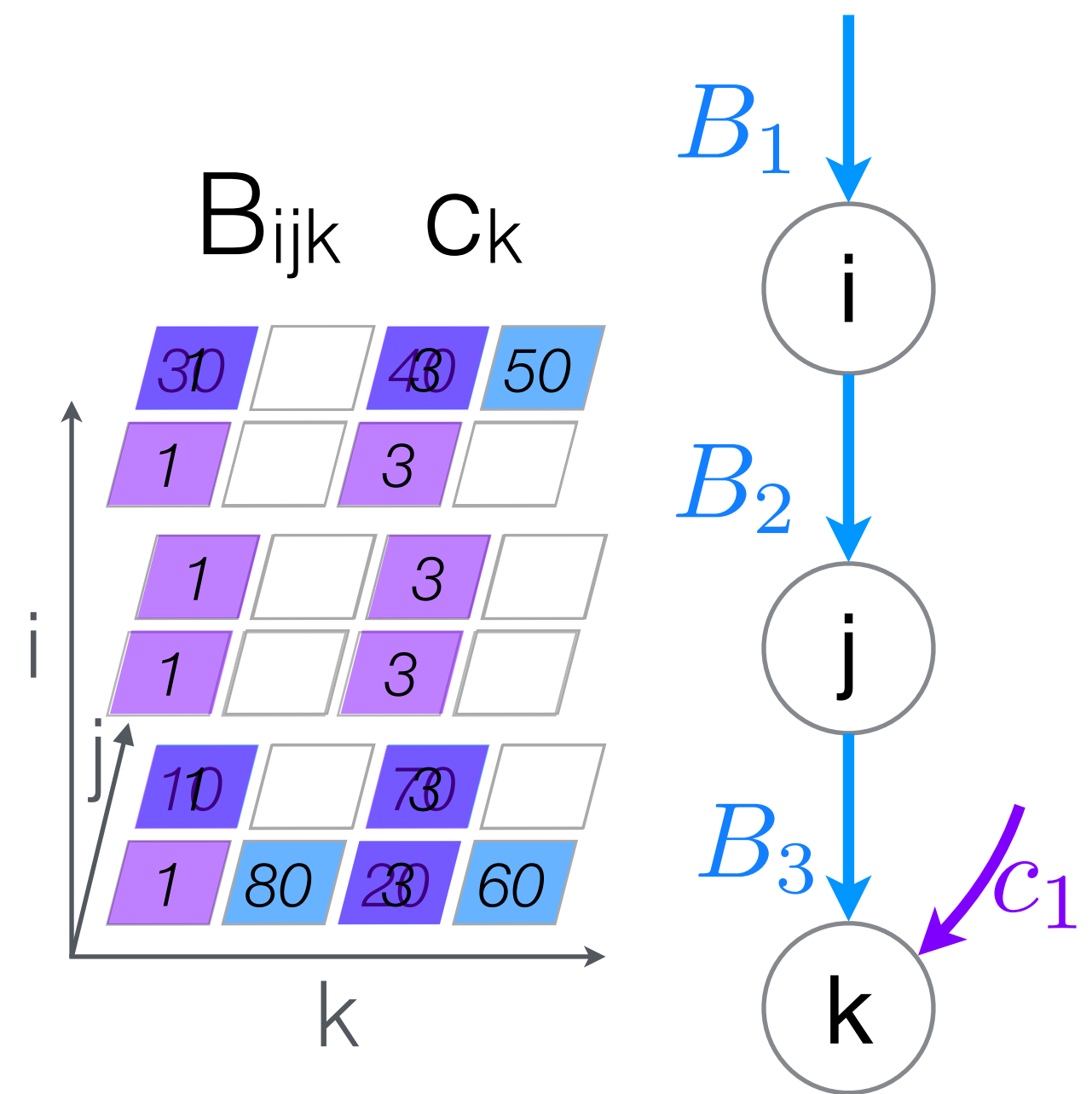
Sparse



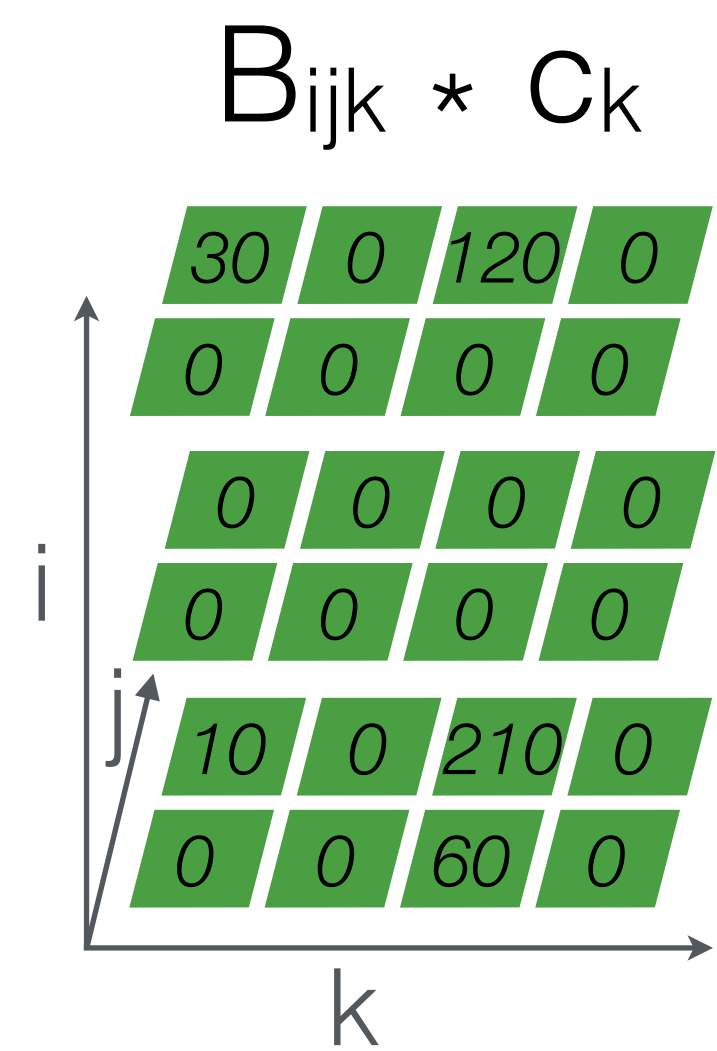
Dense

Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$



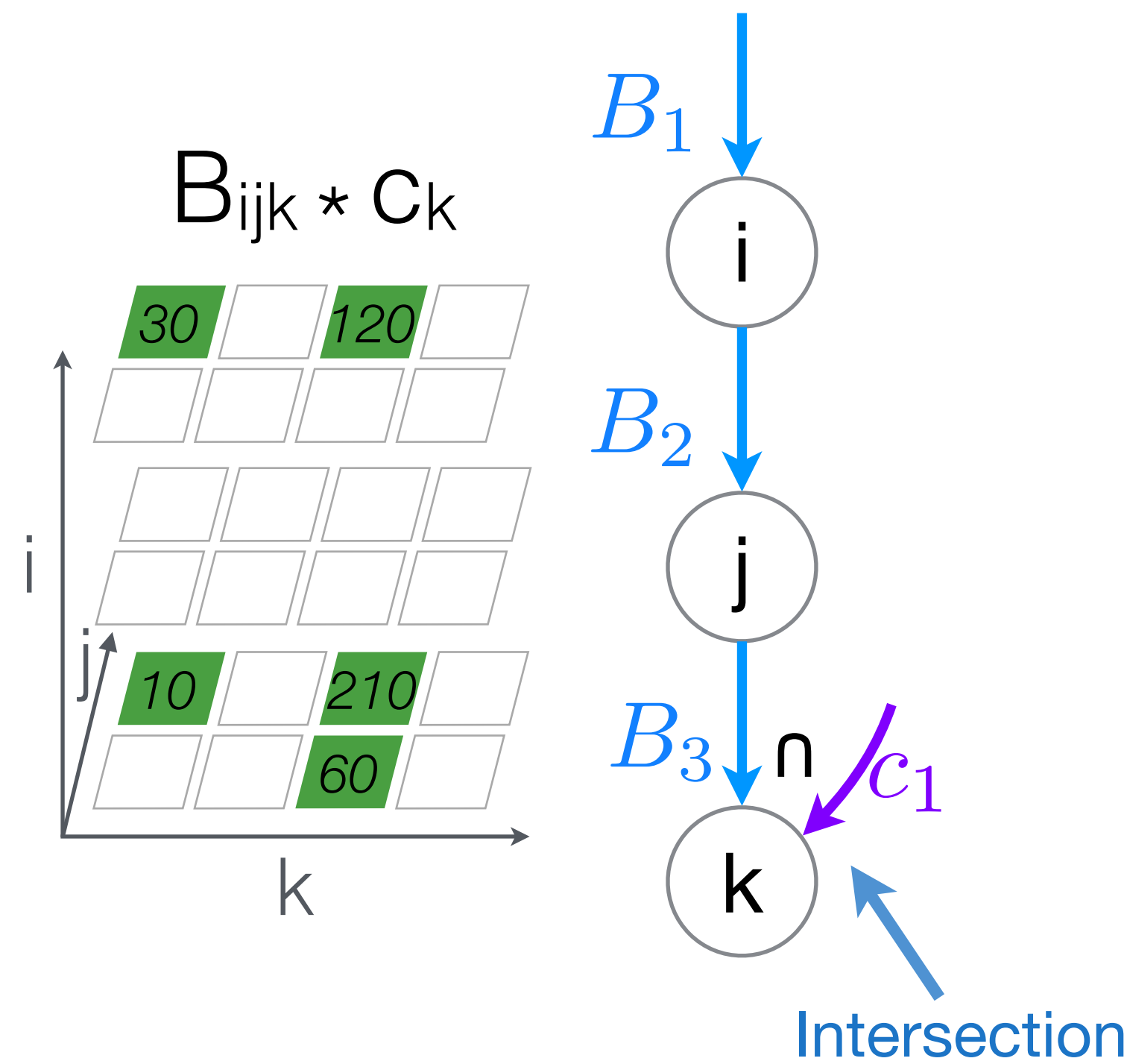
Sparse



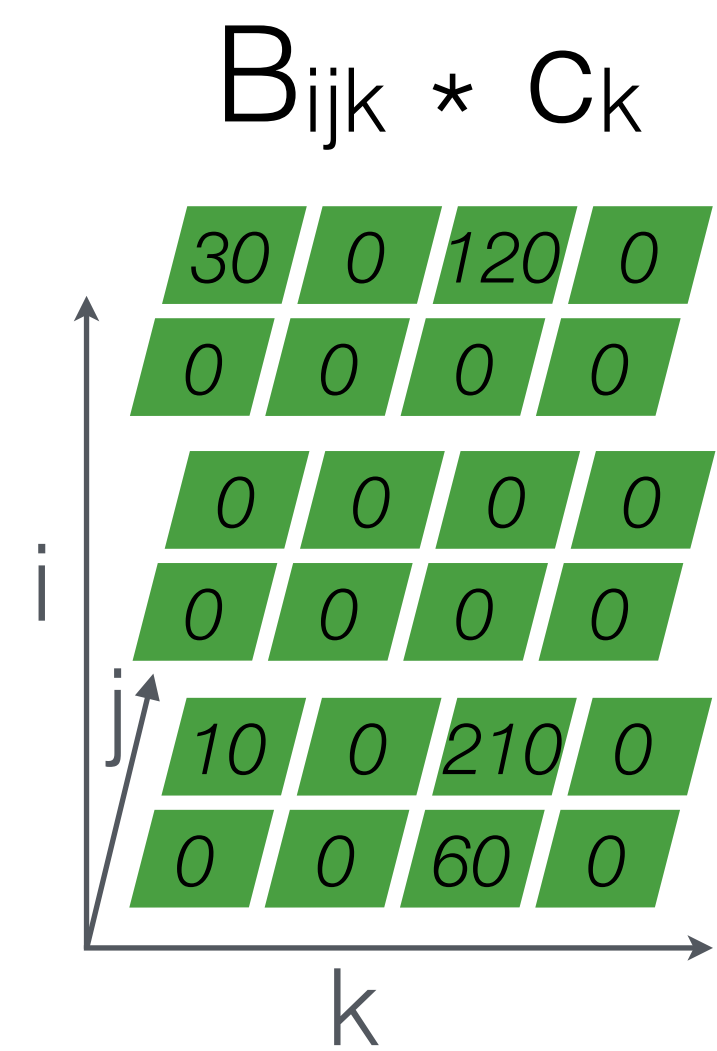
Dense

Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$

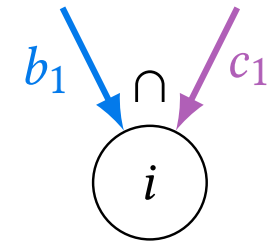


Sparse

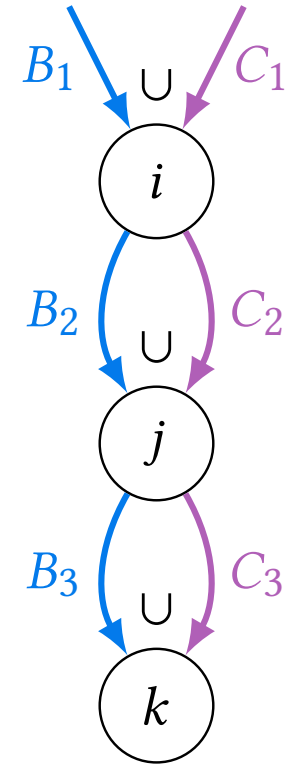


Dense

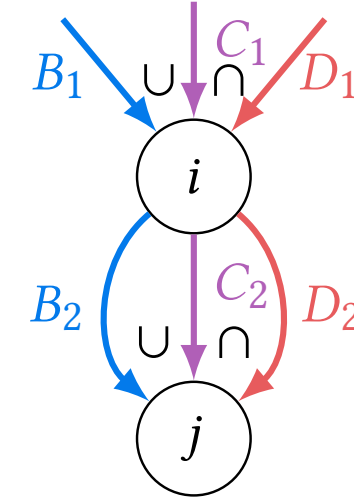
Iteration graph examples



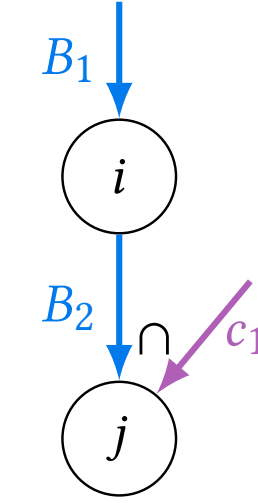
$$\frac{\forall i \, b_i \cap c_i}{i \in b_1 \cap c_1}$$



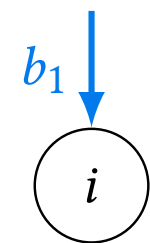
$$\frac{\forall i \forall j \forall k \, B_{ijk} \cup C_{ijk}}{i \in B_1 \cup C_1 \\ j \in B_2 \cup C_2 \\ k \in B_3 \cup C_3}$$



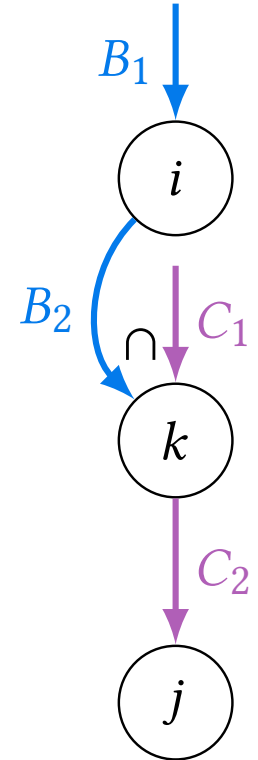
$$\frac{\forall i \forall j \, (B_{ij} \cup C_{ij}) \cap D_{ij}}{i \in (B_1 \cup C_1) \cap D_1 \\ j \in (B_2 \cup C_2) \cap D_2}$$



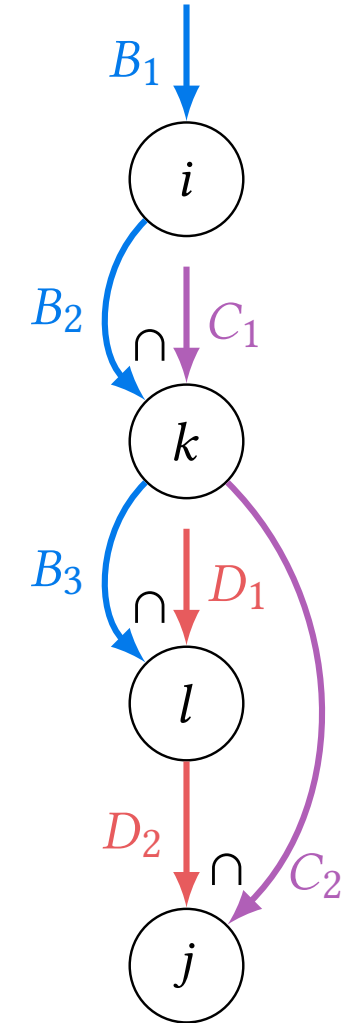
$$\frac{\forall i \forall j \, B_{ij} \cap c_j}{i \in B_1 \cap \mathbb{U}_i \\ j \in B_2 \cap c_1}$$



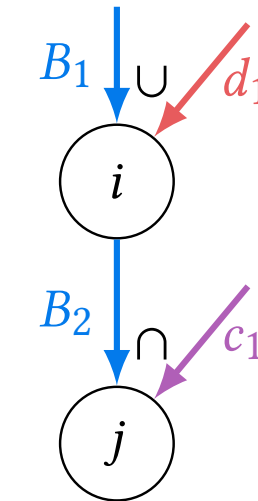
$$\frac{\forall i \, \alpha \cup b_i}{i \in \mathbb{U}_i \cap b_1}$$



$$\frac{\forall i \forall k \forall j \, B_{ik} \cap C_{kj}}{i \in B_1 \cap \mathbb{U}_i \\ k \in B_2 \cap C_1 \\ j \in \mathbb{U}_j \cap C_2}$$



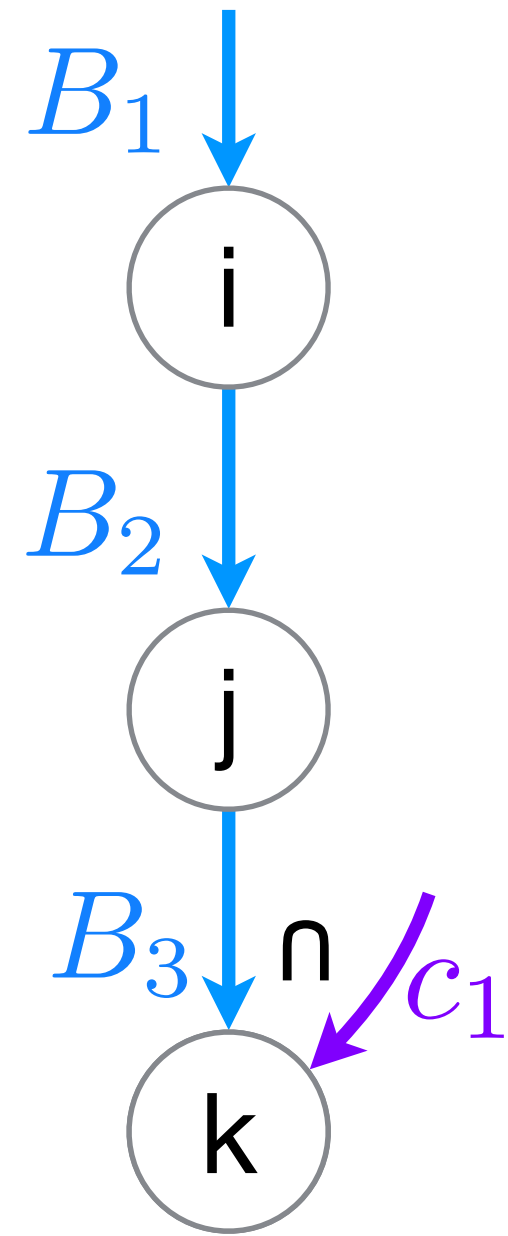
$$\frac{\forall i \forall k \forall l \forall j \, B_{ikl} \cap C_{kj} \cap D_{lj}}{i \in B_1 \cap \mathbb{U}_i \cap \mathbb{U}_i \\ k \in B_2 \cap C_1 \cap \mathbb{U}_k \\ l \in B_3 \cap \mathbb{U}_l \cap D_1 \\ j \in \mathbb{U}_j \cap C_2 \cap D_2}$$



$$\frac{\forall i (\forall j \, B_{ij} \cap c_j) \cup d_i}{i \in (B_1 \cap \mathbb{U}_i) \cup d_1 \\ j \in B_2 \cap c_1}$$

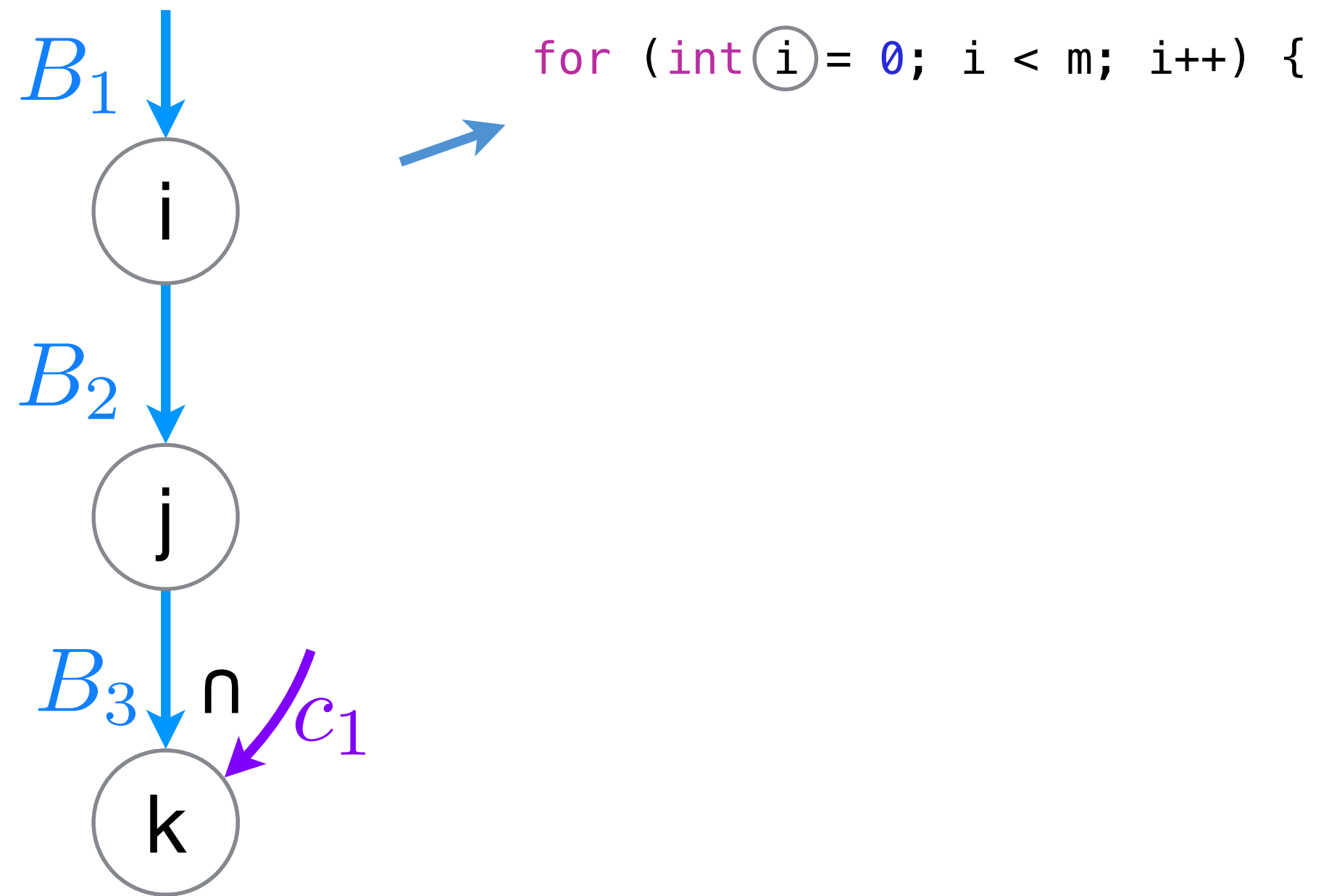
Iteration graphs are lowered to sparse code

$$A_{ij} = \sum_k B_{ijk} c_k$$



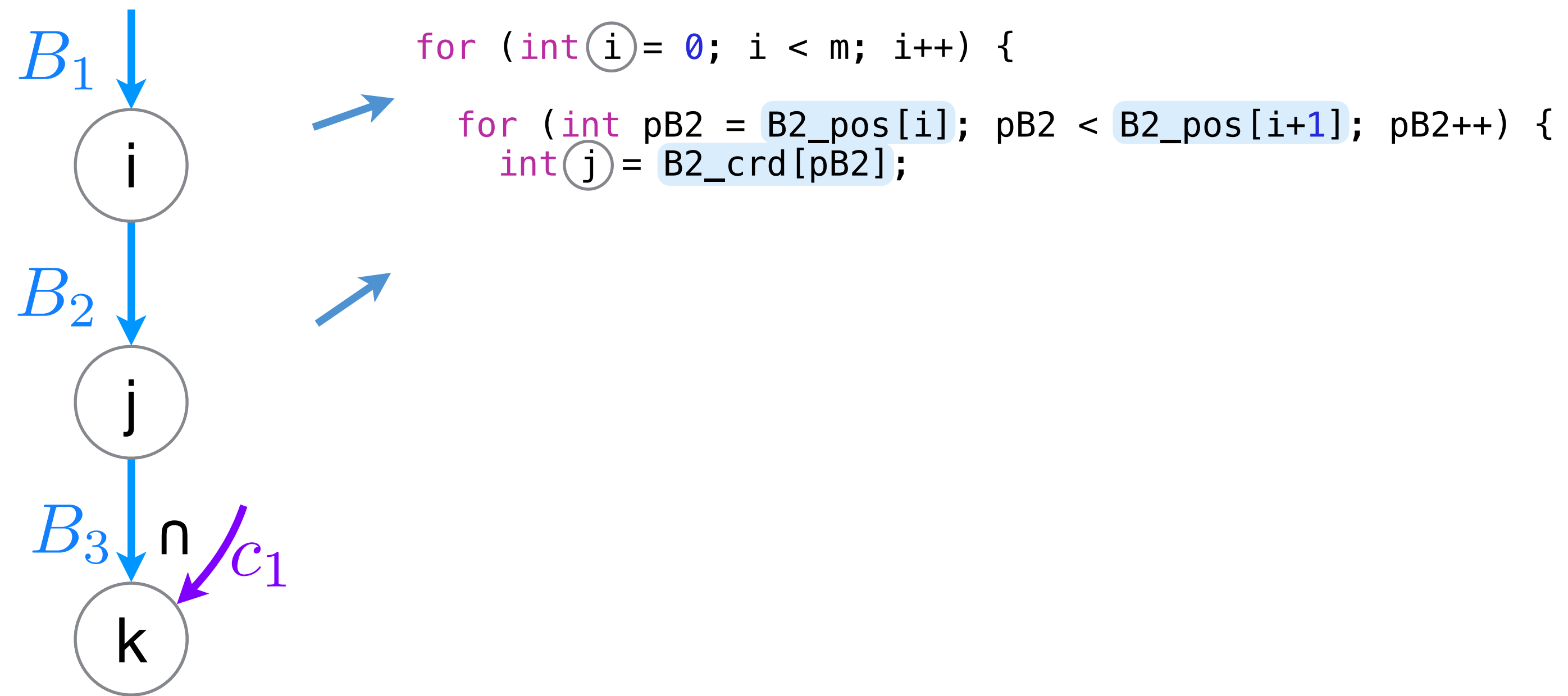
Iteration graphs are lowered to sparse code

$$A_{ij} = \sum_k B_{ijk} c_k$$



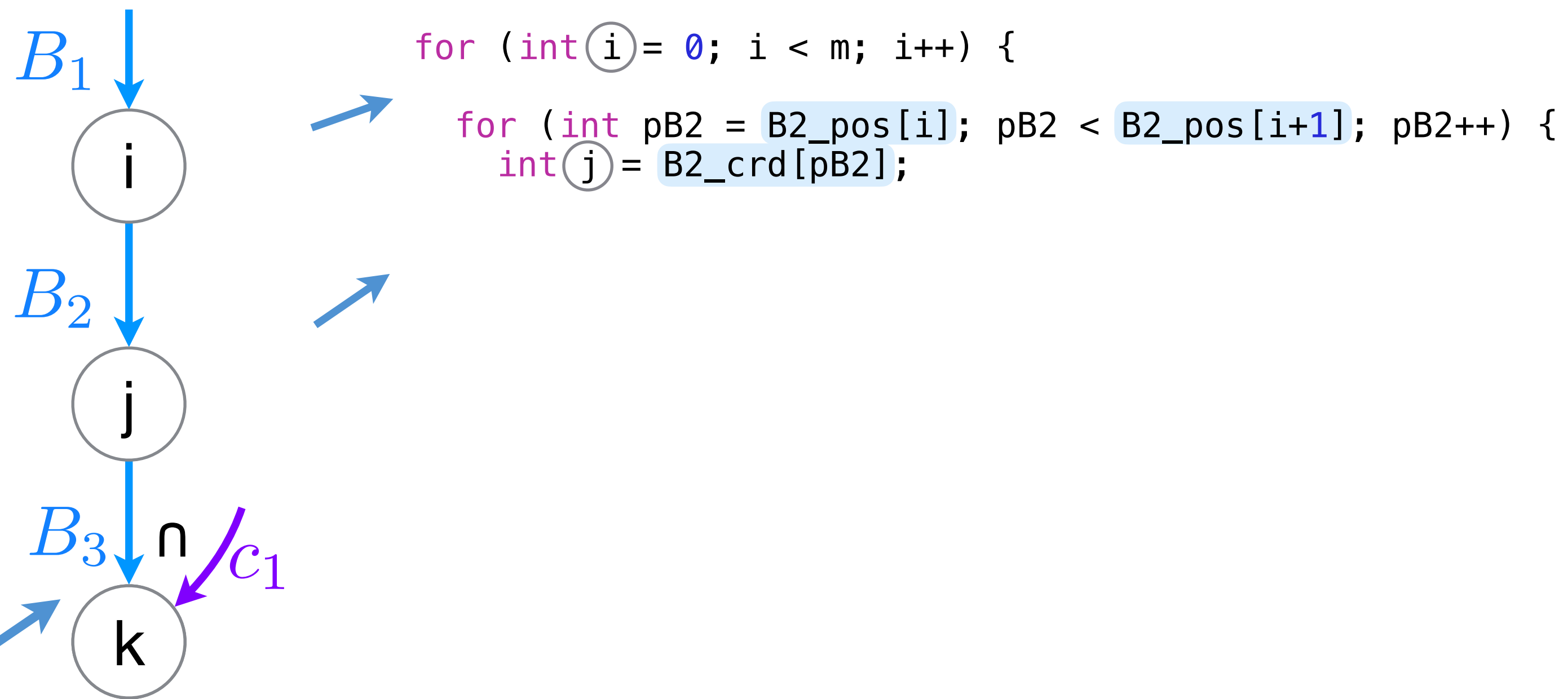
Iteration graphs are lowered to sparse code

$$A_{ij} = \sum_k B_{ijk} c_k$$



Iteration graphs are lowered to sparse code

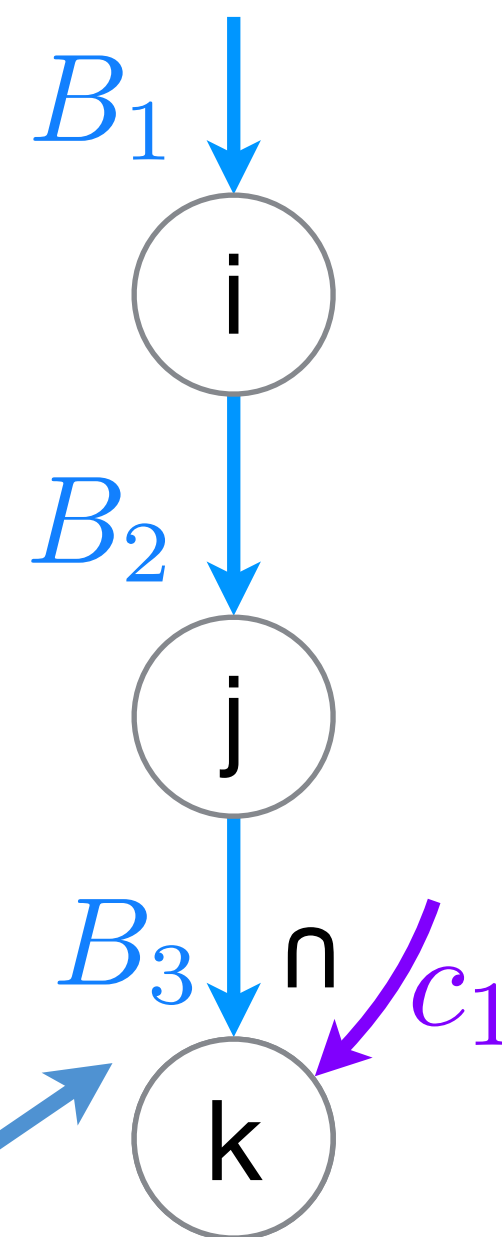
$$A_{ij} = \sum_k B_{ijk} c_k$$



Key operation is to coiterate
over data structures

Iteration graphs are lowered to sparse code

$$A_{ij} = \sum_k B_{ijk} c_k$$



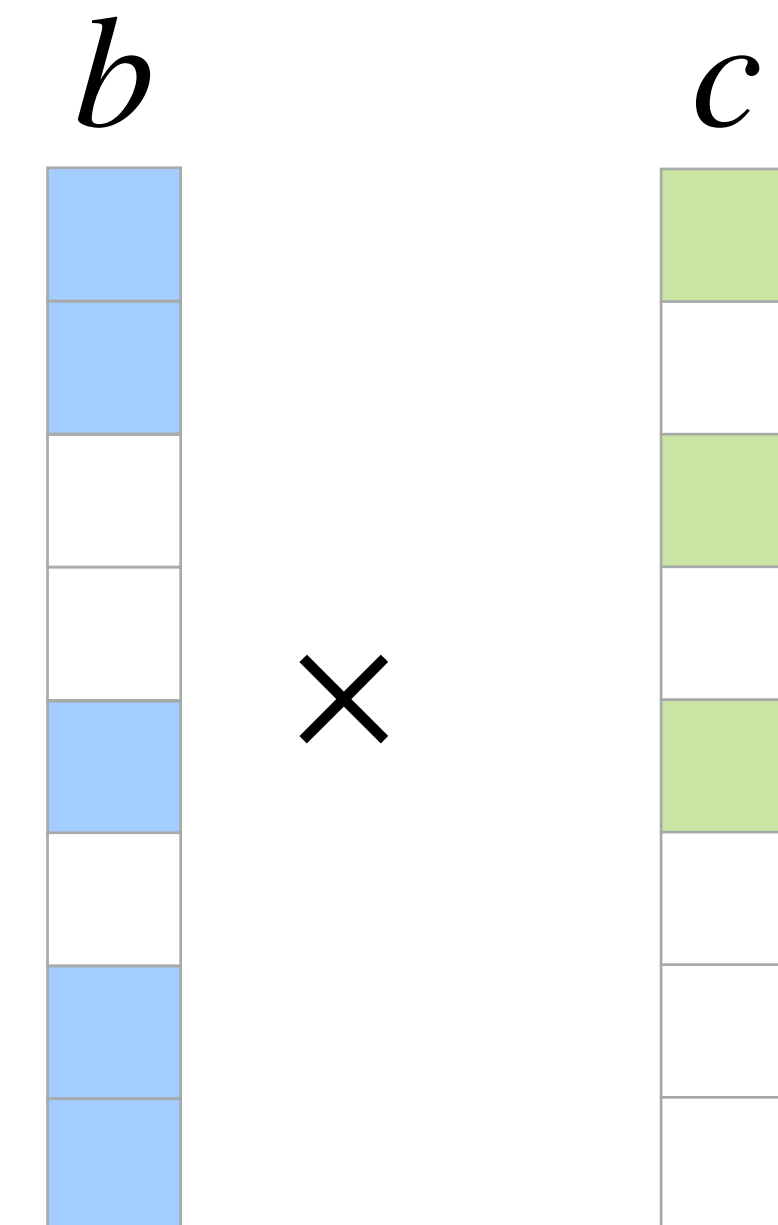
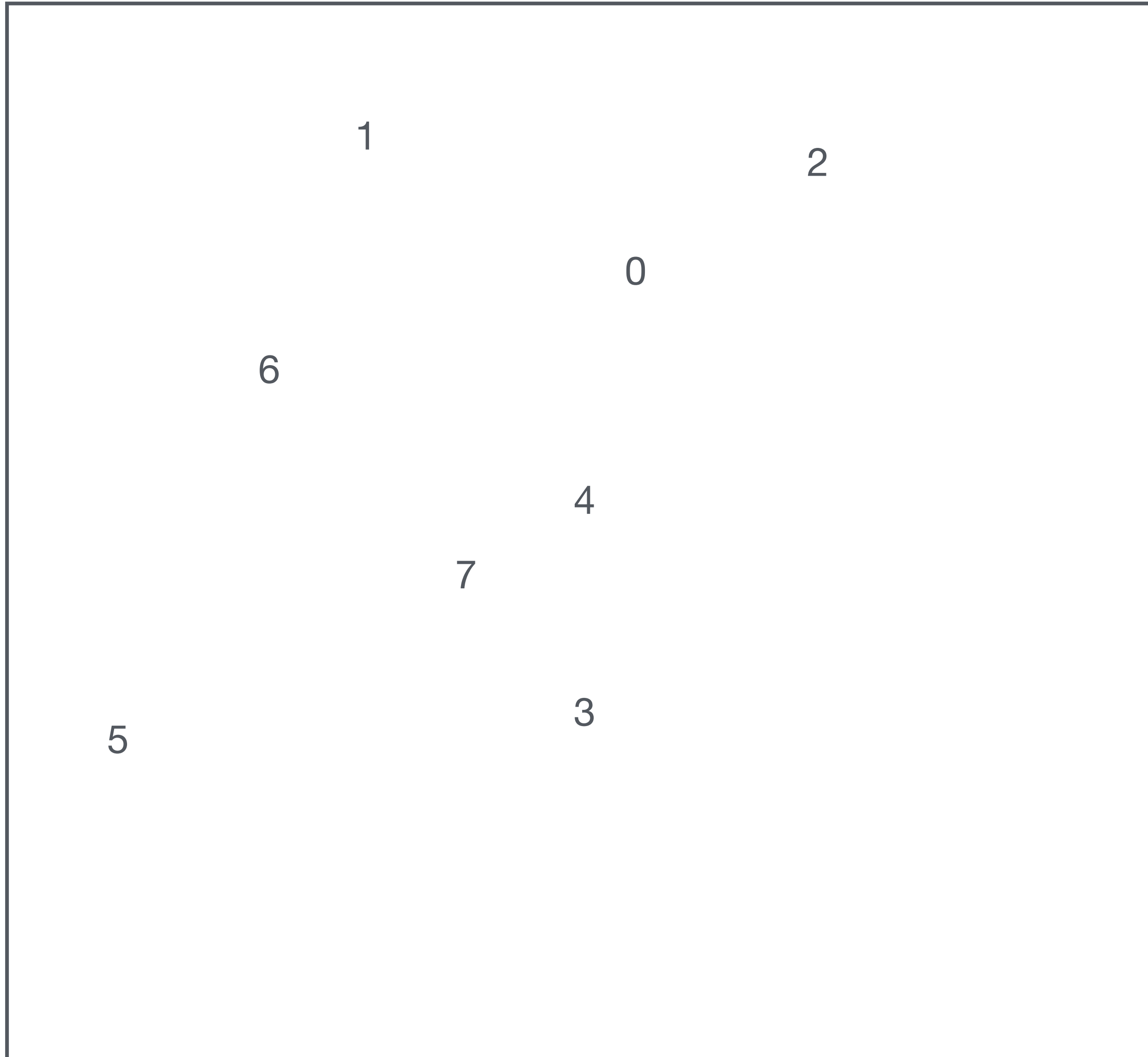
```
for (int i = 0; i < m; i++) {  
  for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {  
    int j = B2_crd[pB2];  
  
    int pA2 = i*n + j;  
    int pB3 = B3_pos[pB2];  
    int pc1 = c1_pos[0];  
    while (pB3 < B3_pos[pB2+1] && pc1 < c1_pos[1]) {  
      int kB = B3_crd[pB3];  
      int kc = c1_crd[pc1];  
      int k = min(kB, kc);  
      if (kB == k && kc == k) {  
        A[pA2] += B[pB3] * c[pc1];  
      }  
      if (kB == k) pB3++;  
      if (kc == k) pc1++;  
    }  
  }  
}
```

Key operation is to coiterate
over data structures

Intersection coiteration

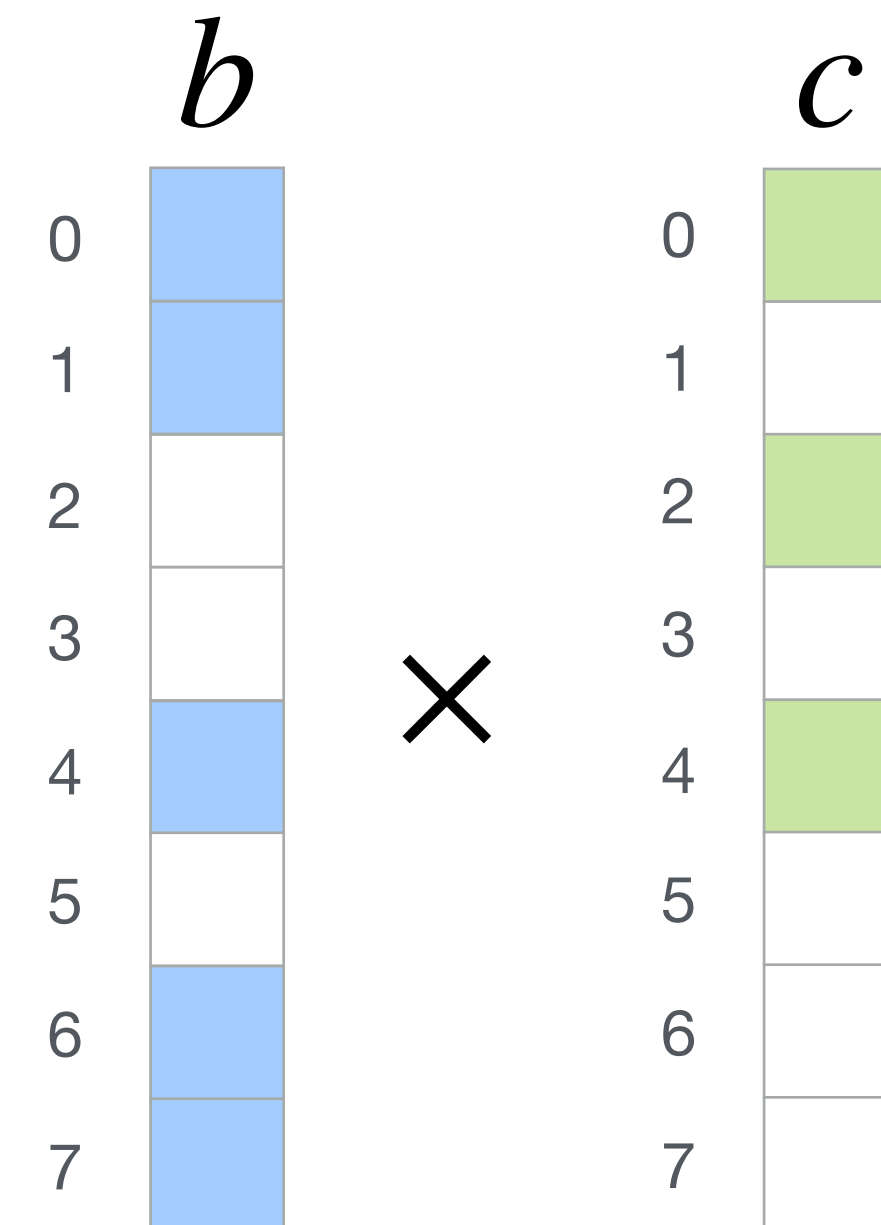
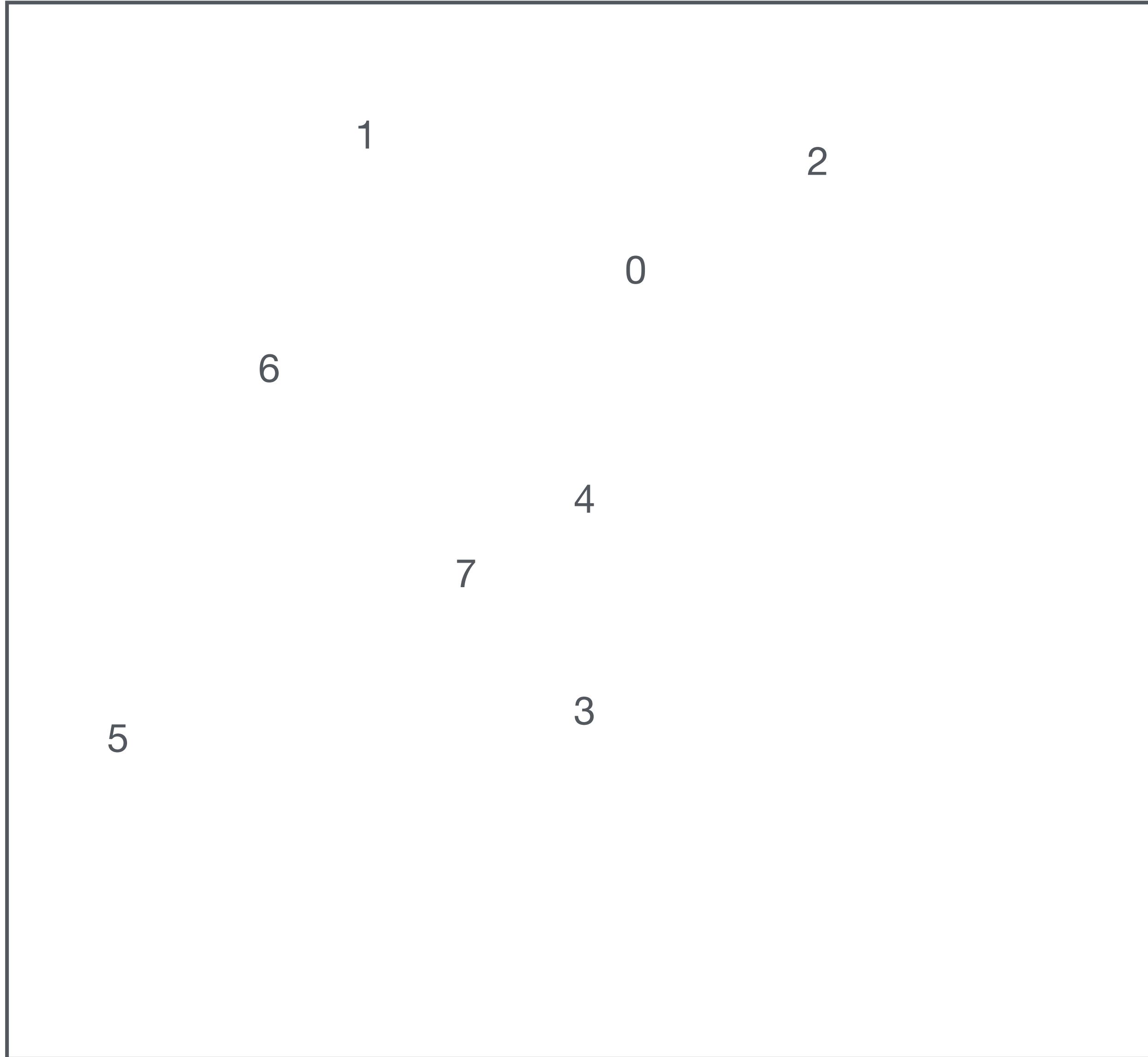
Data structure coiteration

Coordinate Space



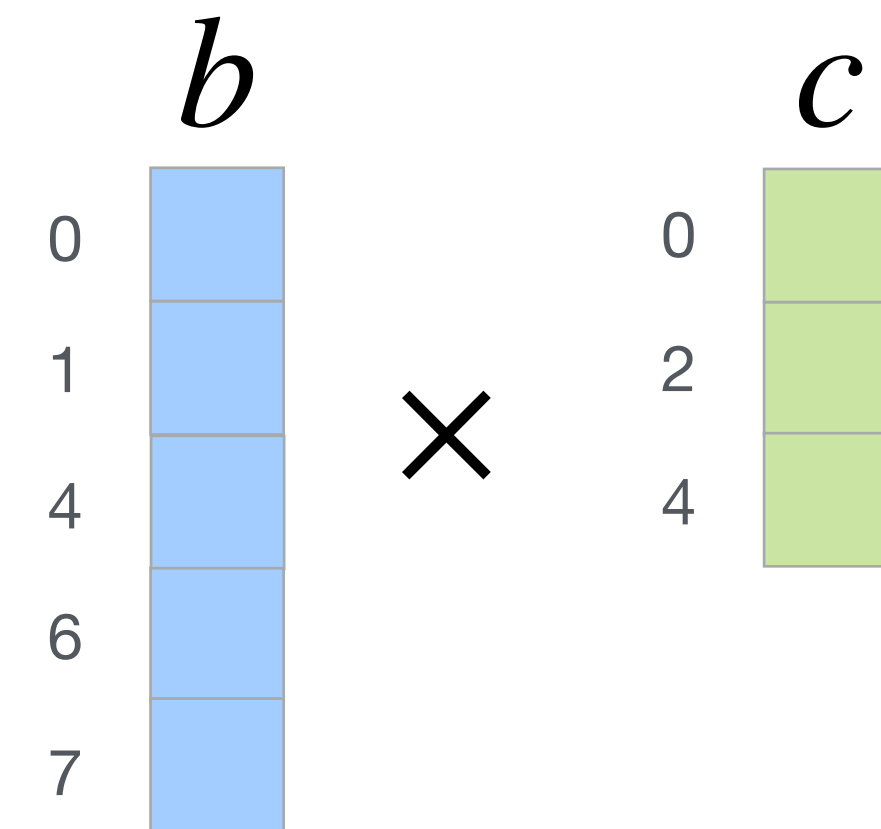
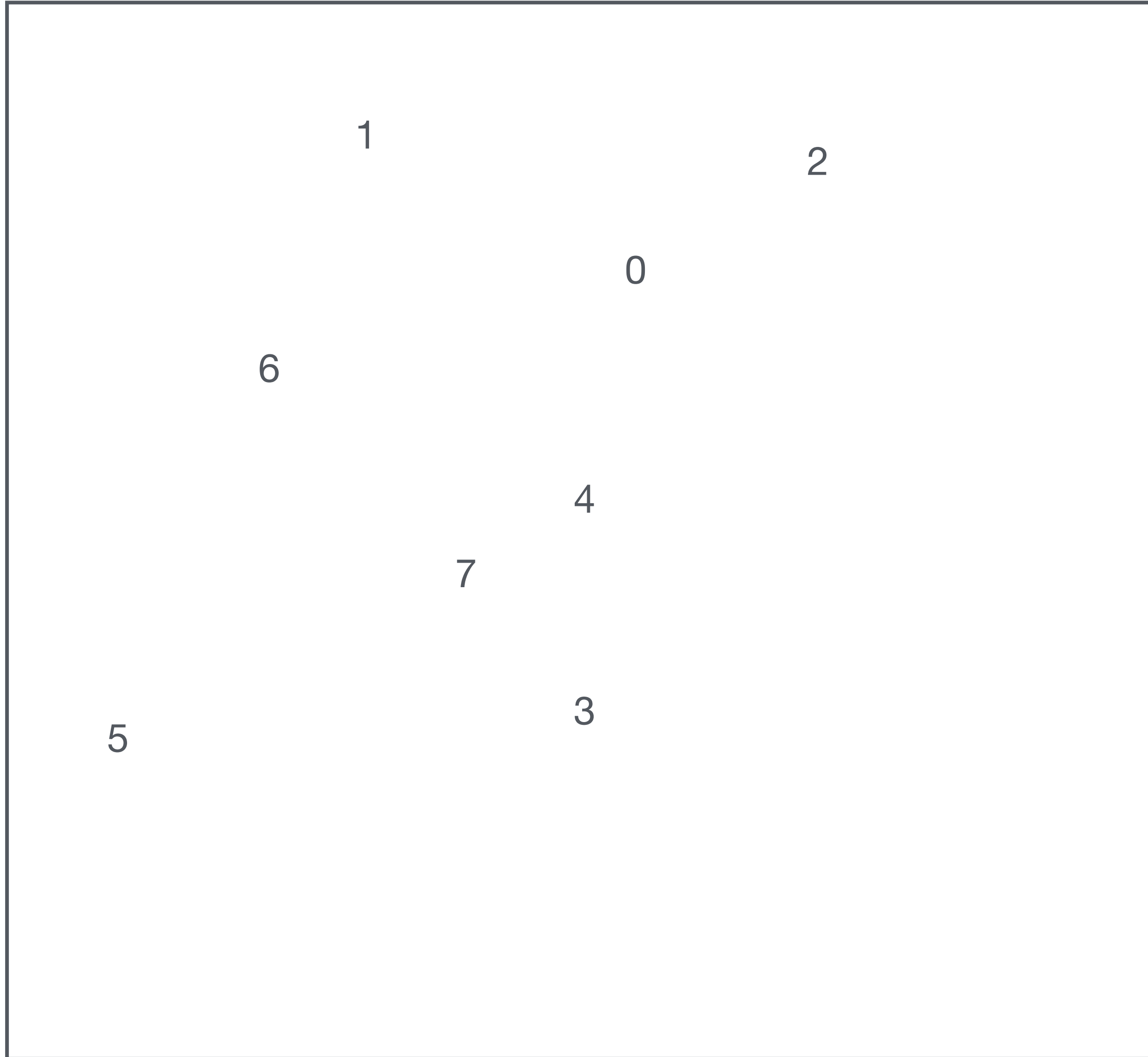
Data structure coiteration

Coordinate Space



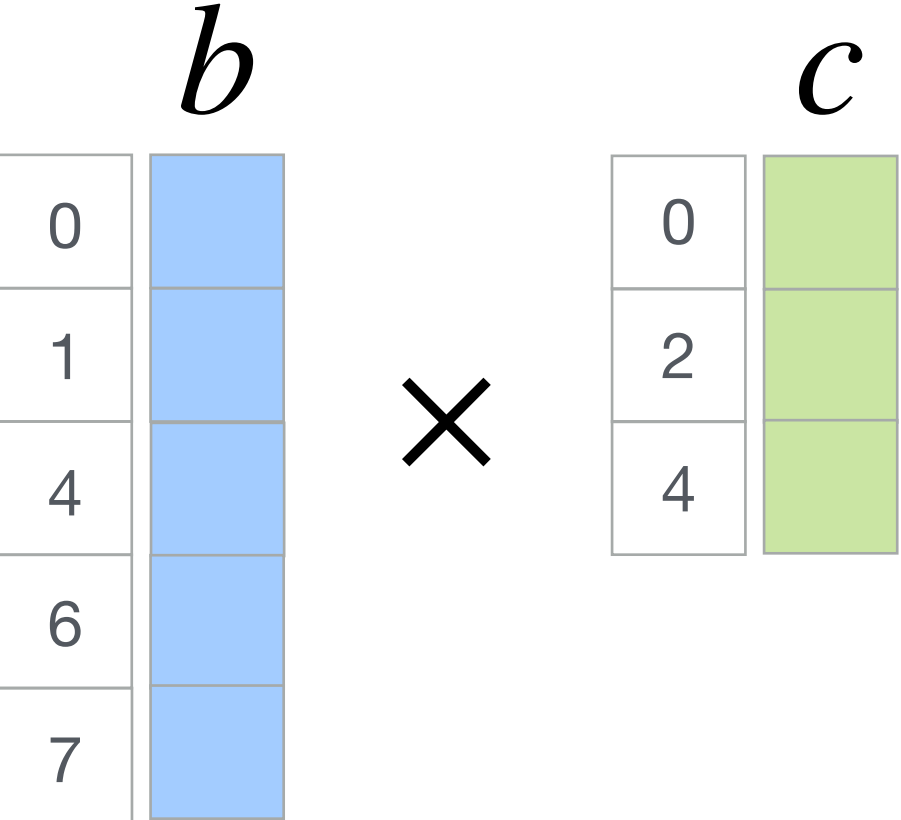
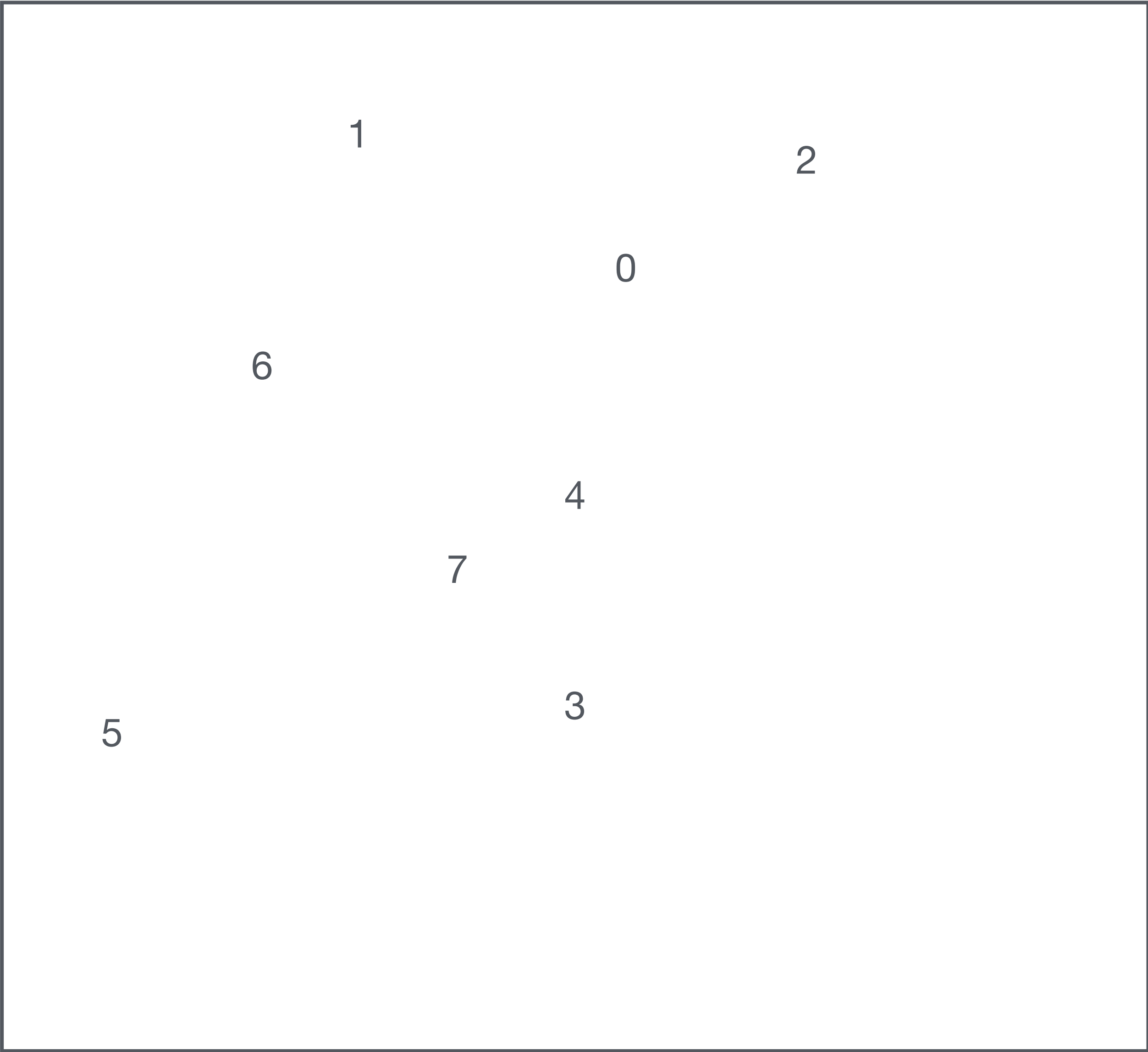
Data structure coiteration

Coordinate Space

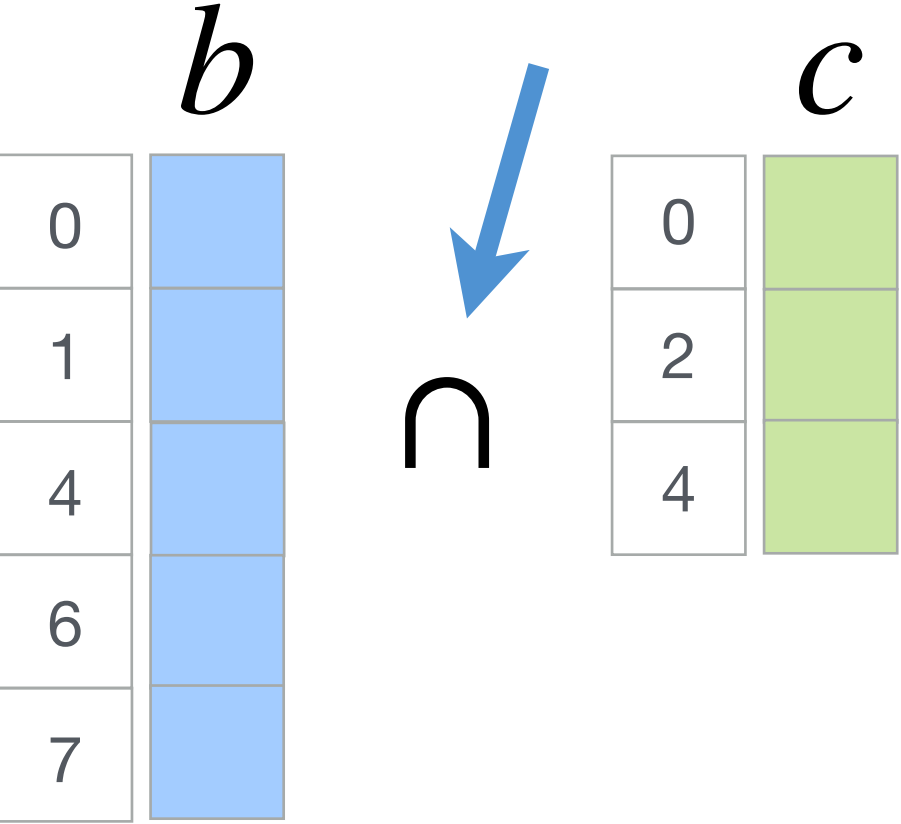
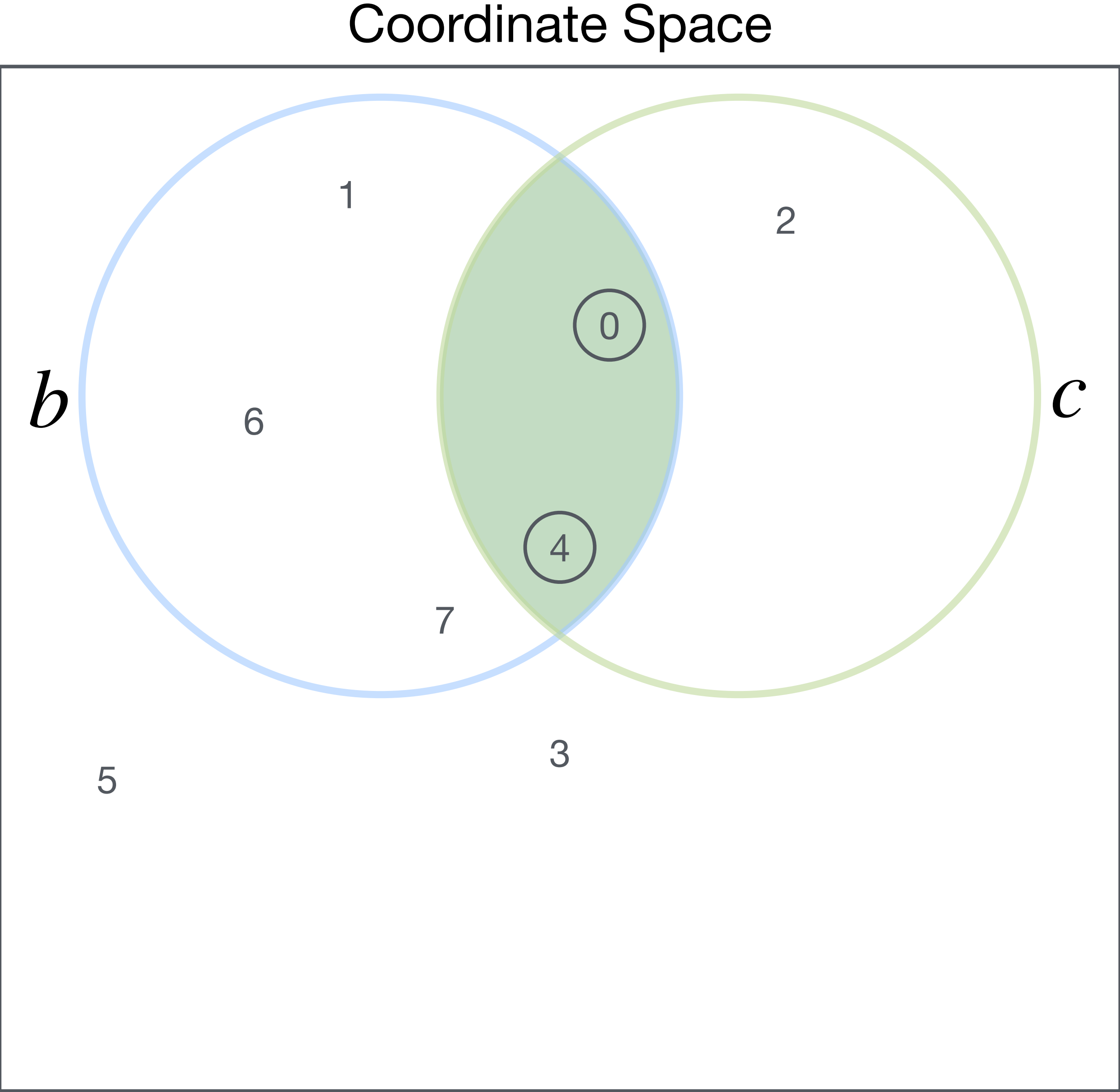


Data structure coiteration

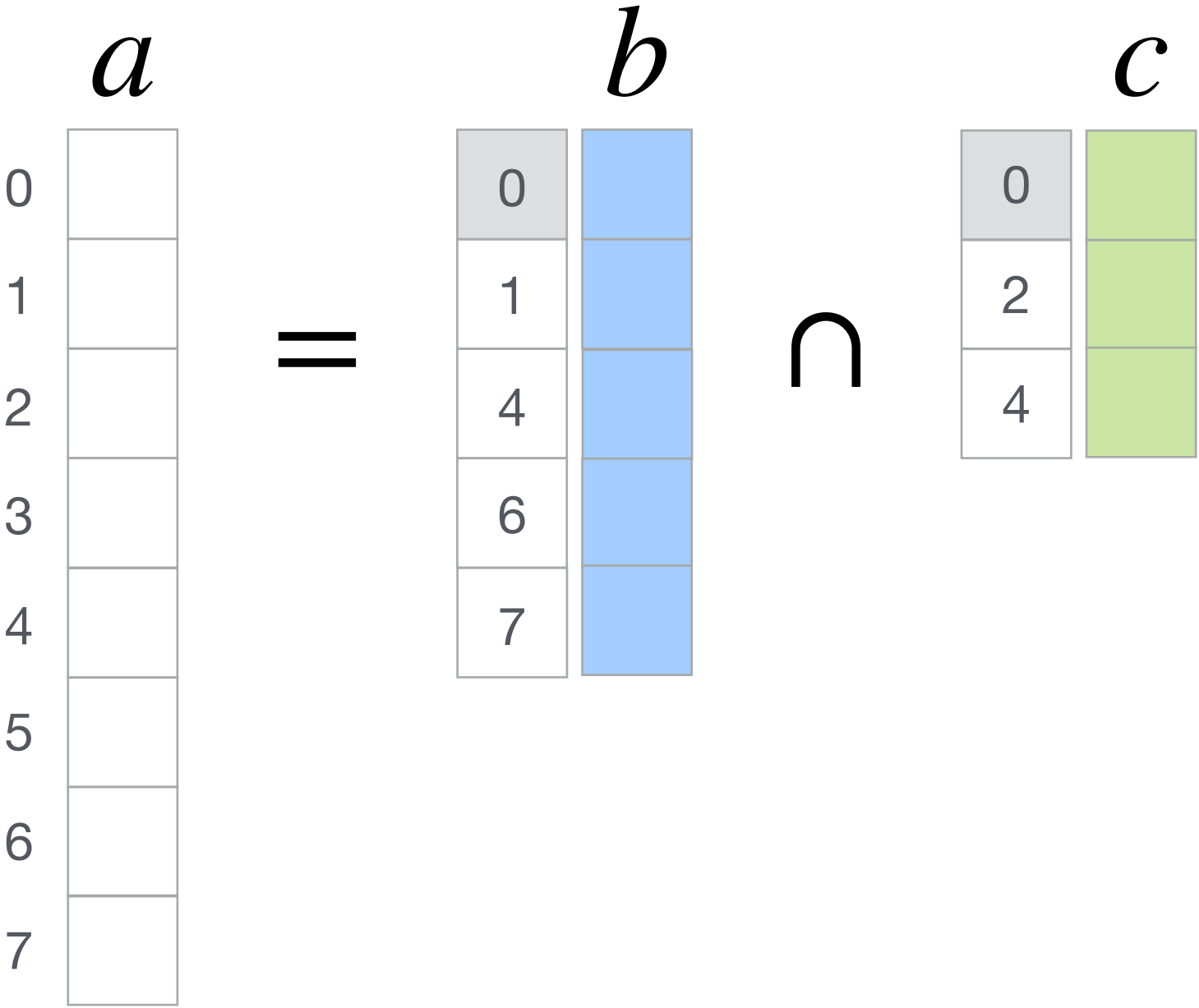
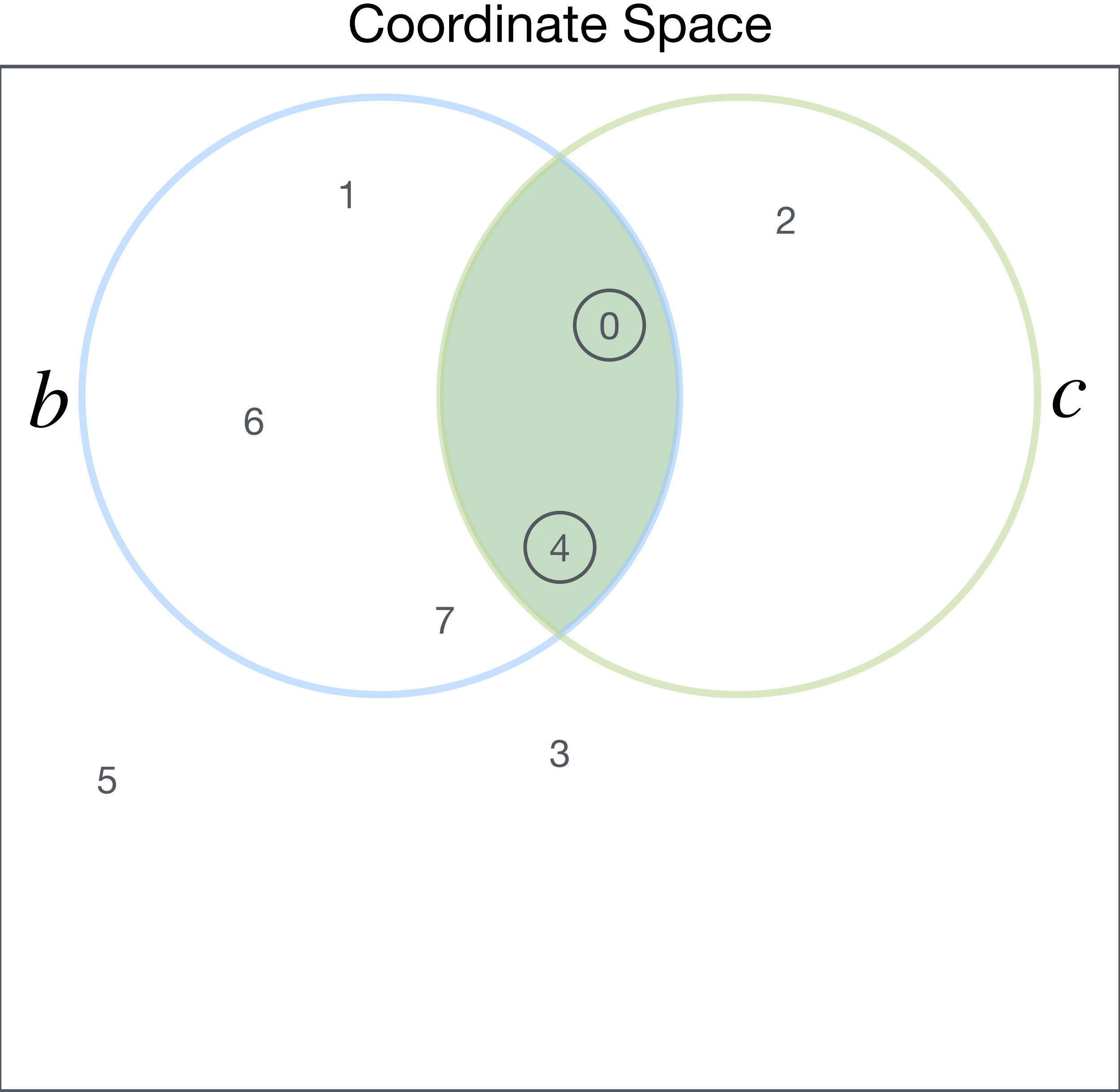
Coordinate Space



Data structure coiteration

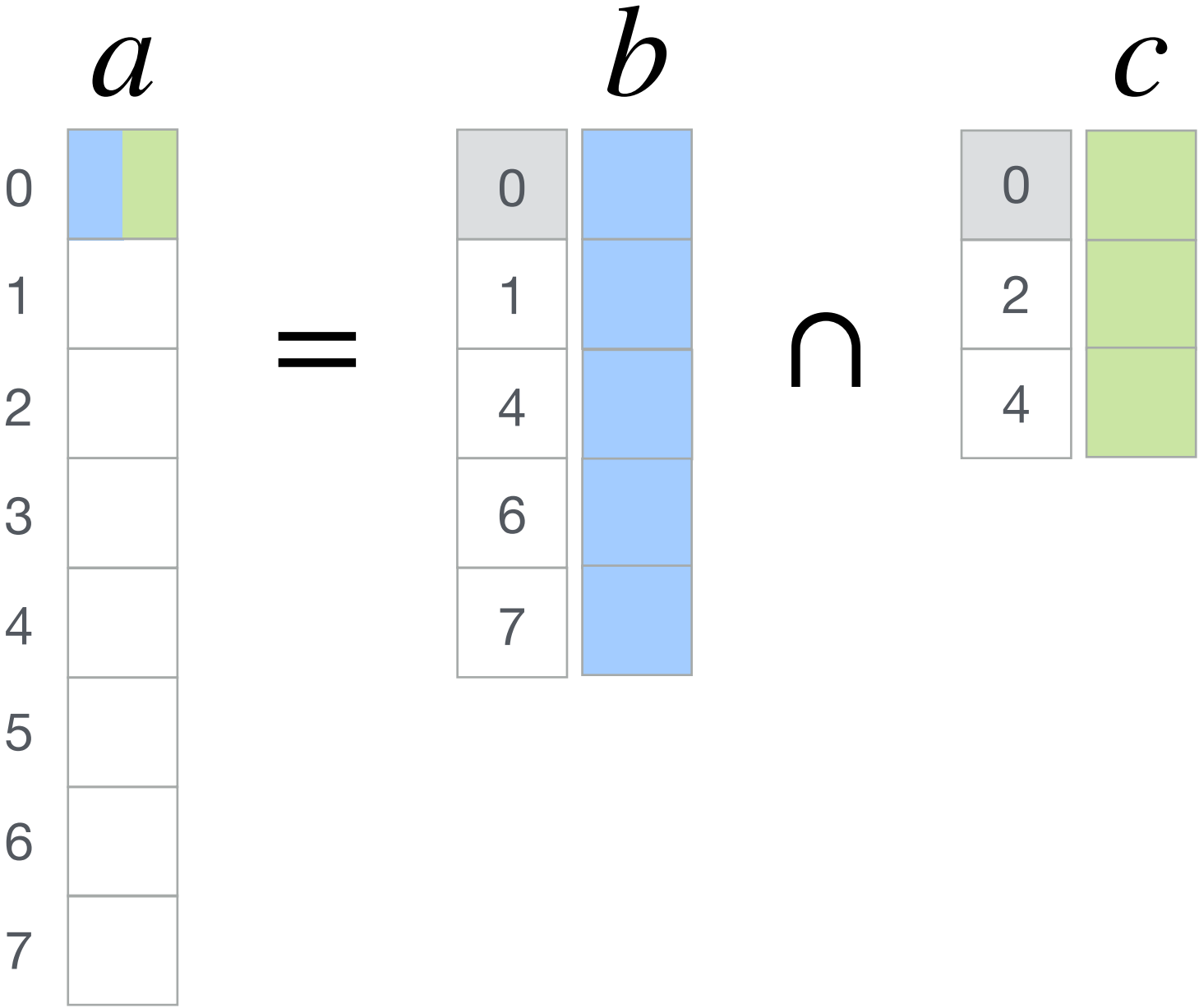
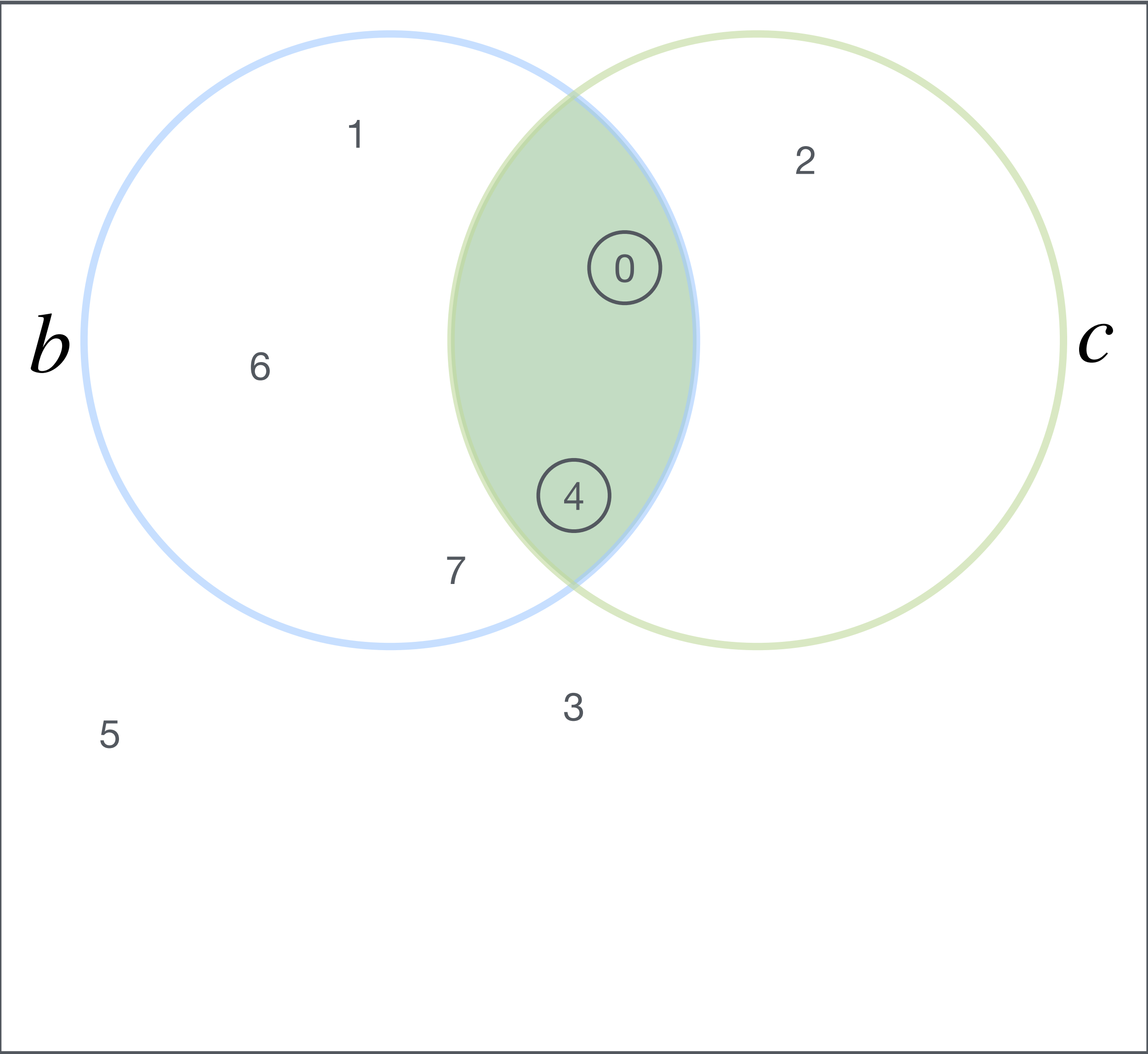


Data structure coiteration

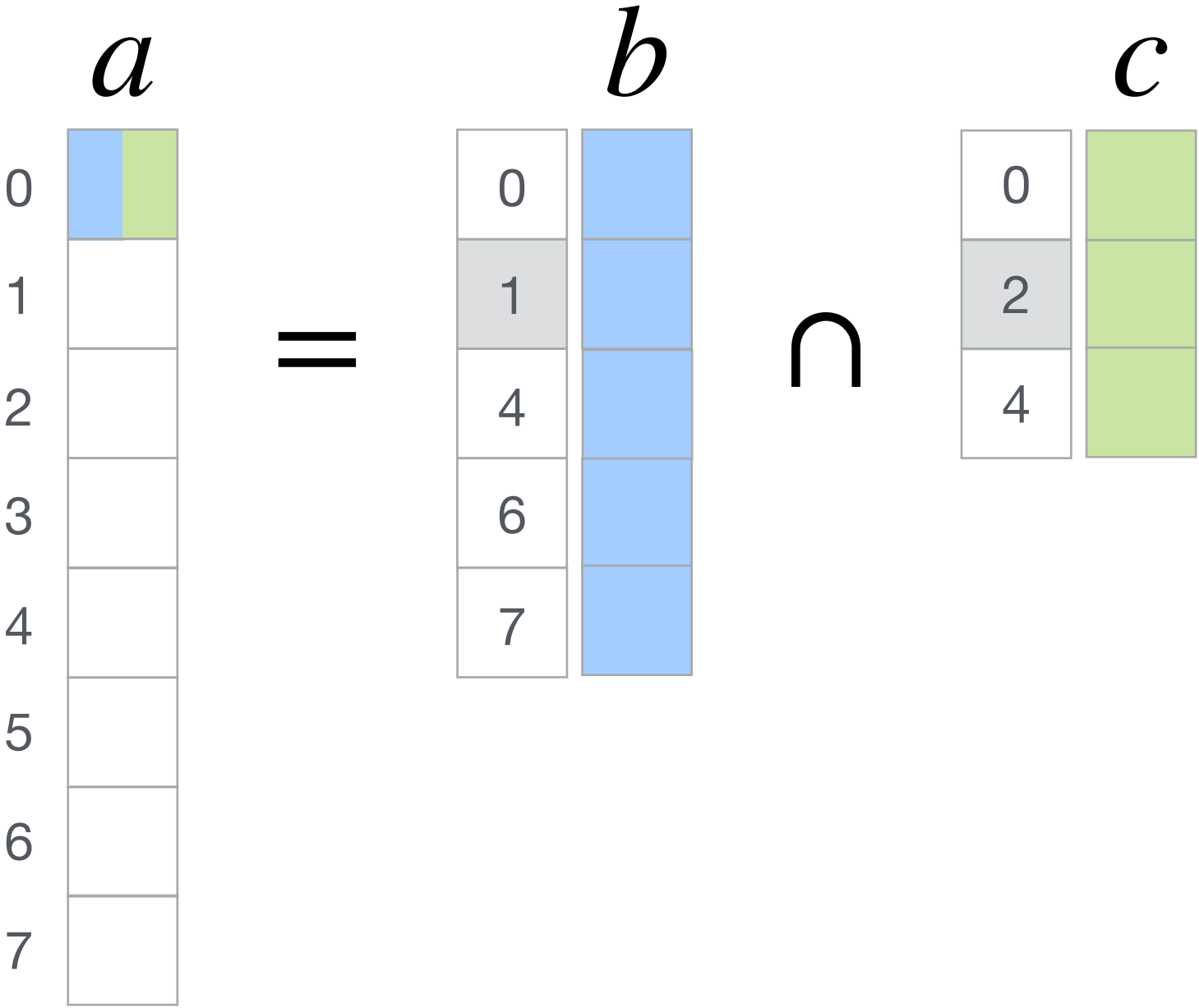
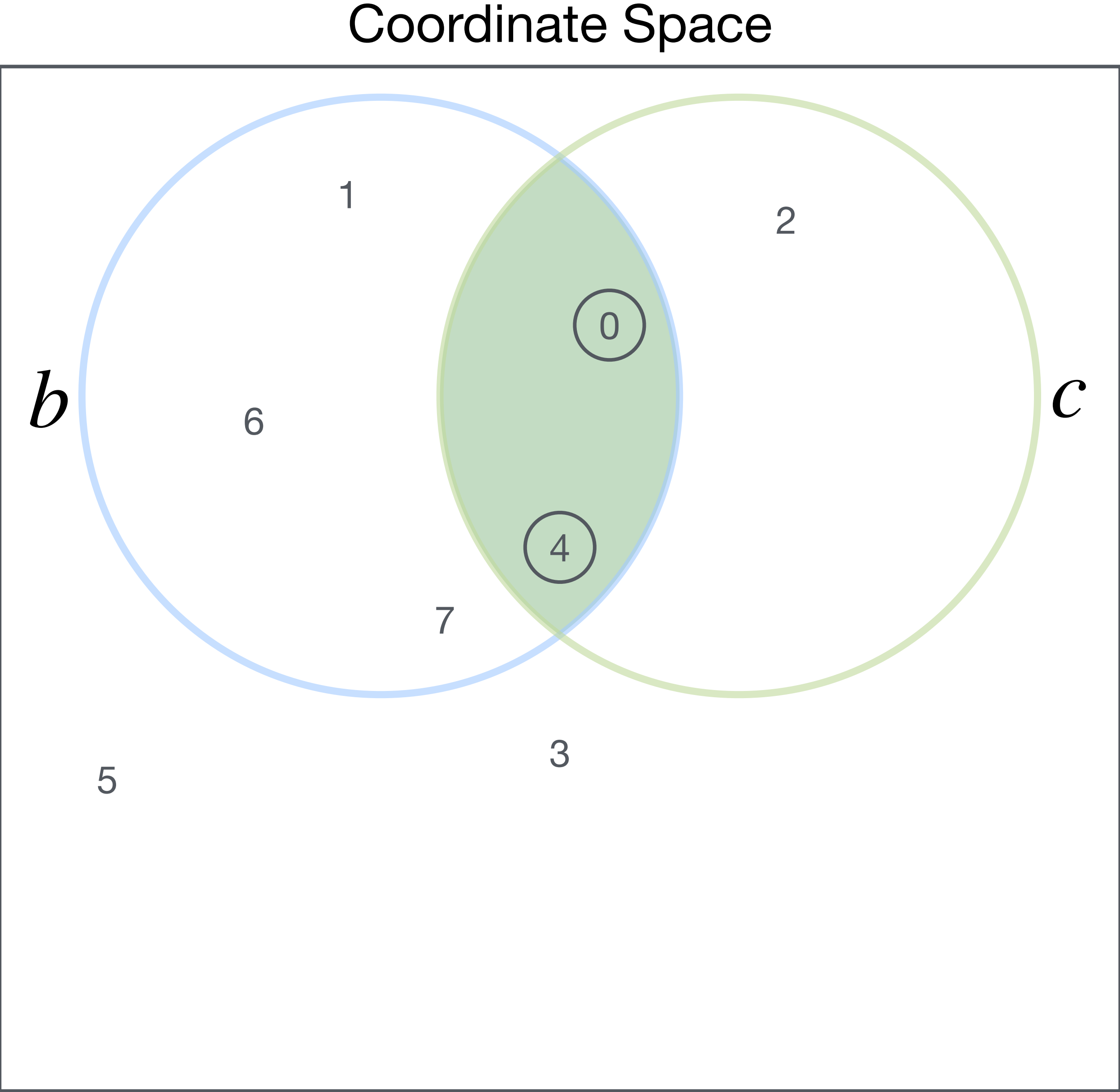


Data structure coiteration

Coordinate Space

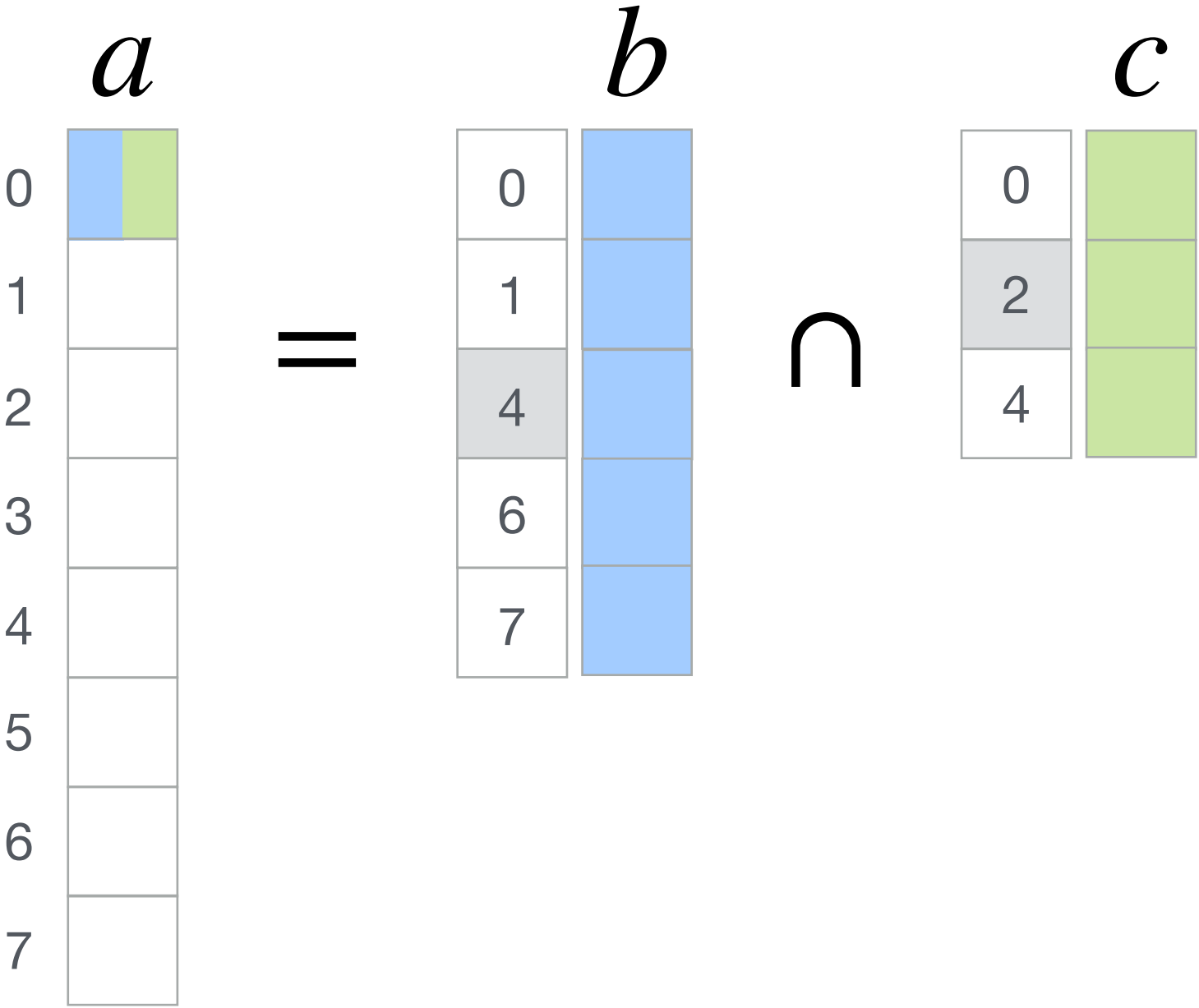
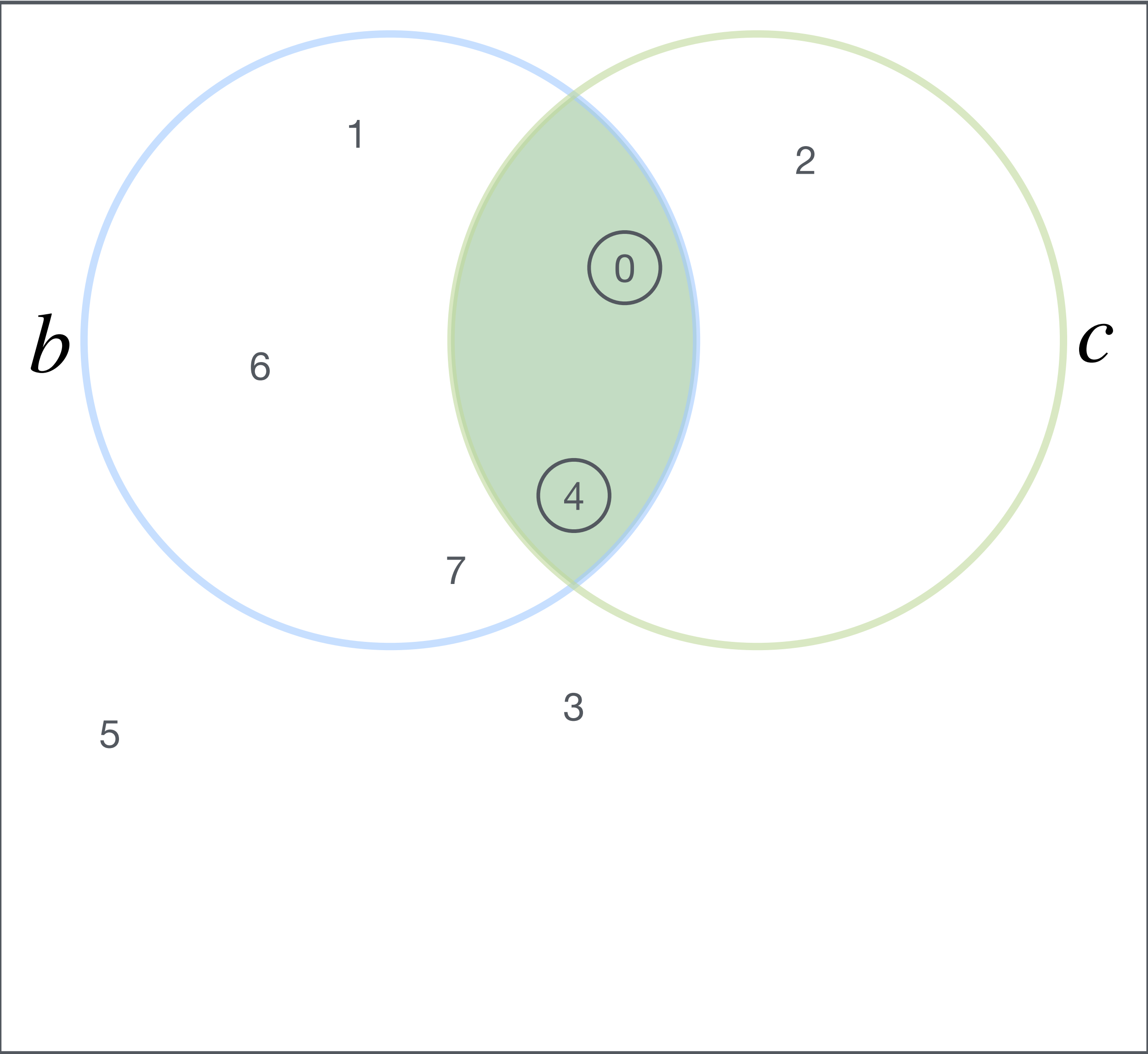


Data structure coiteration

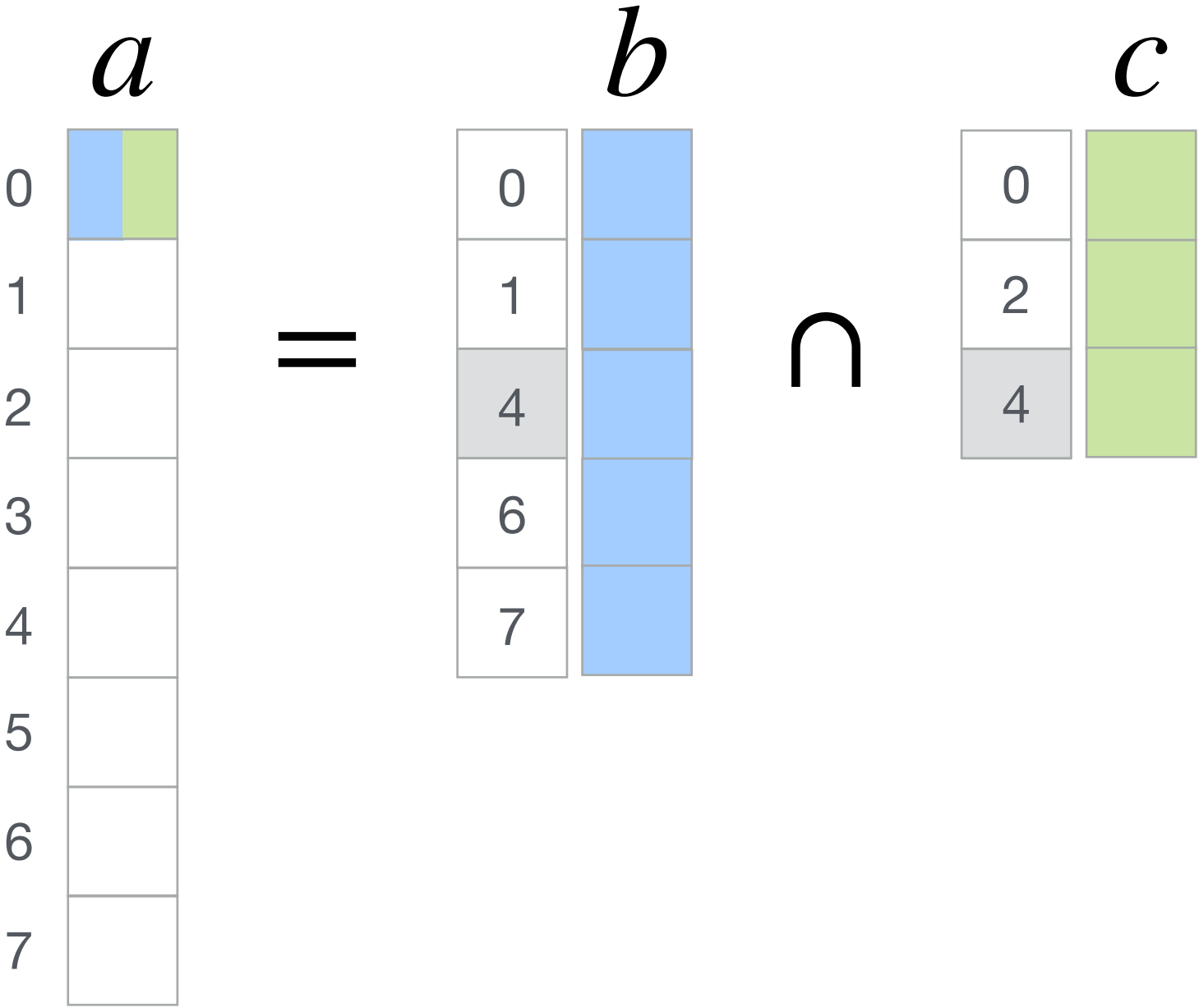
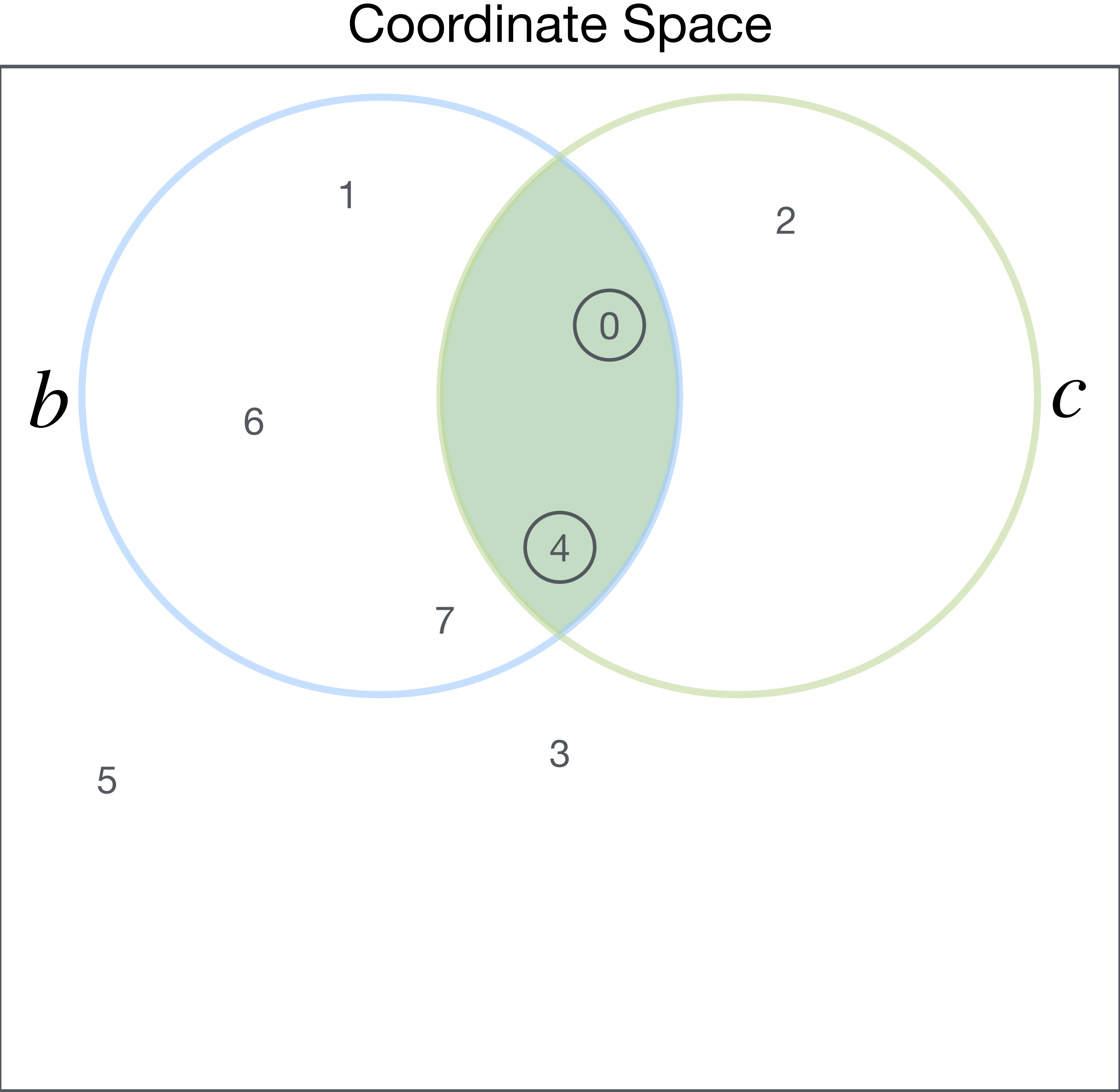


Data structure coiteration

Coordinate Space

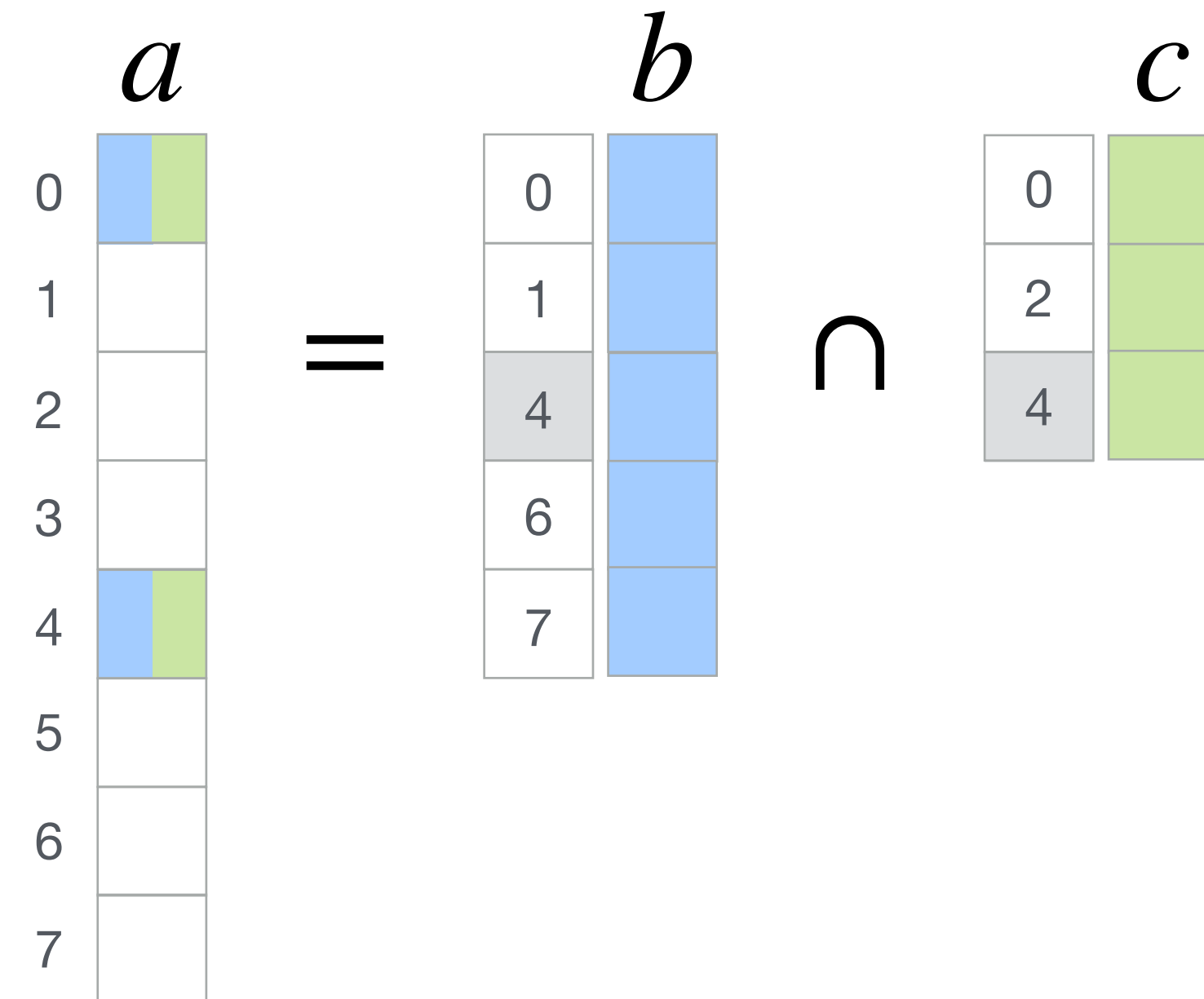
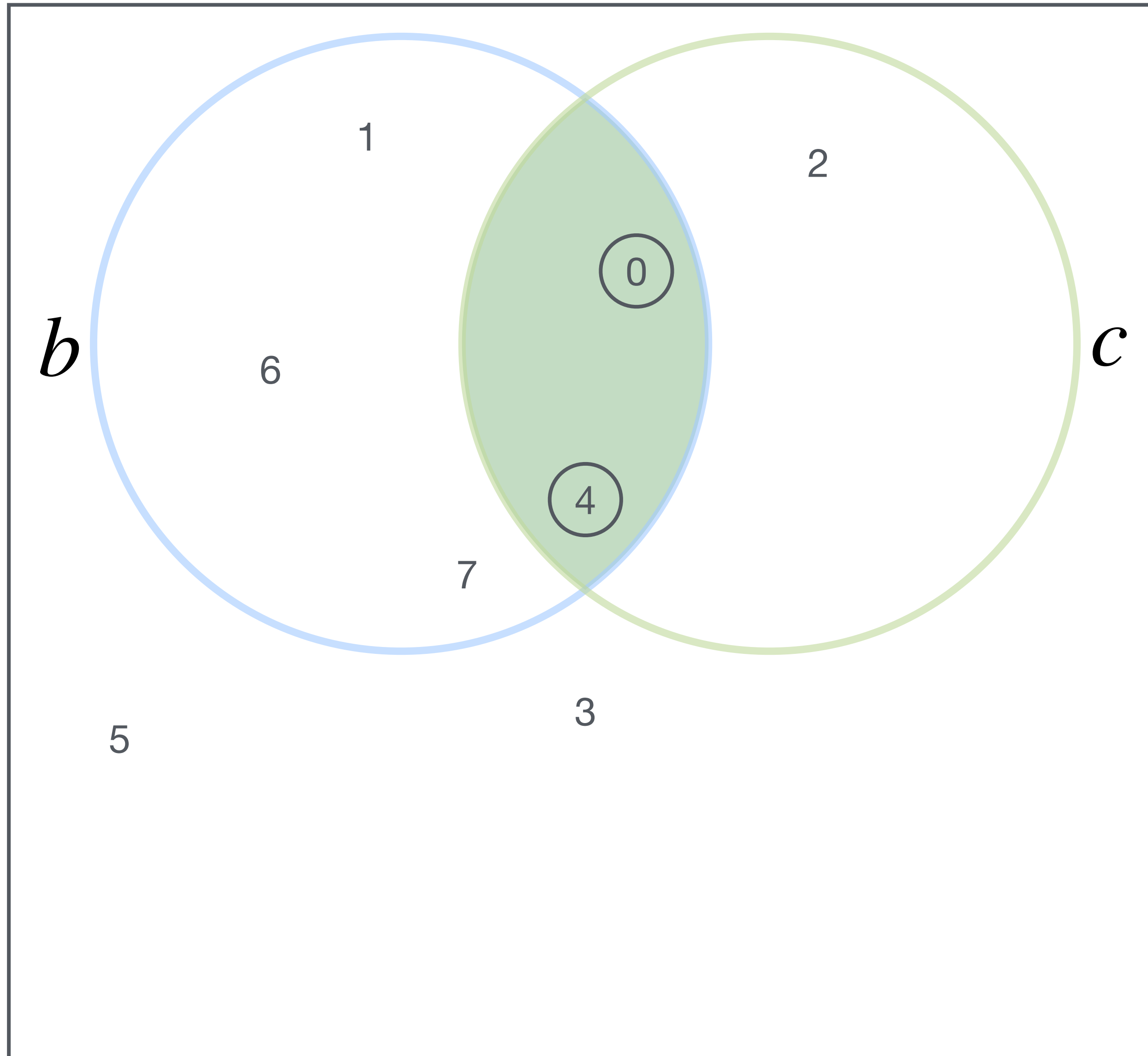


Data structure coiteration

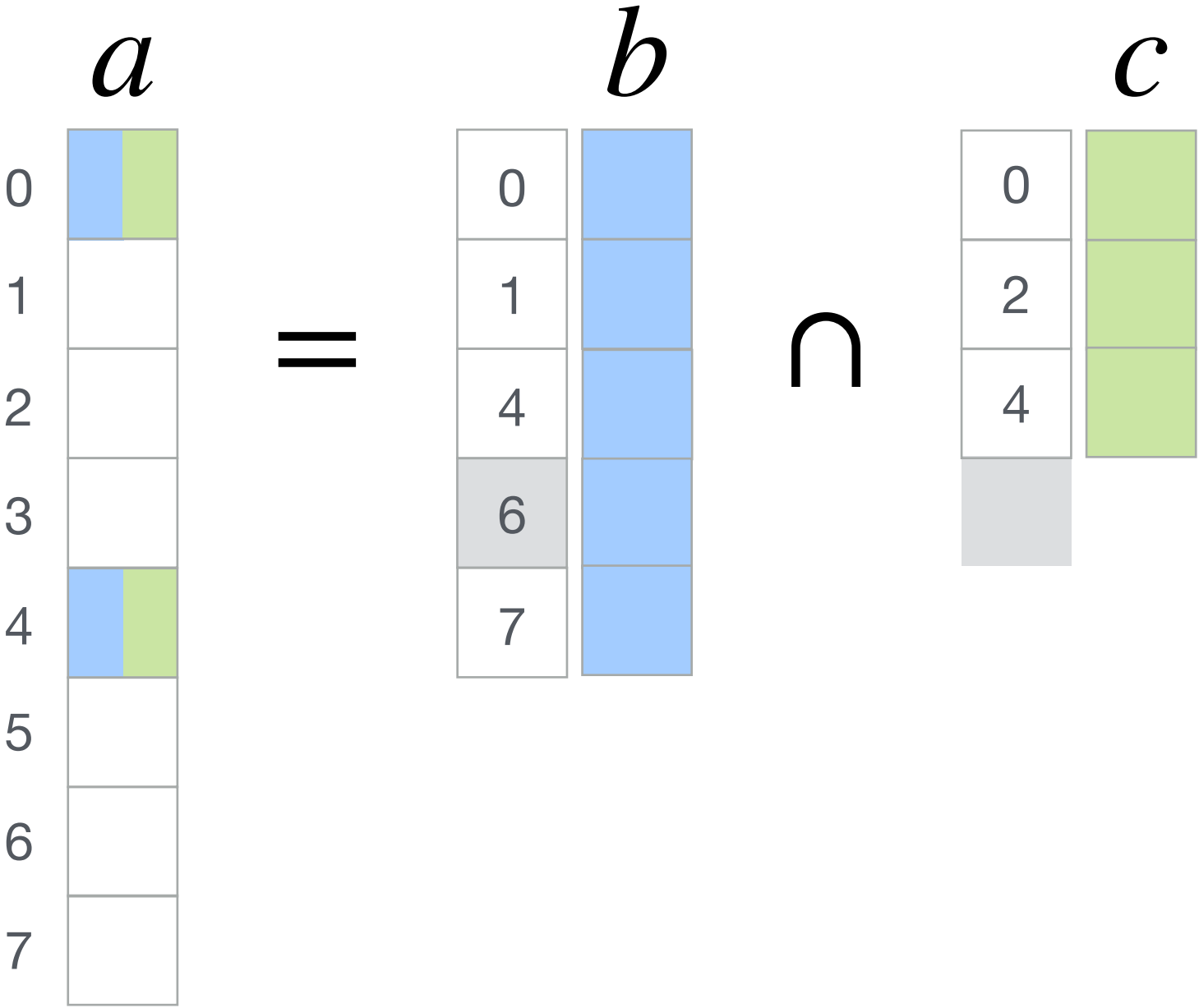
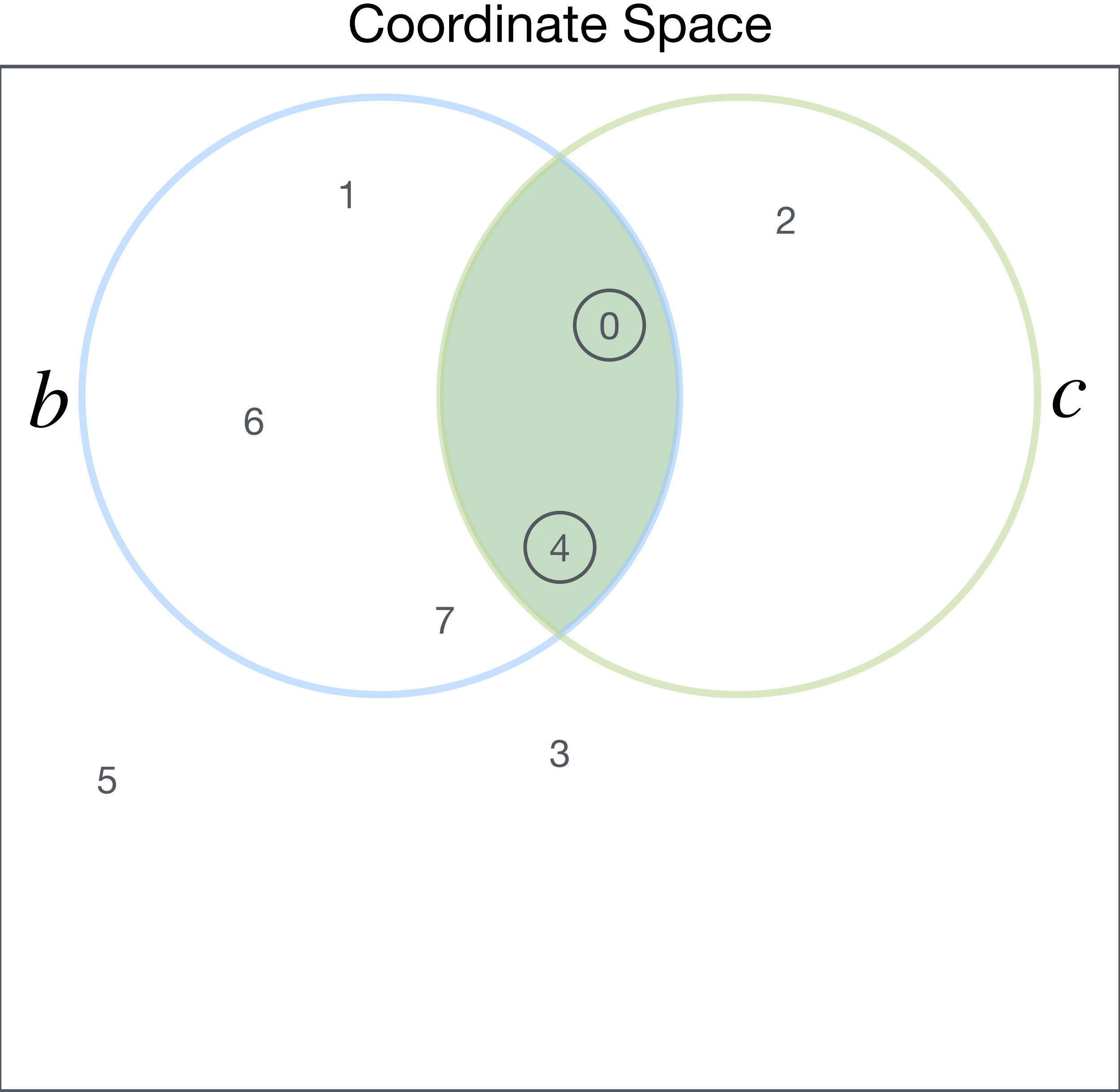


Data structure coiteration

Coordinate Space

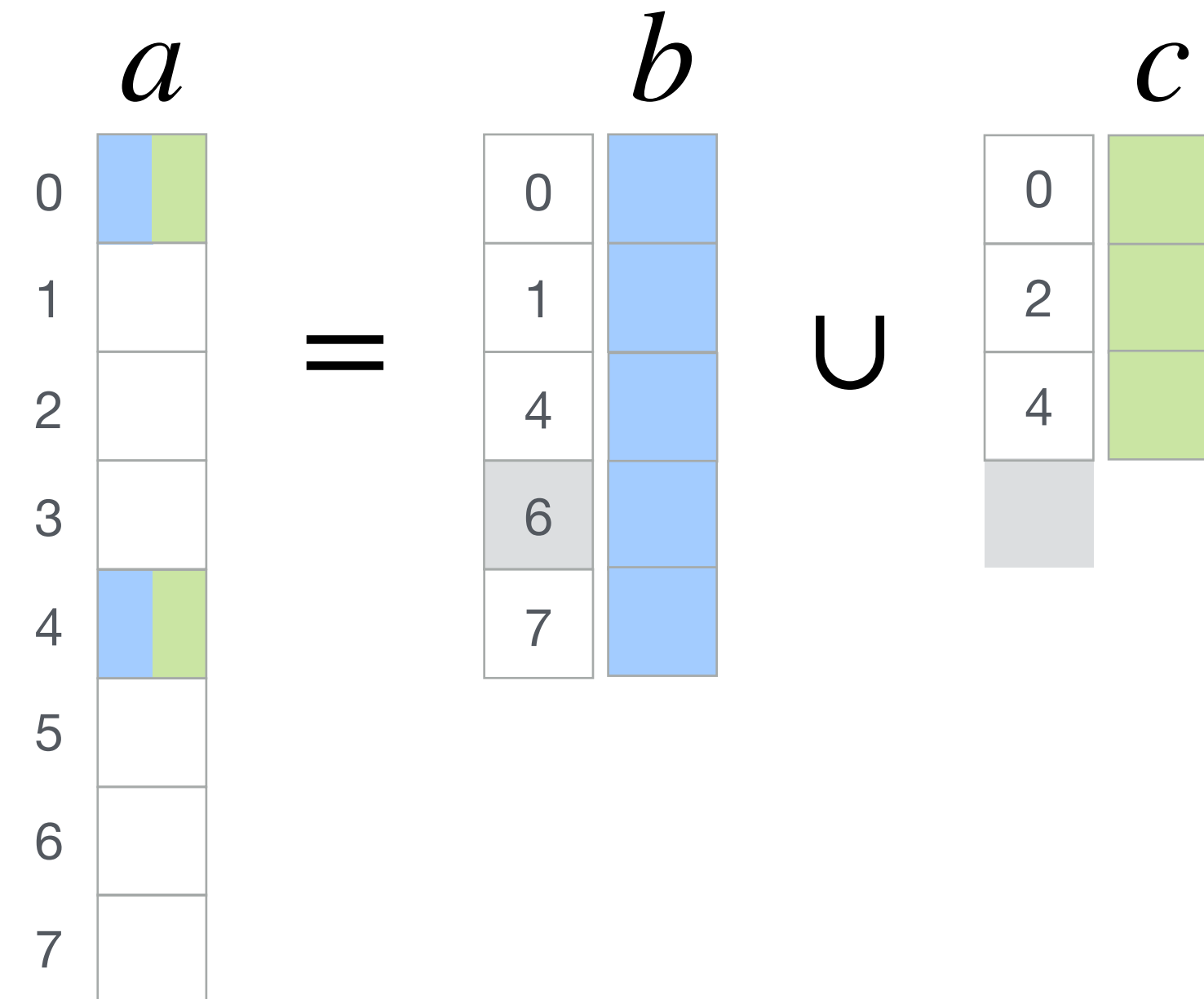
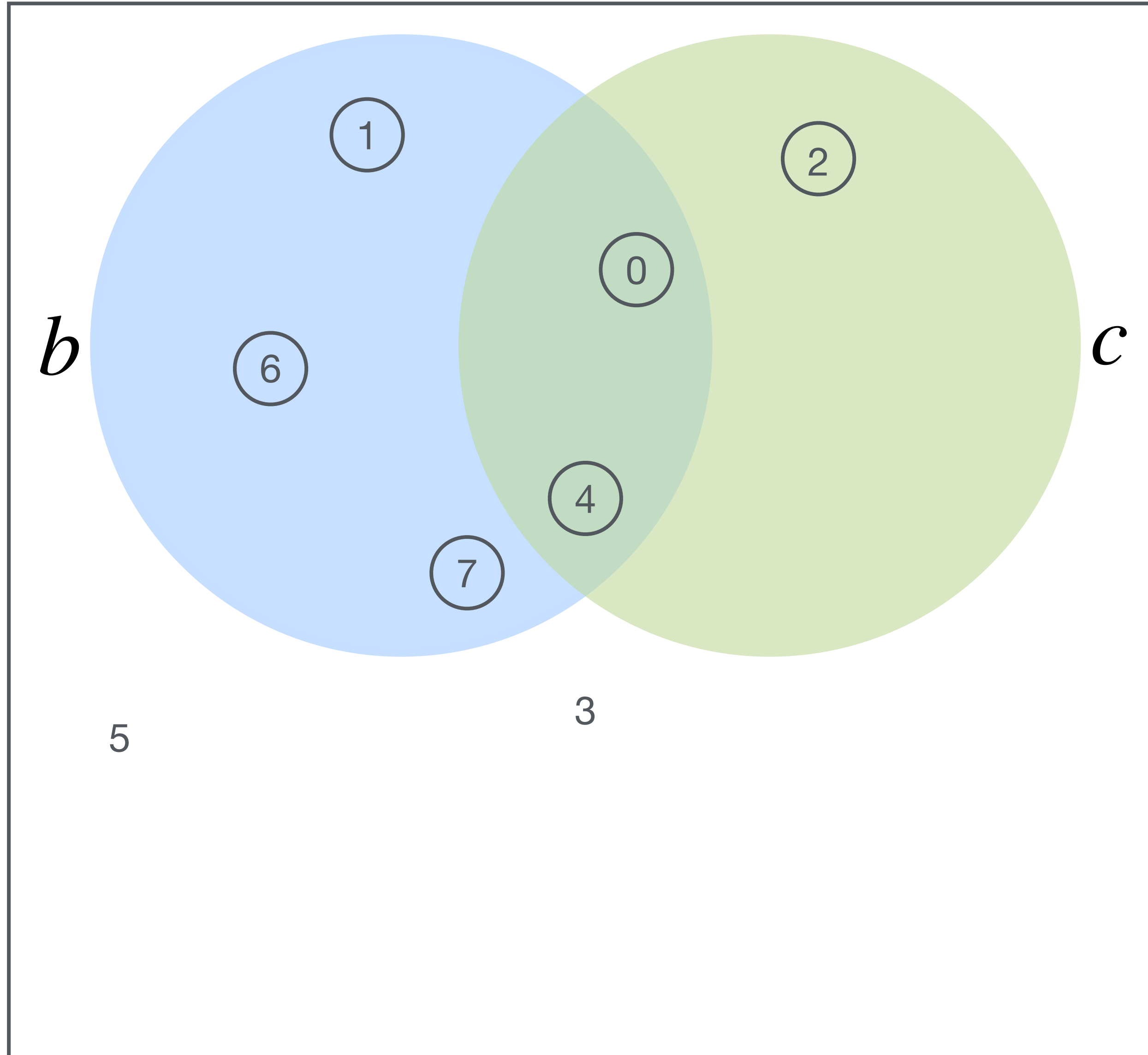


Data structure coiteration

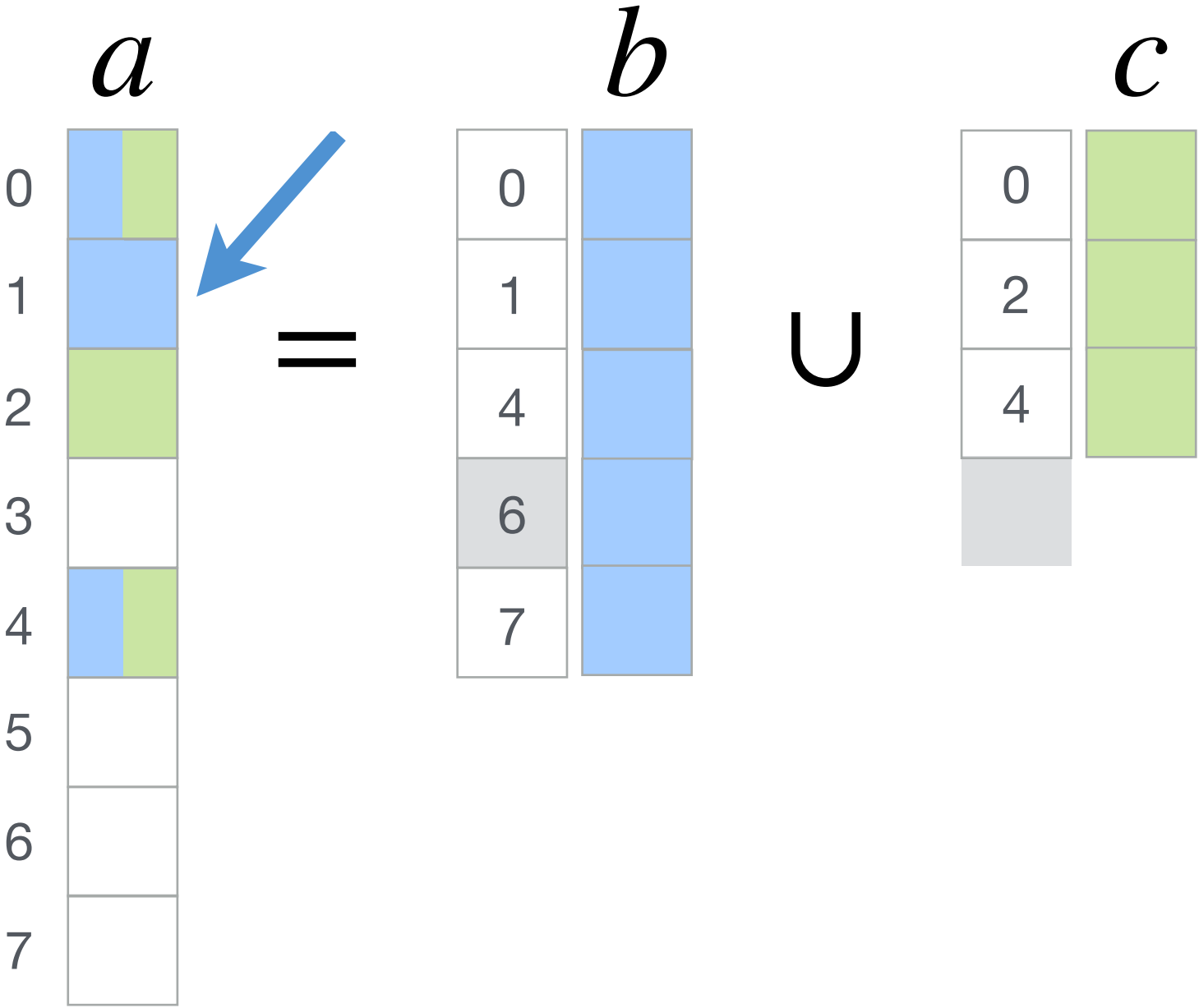
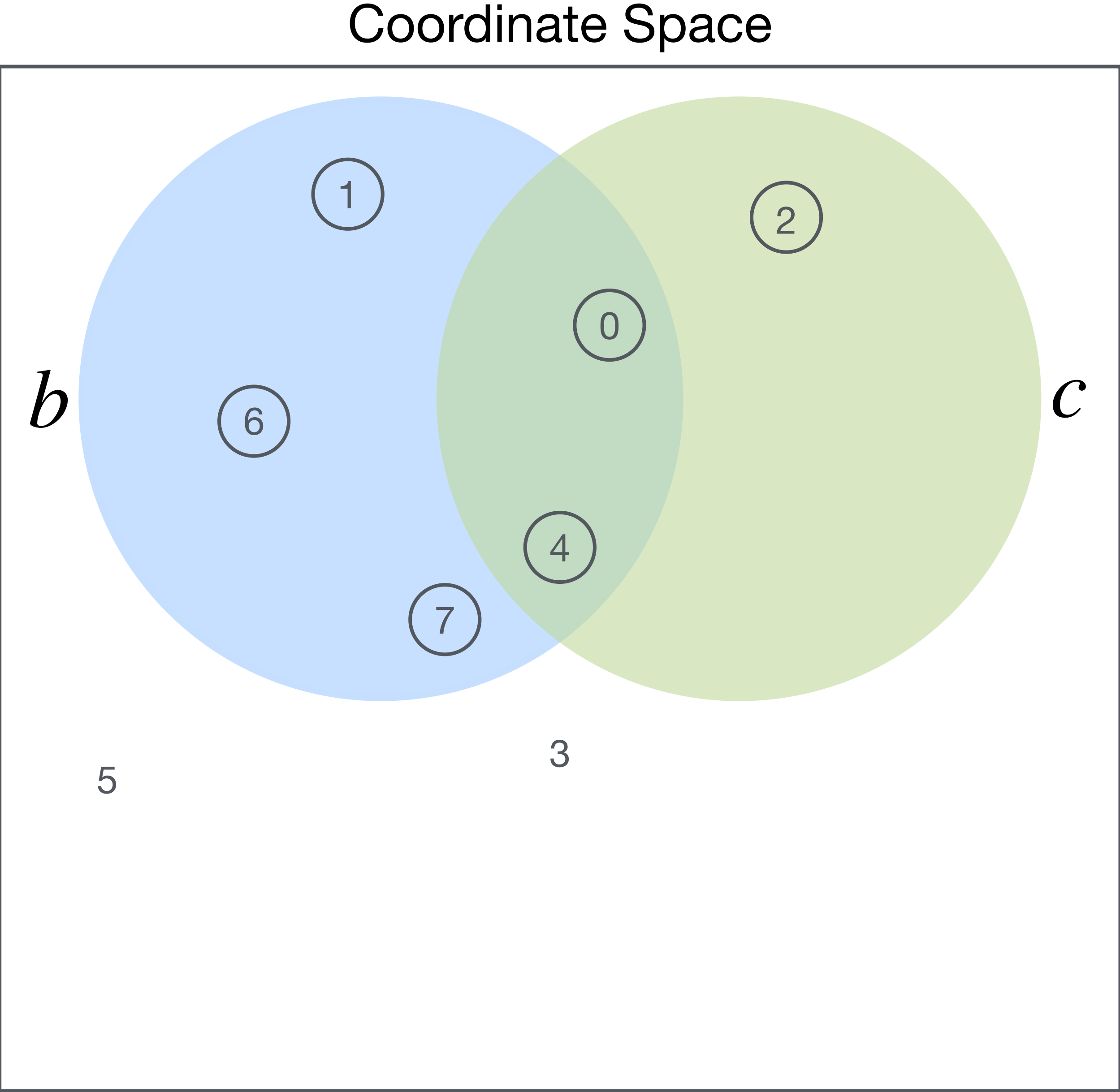


Data structure coiteration

Coordinate Space

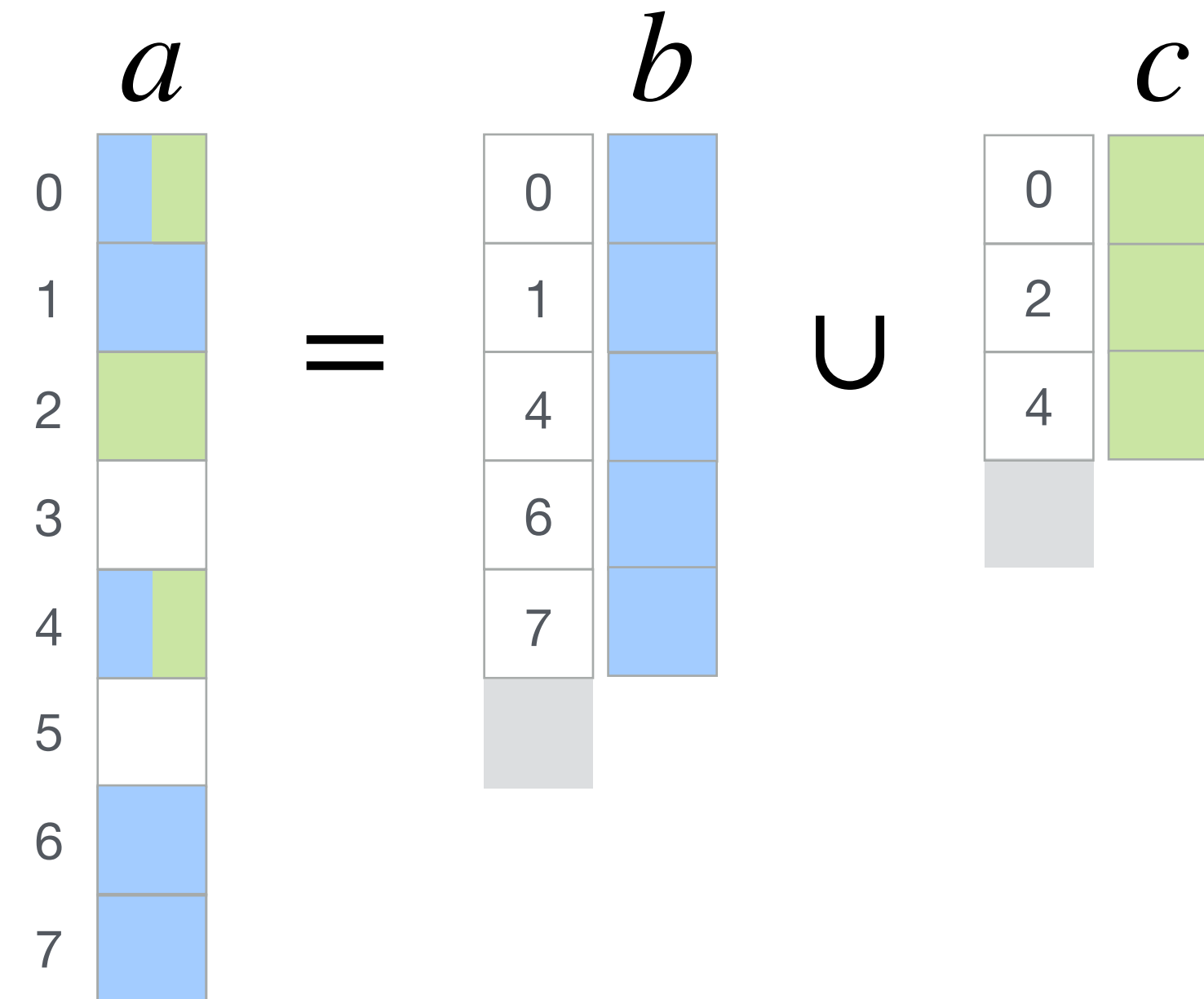
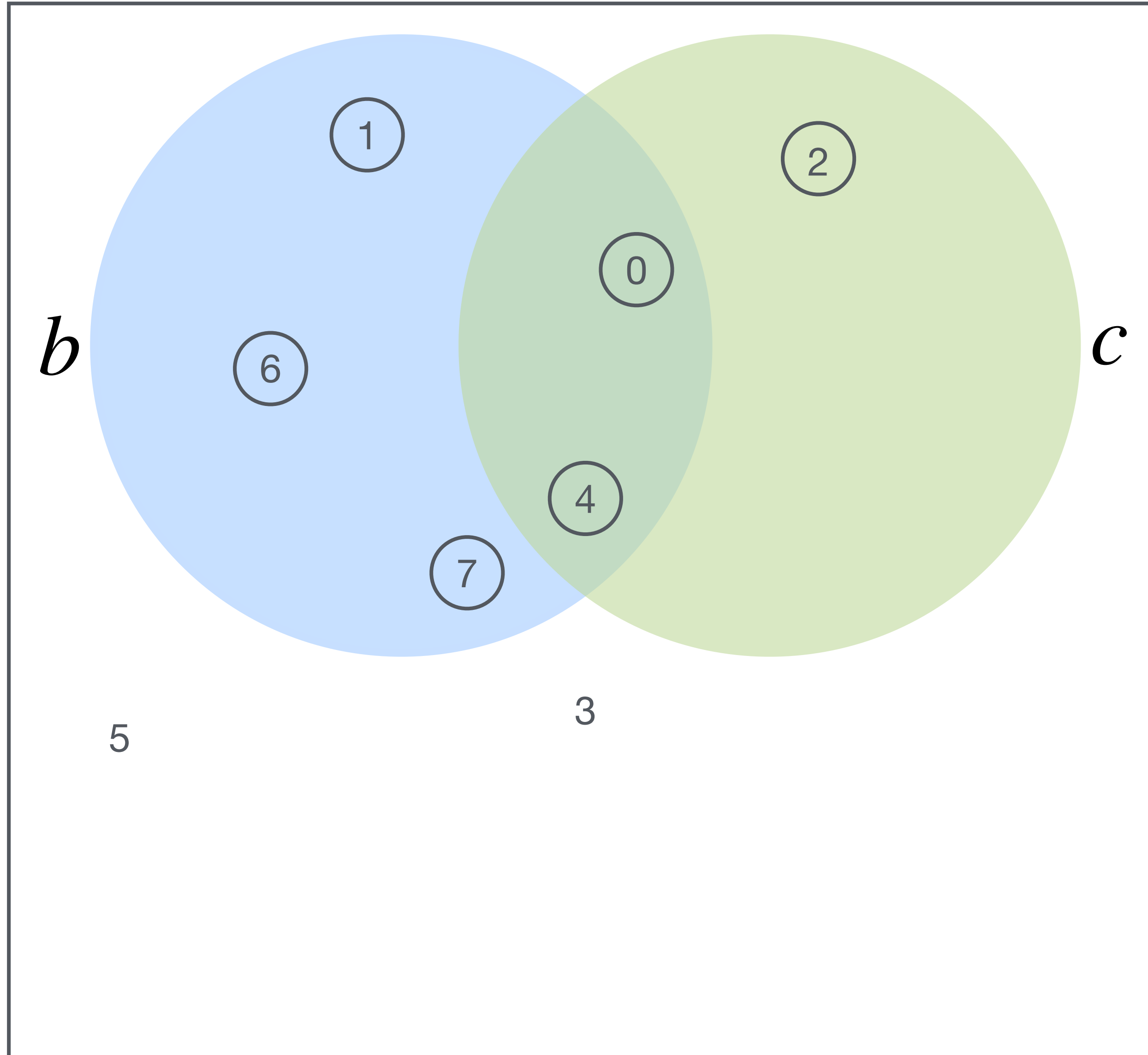


Data structure coiteration



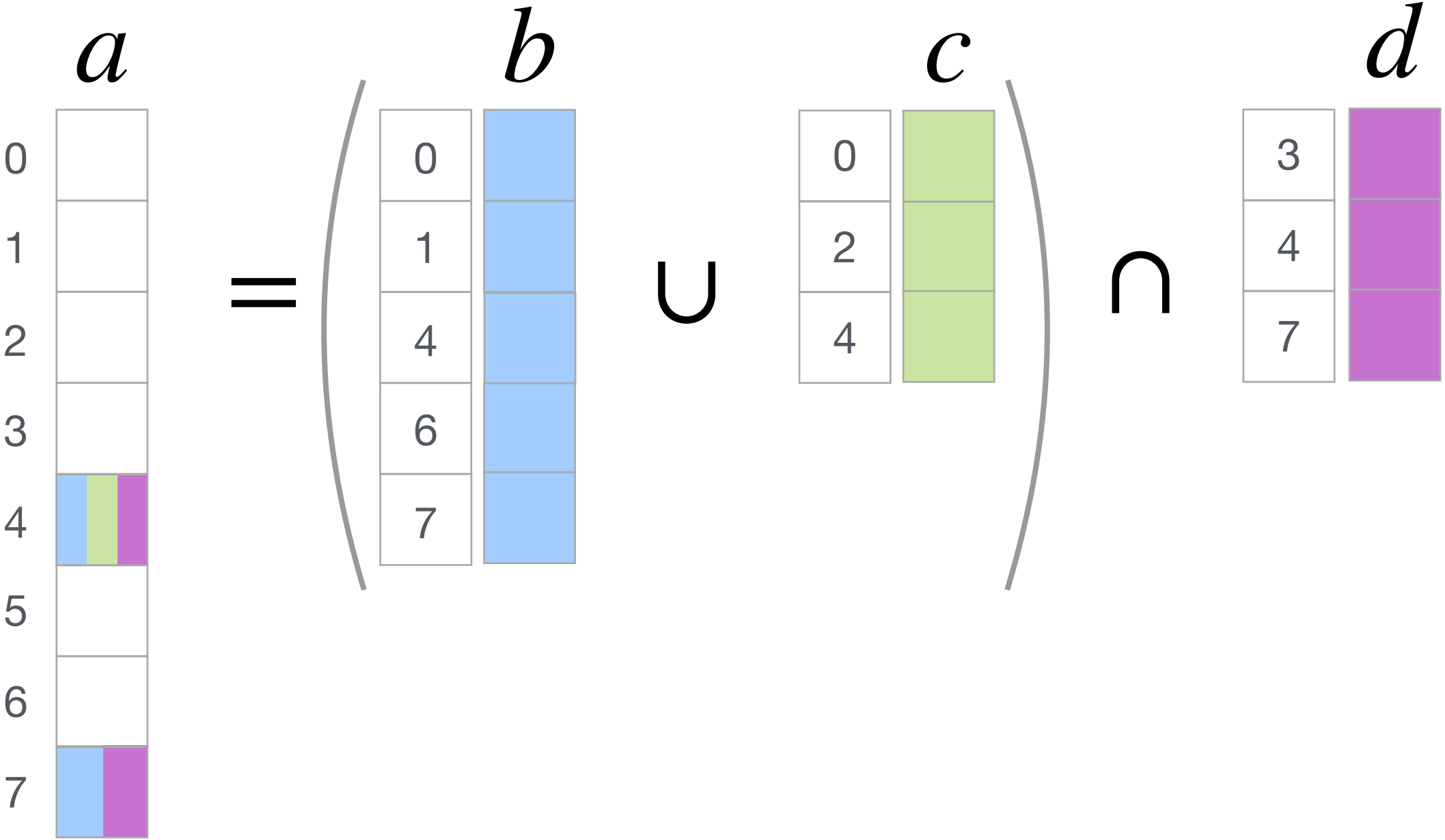
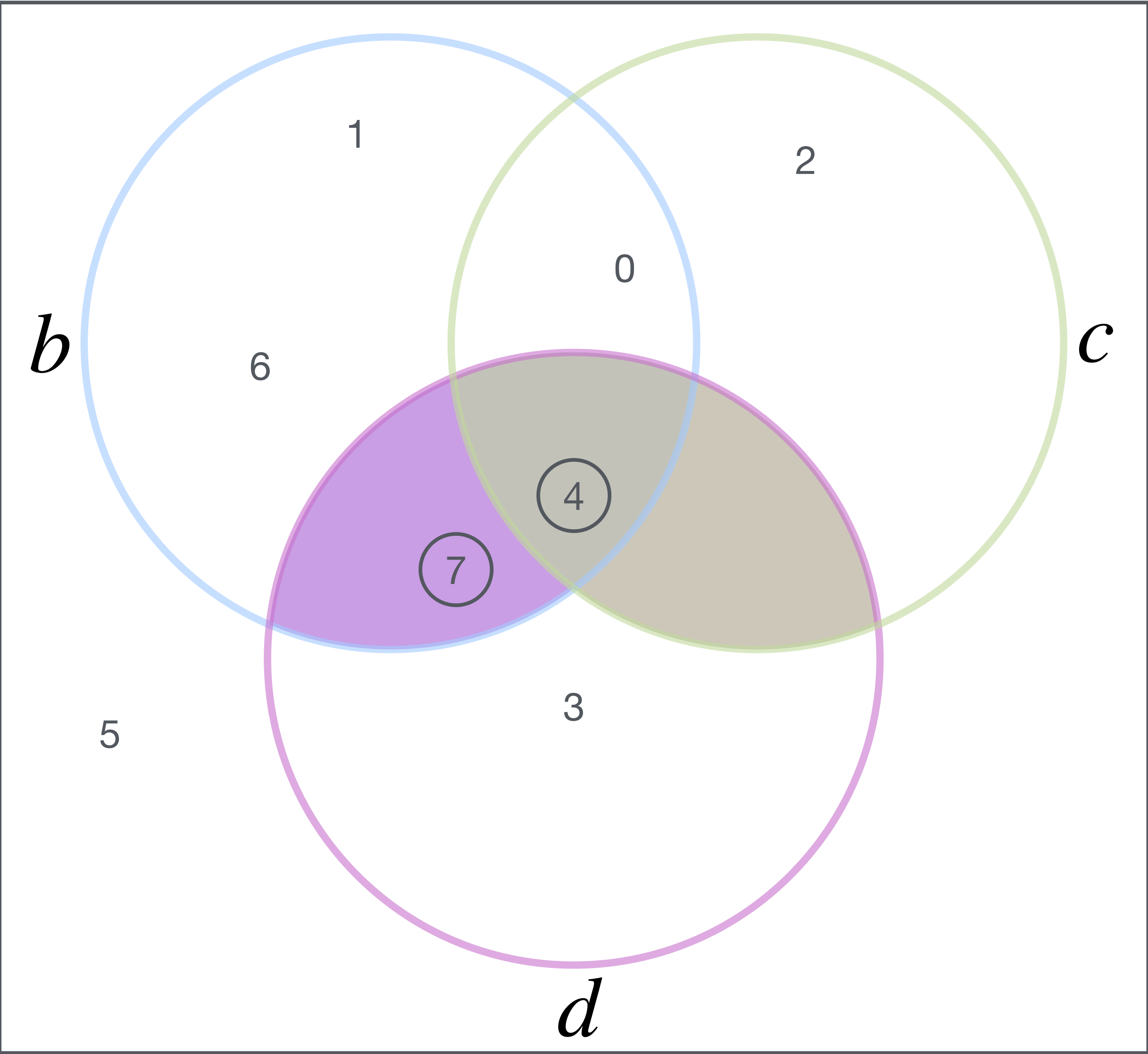
Data structure coiteration

Coordinate Space



Data structure coiteration

Coordinate Space



Iteration lattice for multiplications

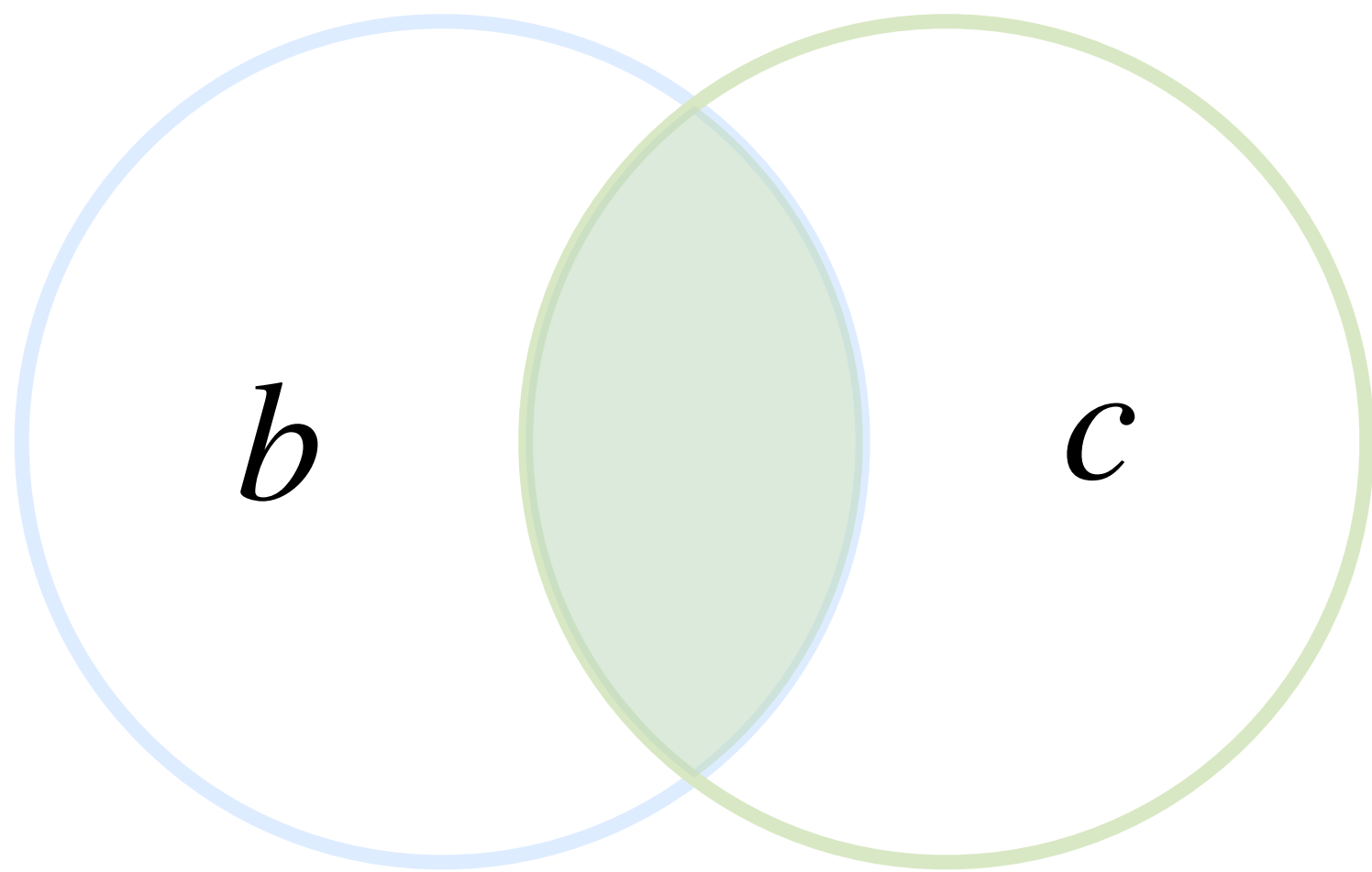
$$a_i = b_i c_i$$

Iteration lattice for multiplications

$$a_i = b_i c_i$$



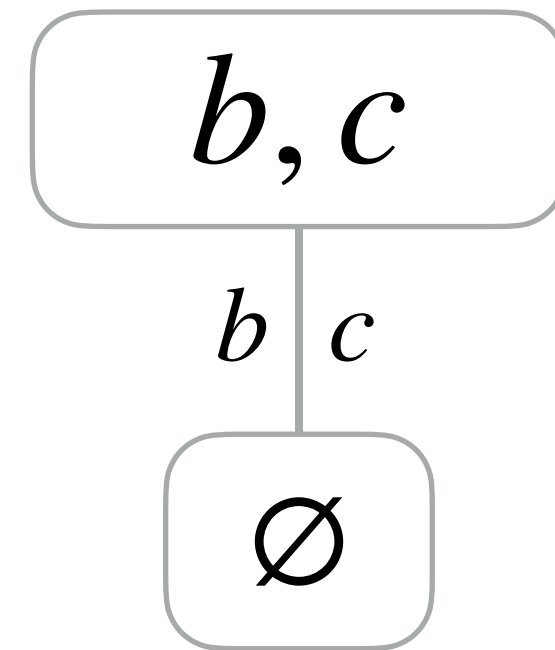
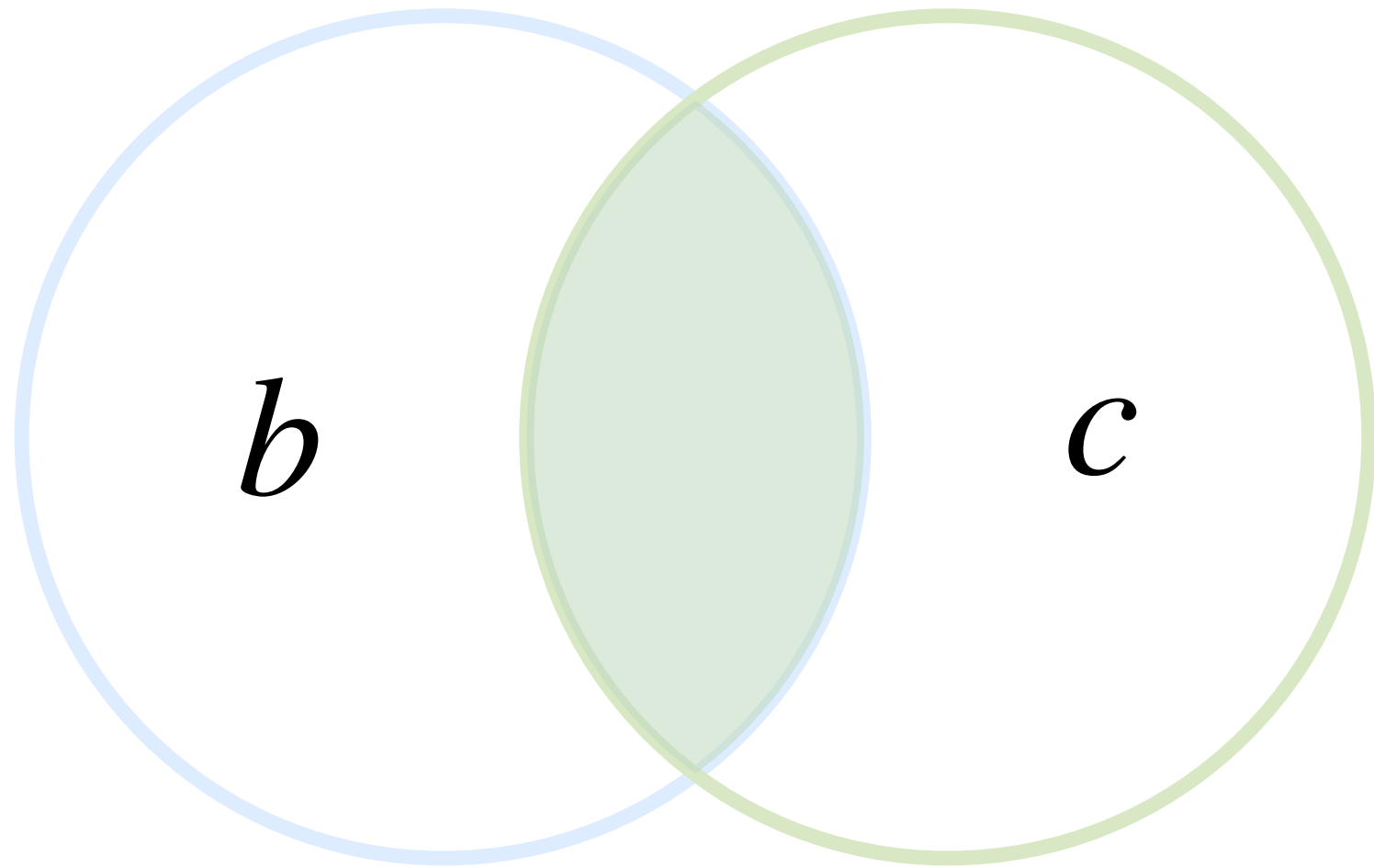
Multiplication requires intersection



$$b \cap c$$

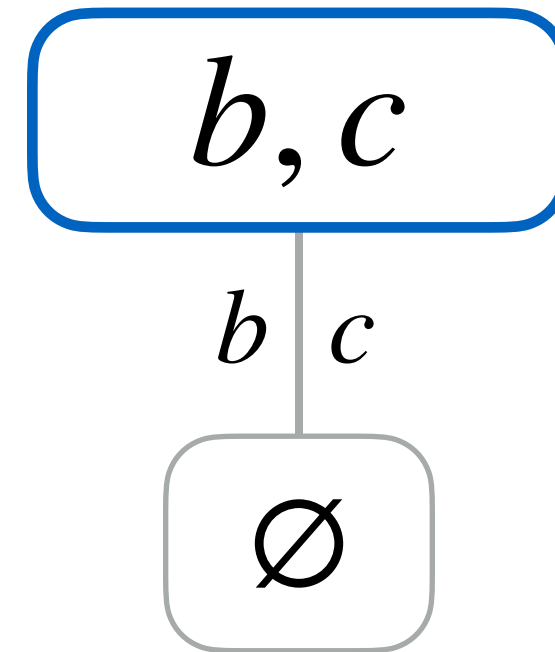
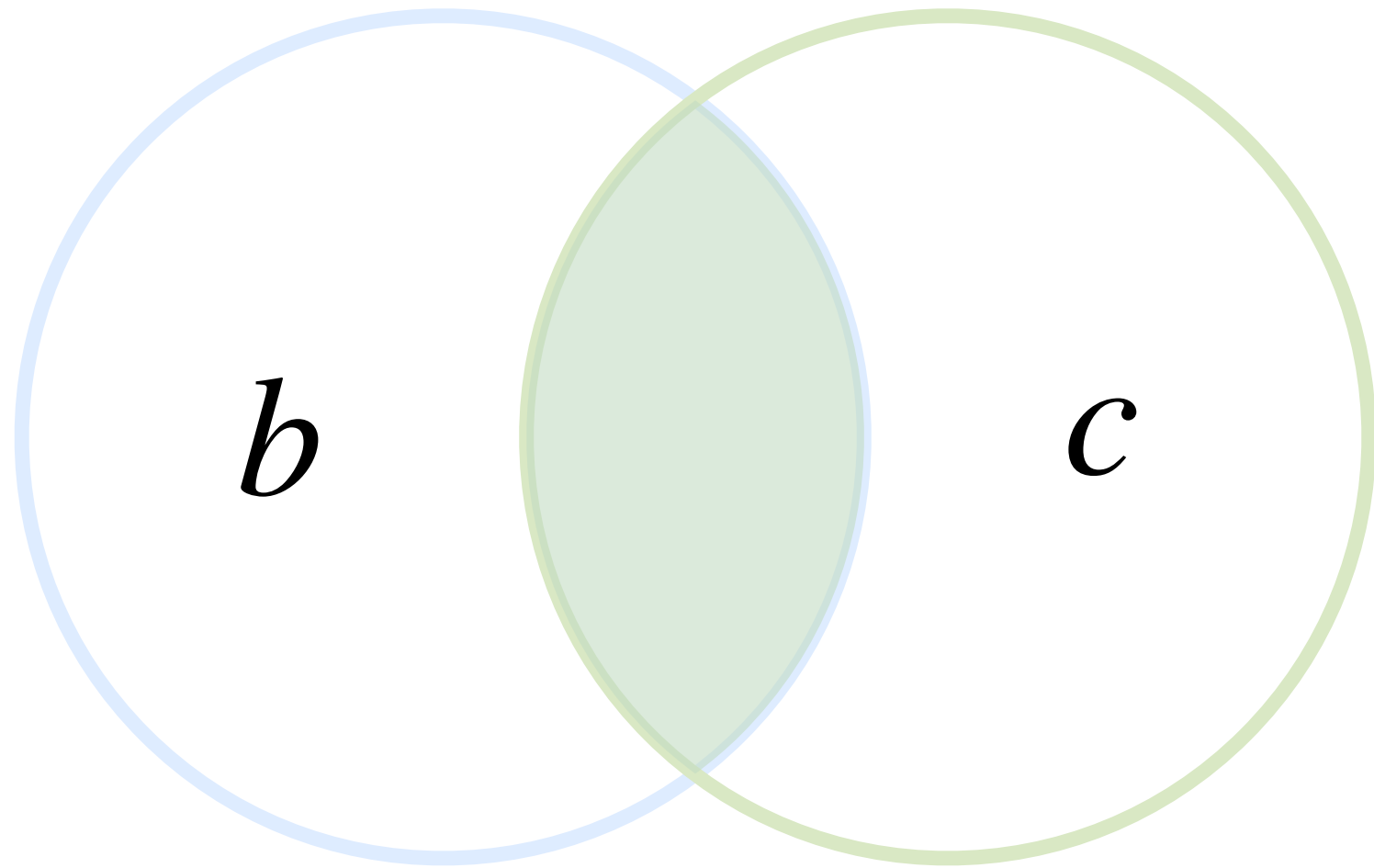
Iteration lattice for multiplications

$$a_i = b_i c_i$$



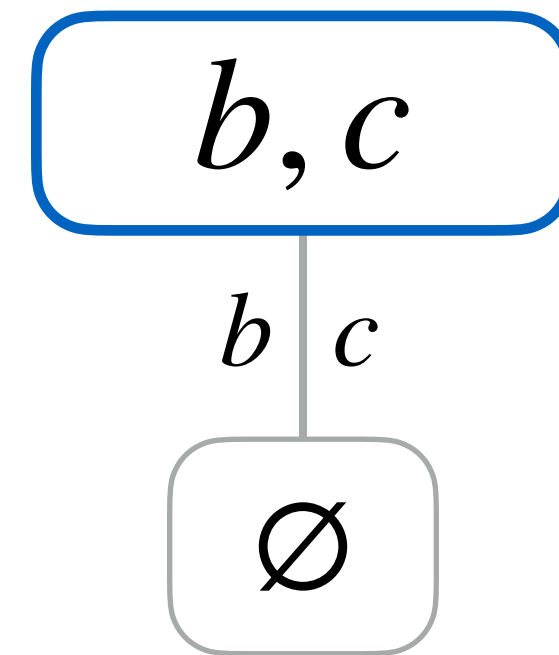
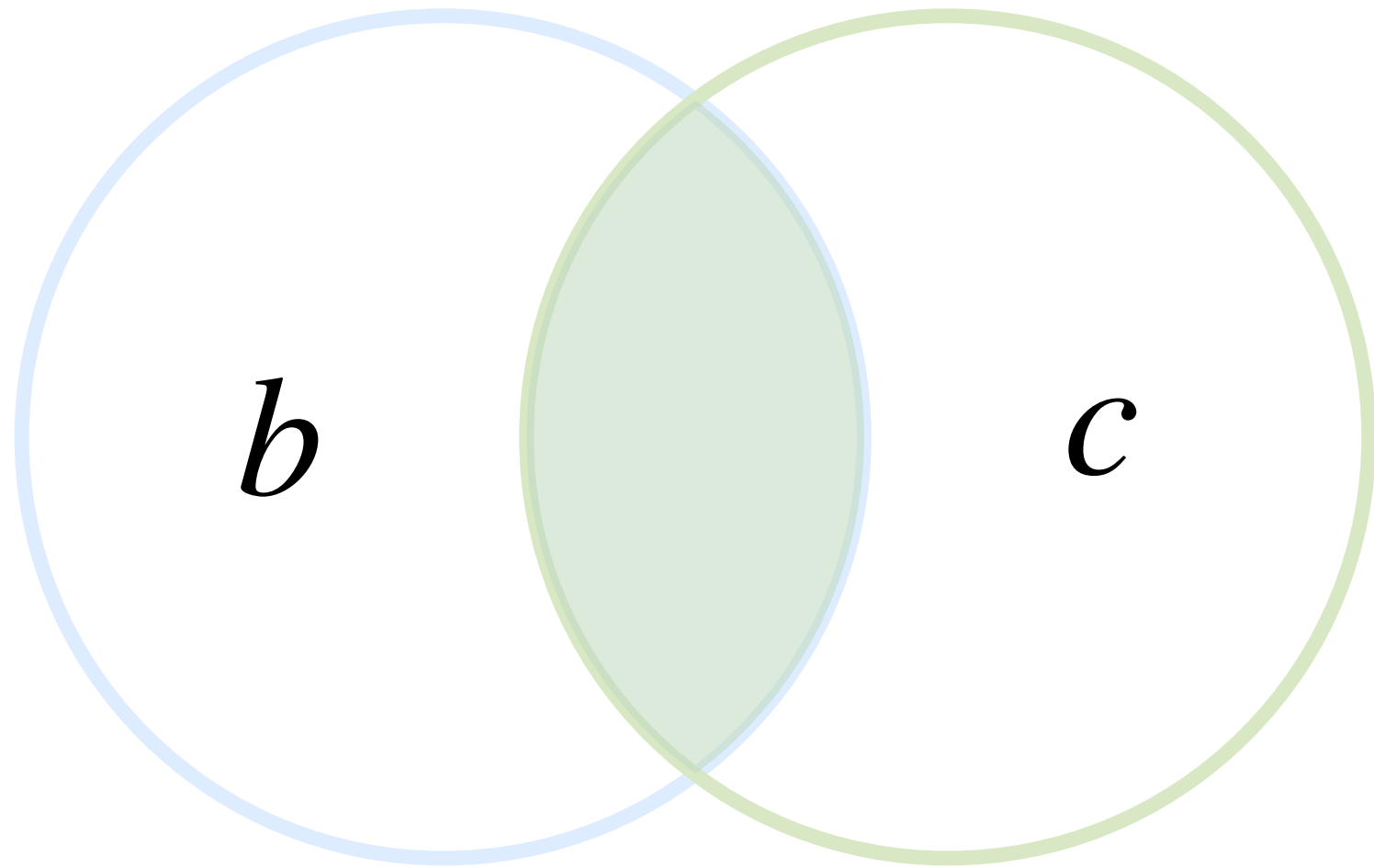
Iteration lattice for multiplications

$$a_i = b_i c_i$$



Iteration lattice for multiplications

$$a_i = b_i c_i$$

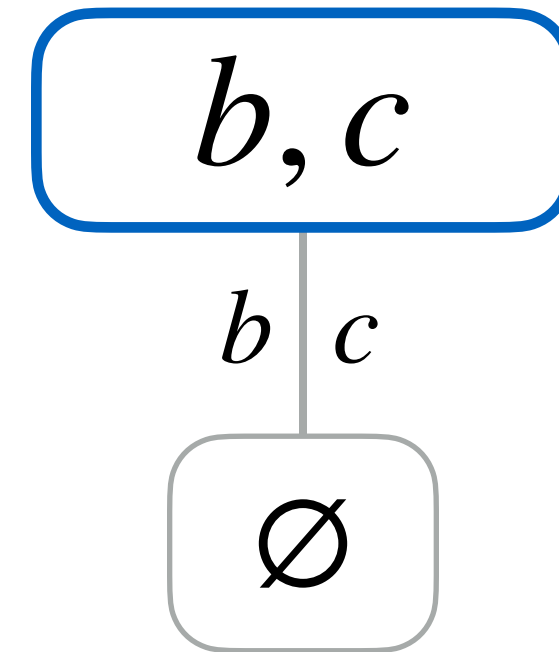
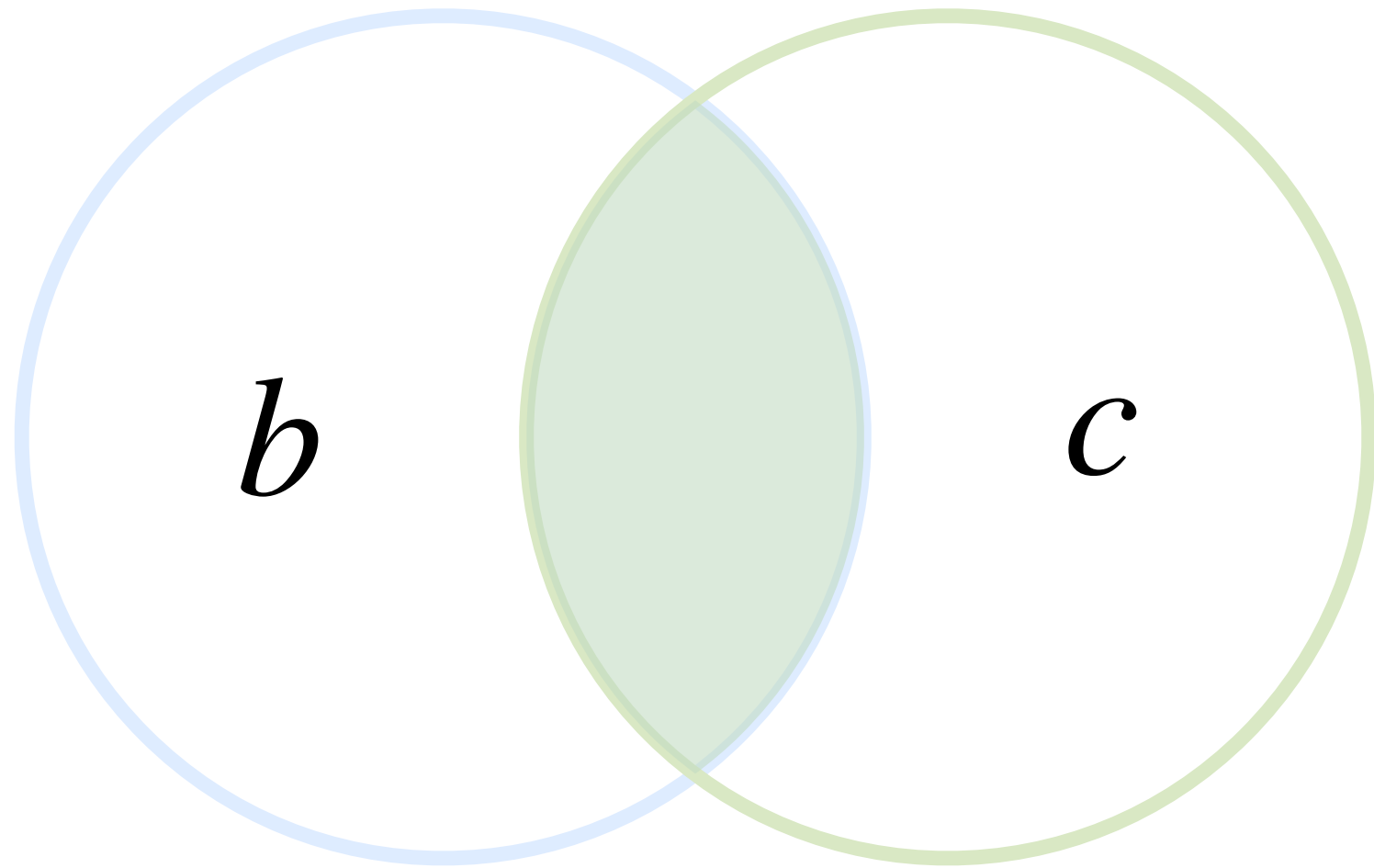


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);

    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for multiplications

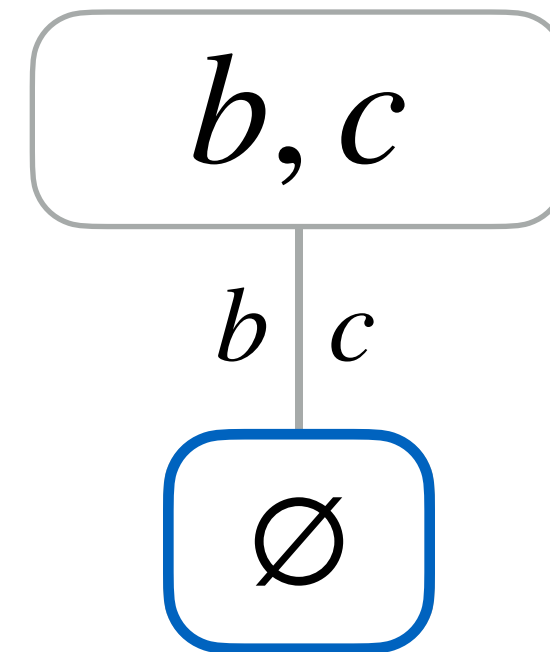
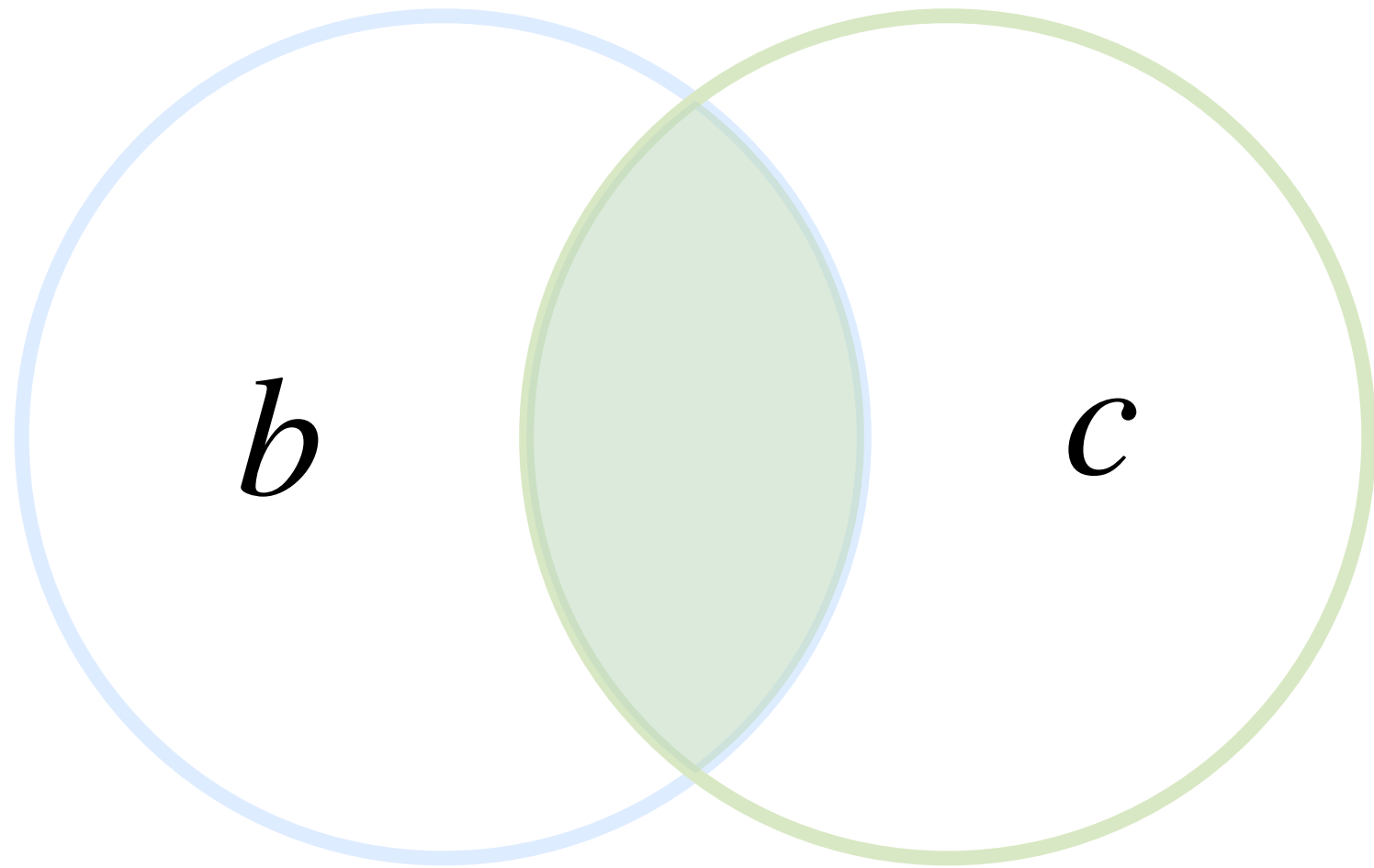
$$a_i = b_i c_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for multiplications

$$a_i = b_i c_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

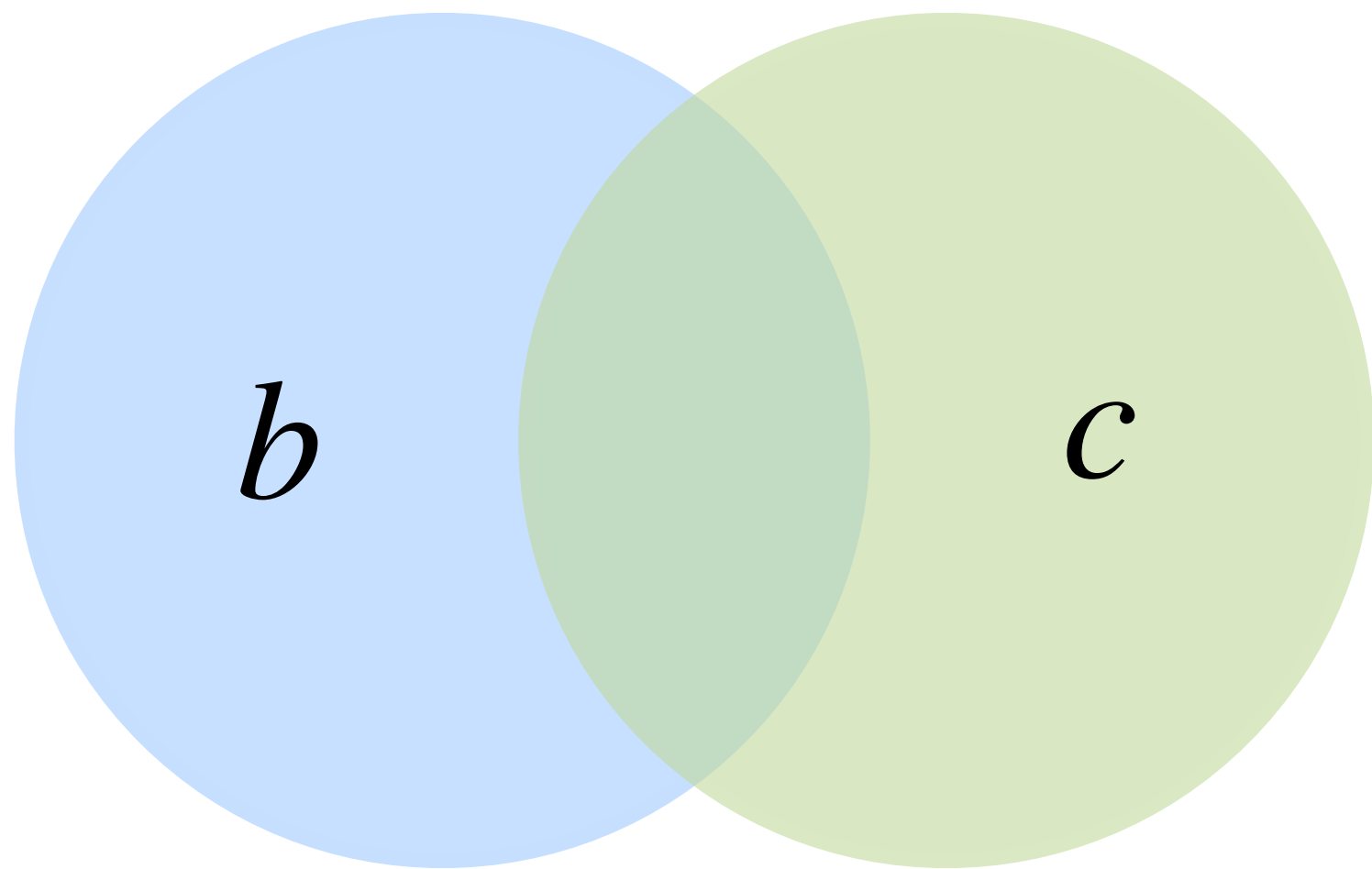

Iteration lattice for additions

$$a_i = b_i + c_i$$

Iteration lattice for additions

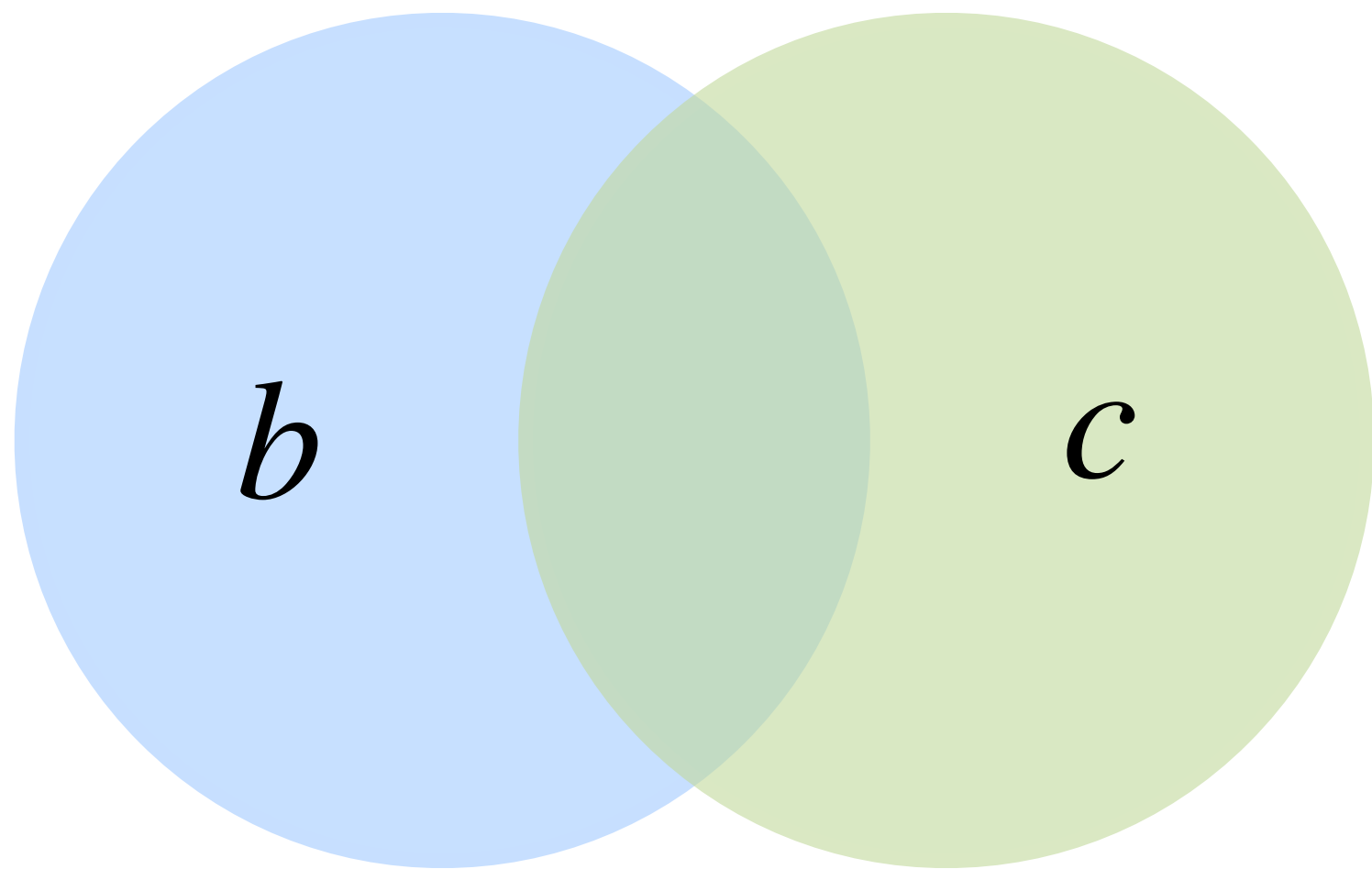
$$a_i = b_i + c_i$$

Addition requires union



Iteration lattice for additions

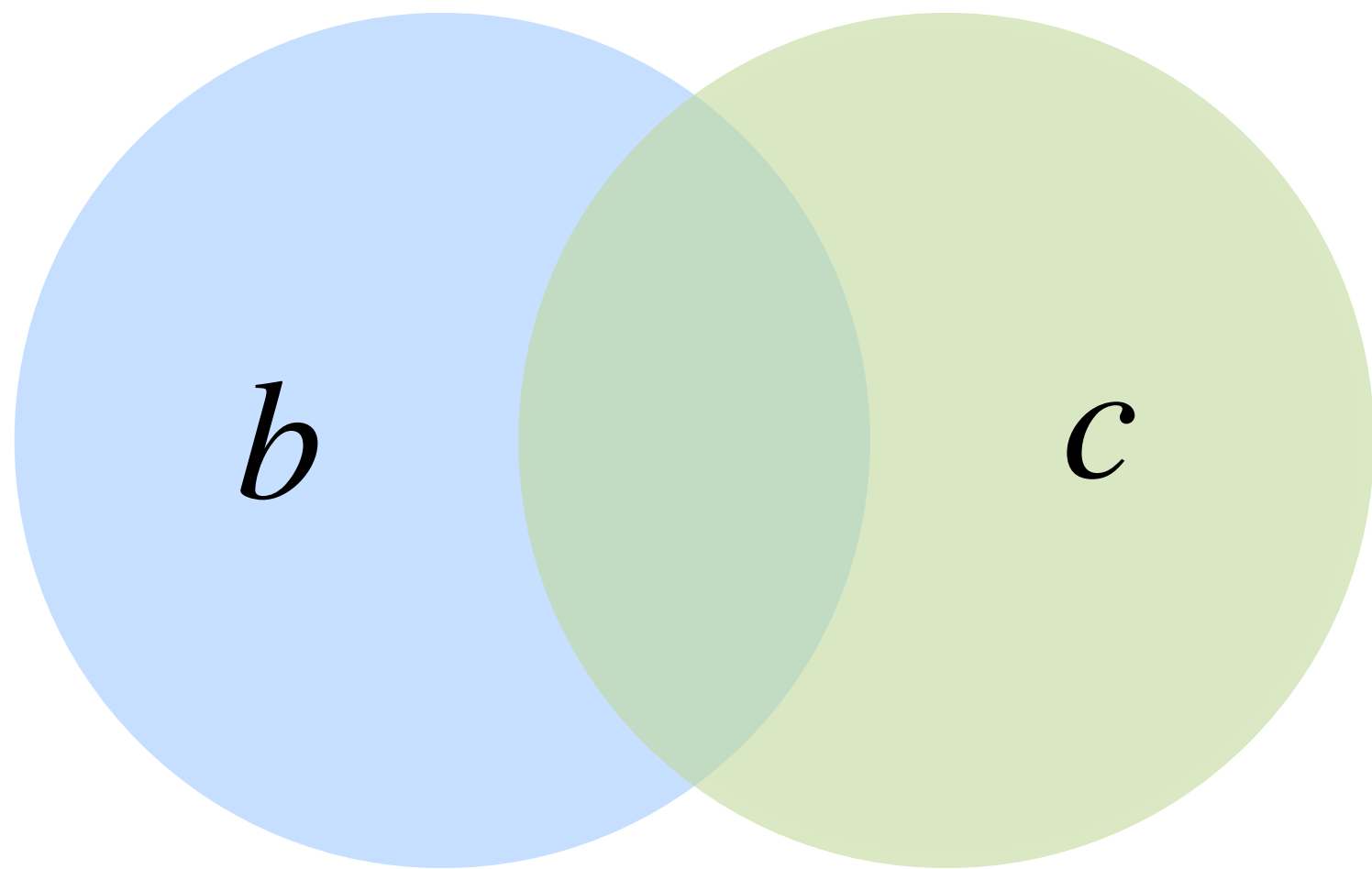
$$a_i = b_i + c_i$$



$$b \cup c$$

Iteration lattice for additions

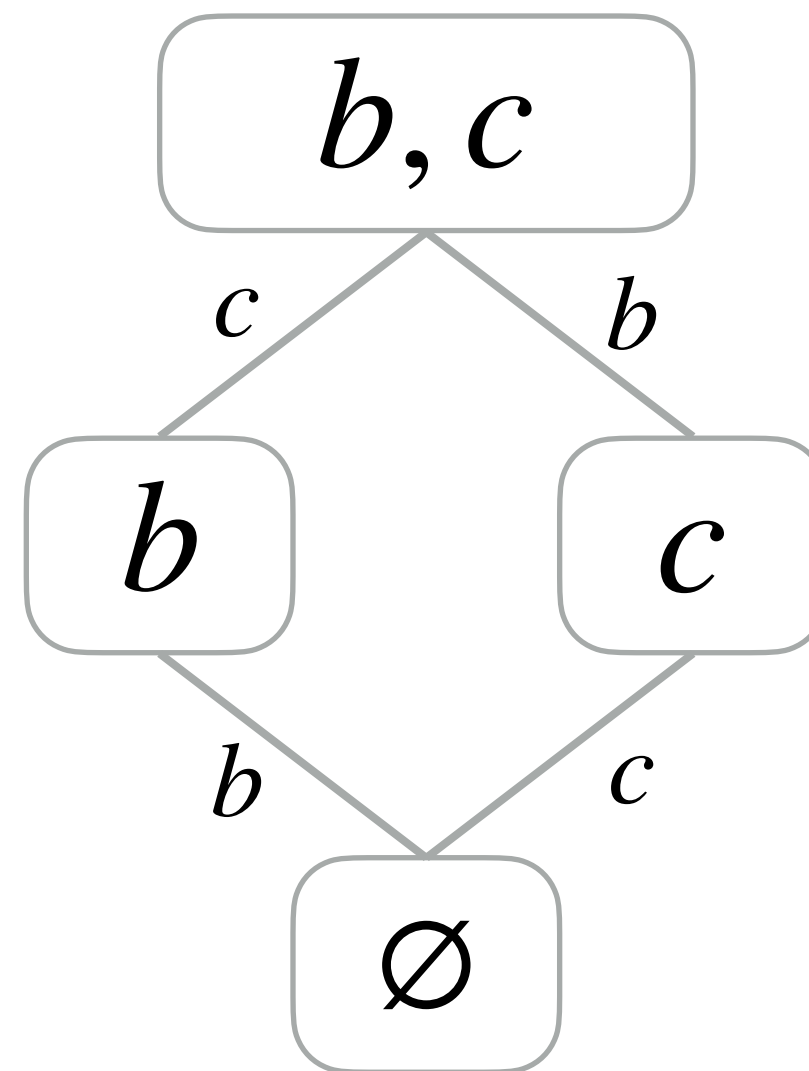
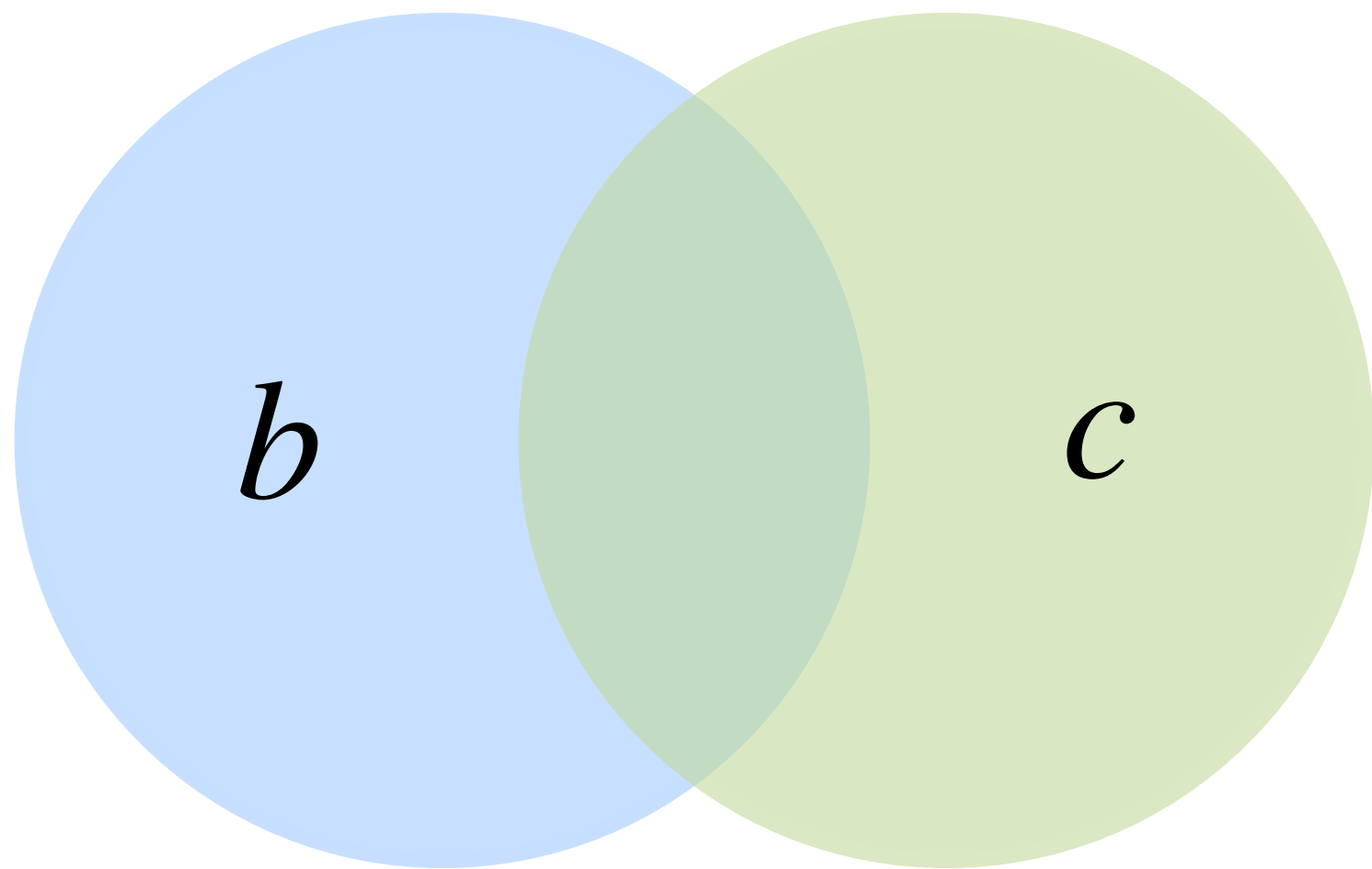
$$a_i = b_i + c_i$$



$$b \cup c = (b \cap c) \cup b \cup c$$

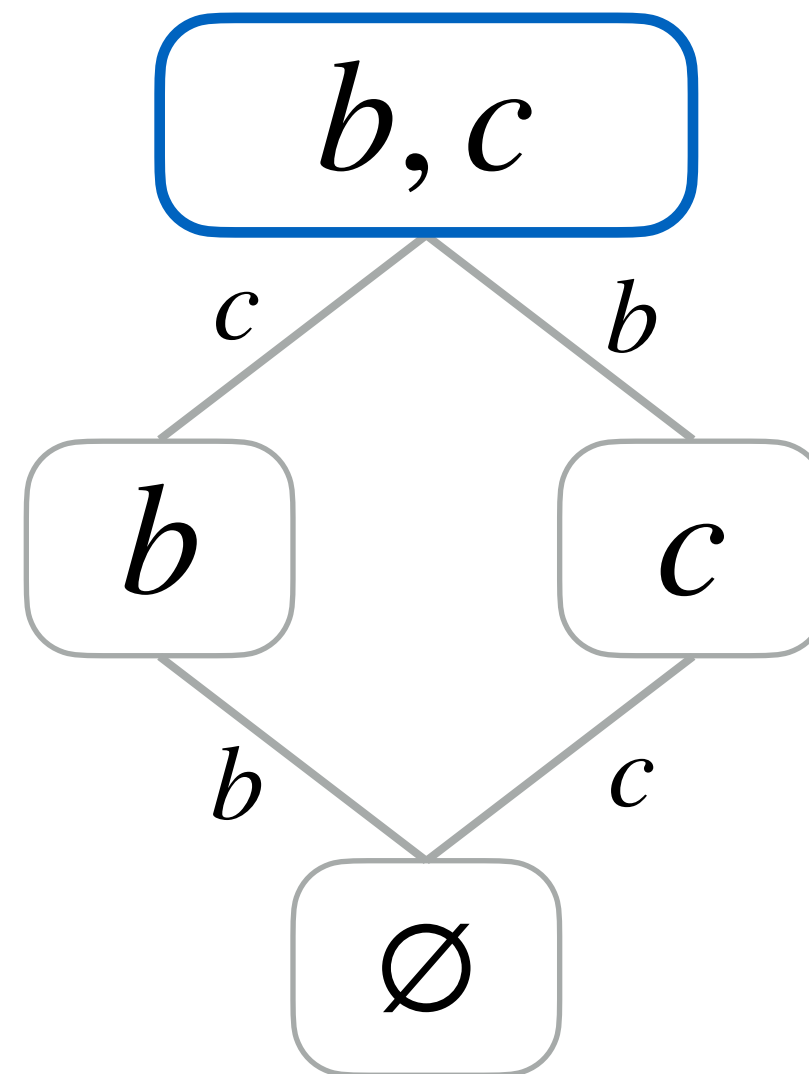
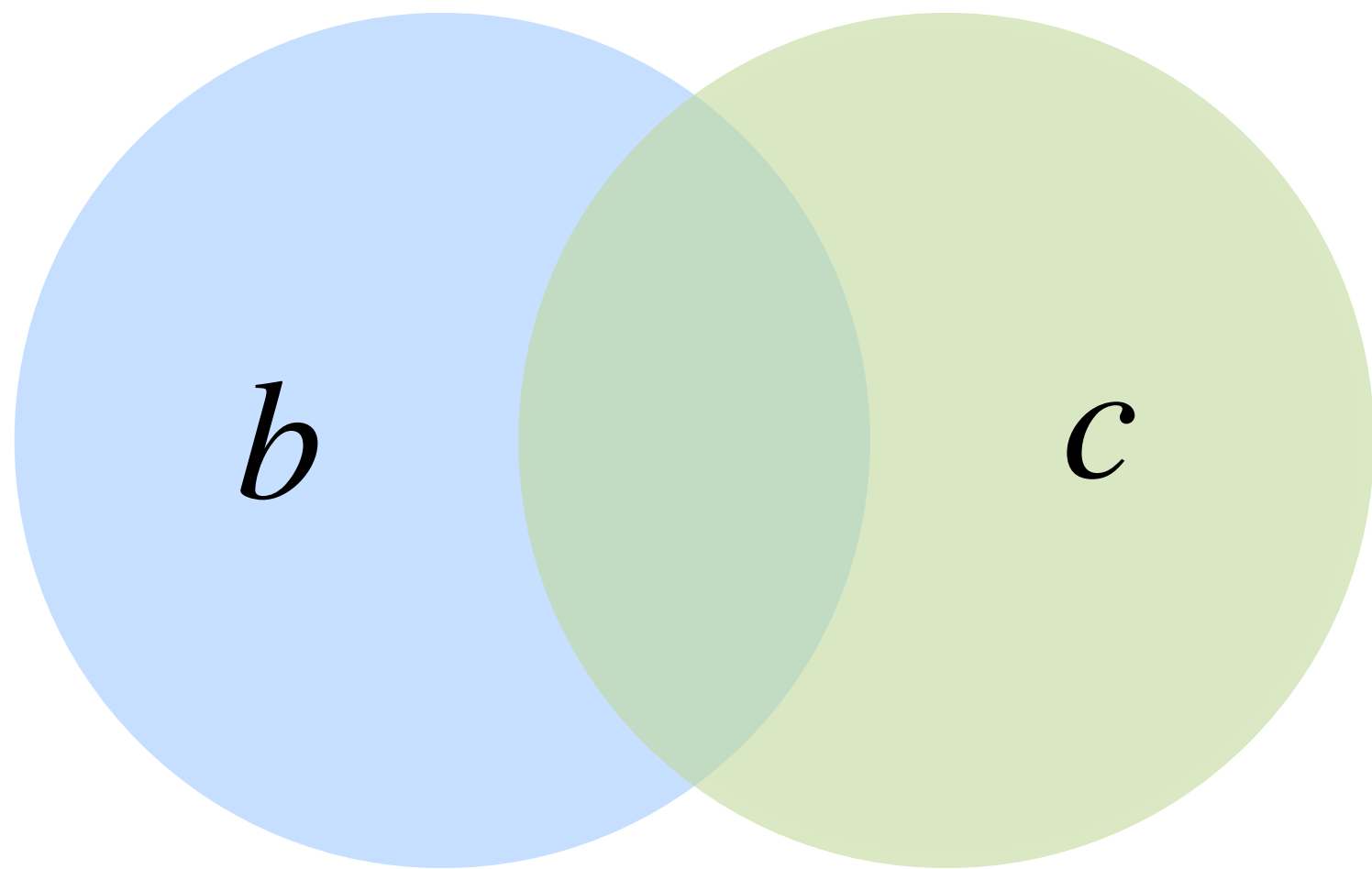
Iteration lattice for additions

$$a_i = b_i + c_i$$



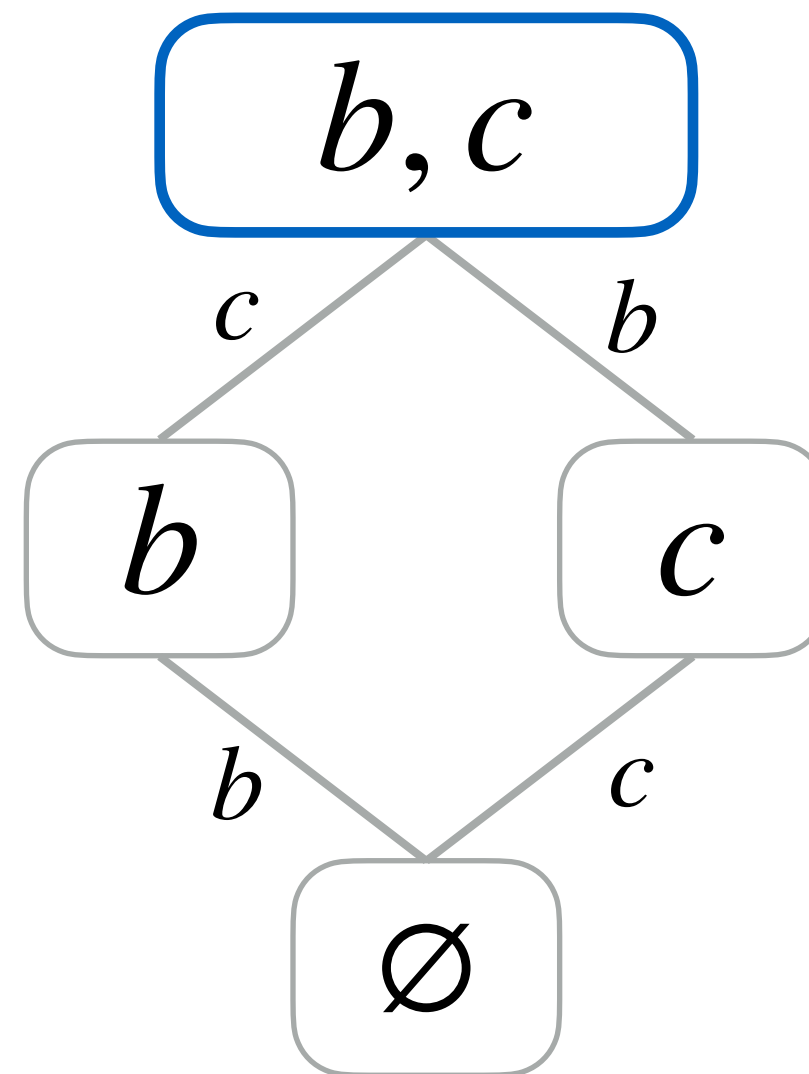
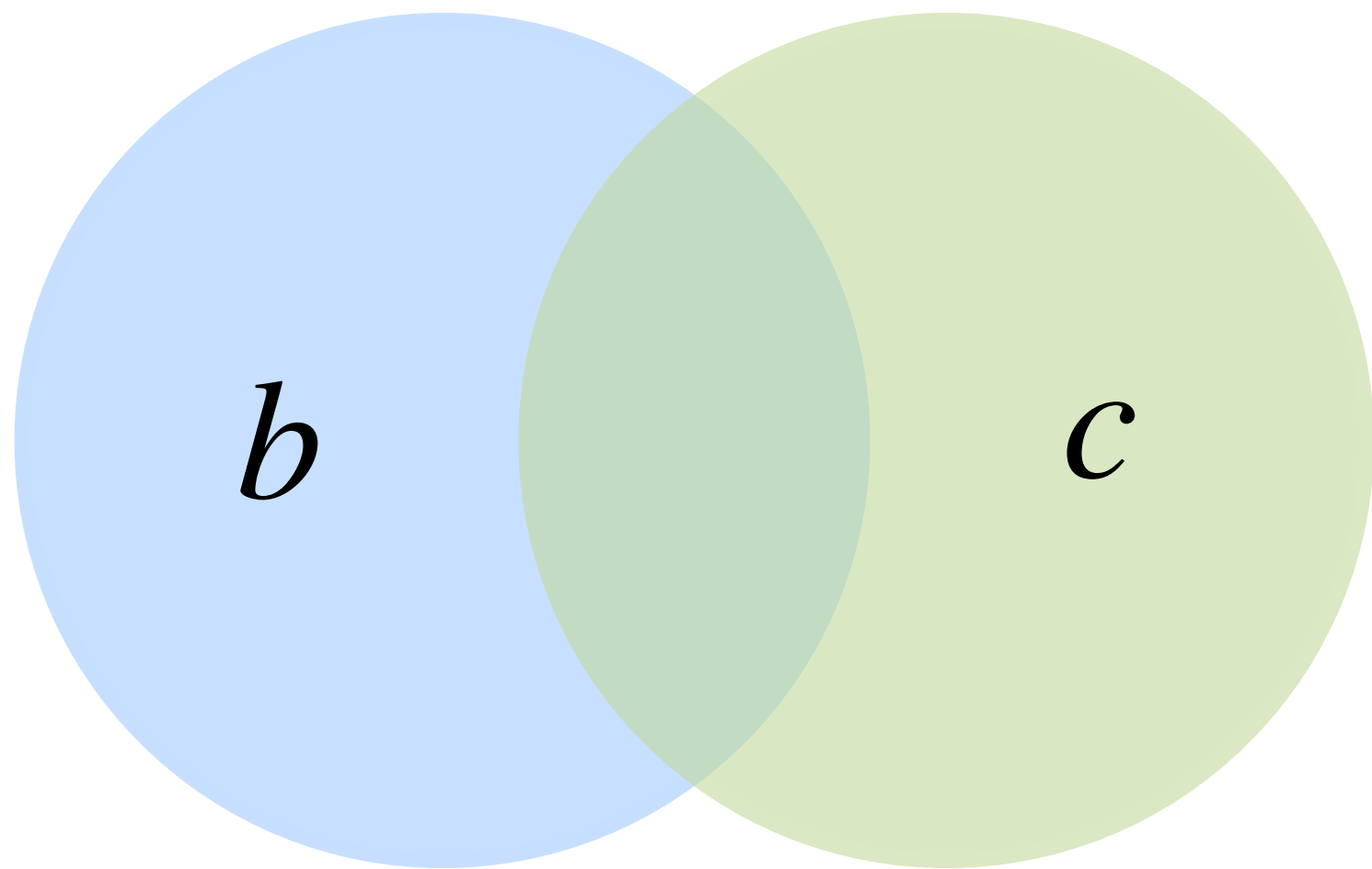
Iteration lattice for additions

$$a_i = b_i + c_i$$



Iteration lattice for additions

$$a_i = b_i + c_i$$

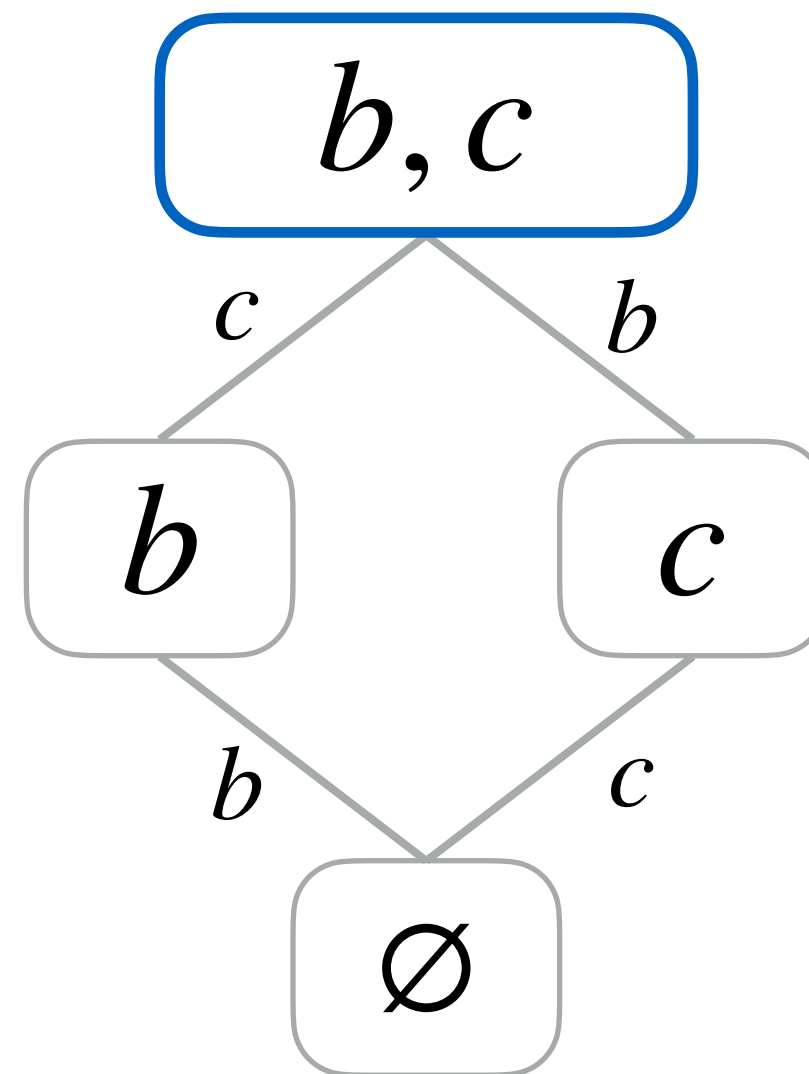
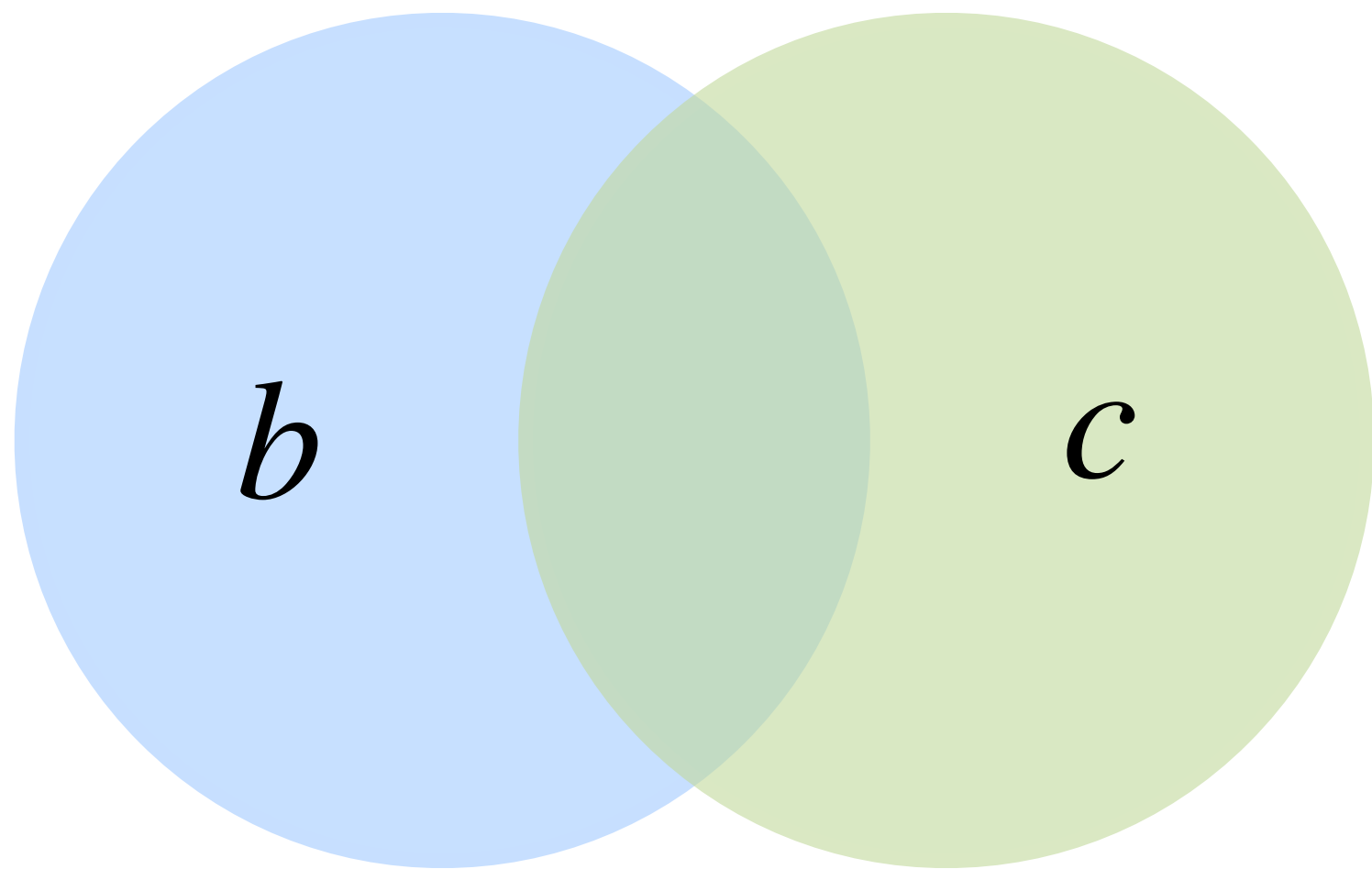


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
```

```
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for additions

$$a_i = b_i + c_i$$

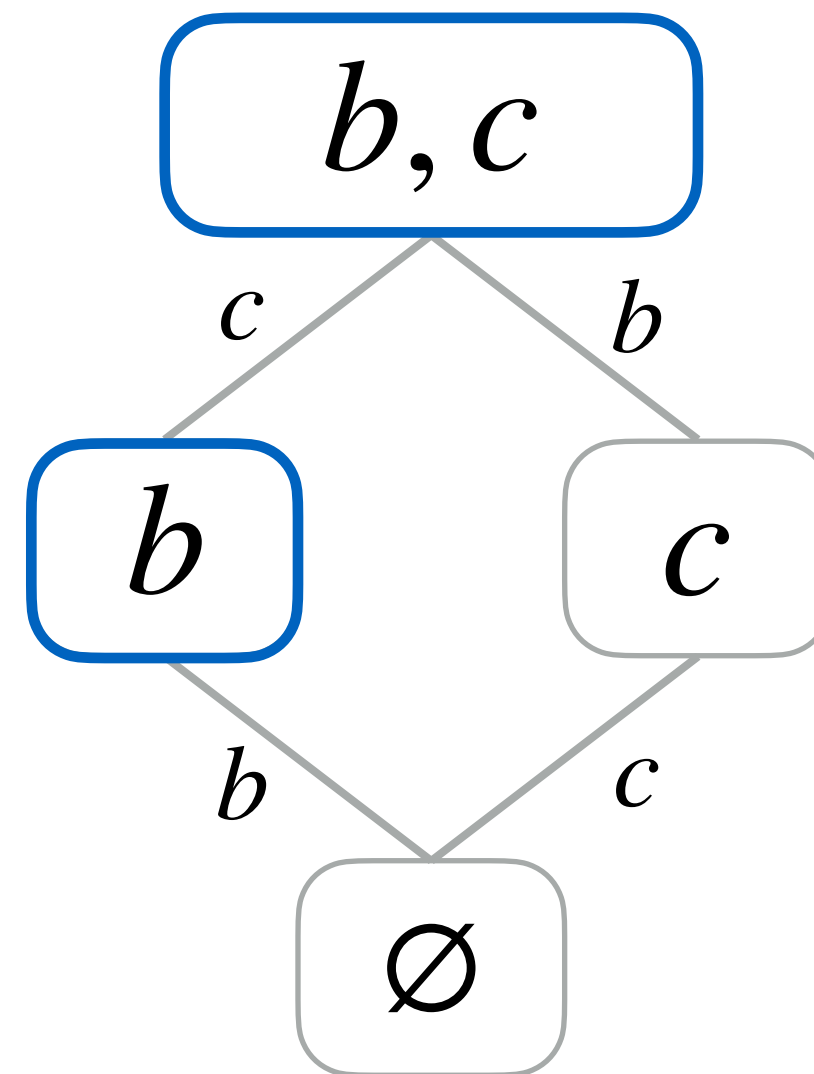
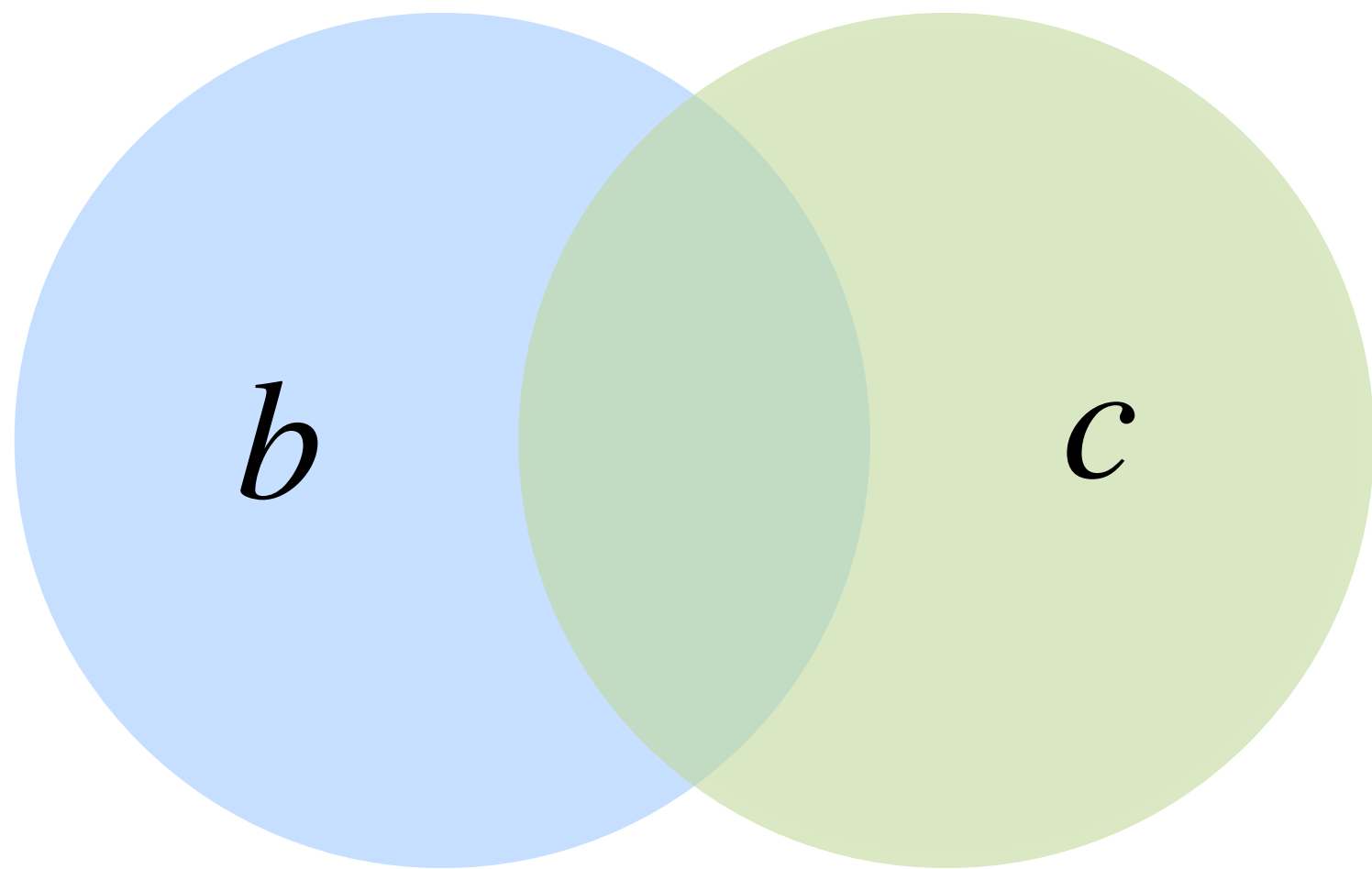


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
}
```

```
if (ib == i) pb1++;
if (ic == i) pc1++;
}
```


Iteration lattice for additions

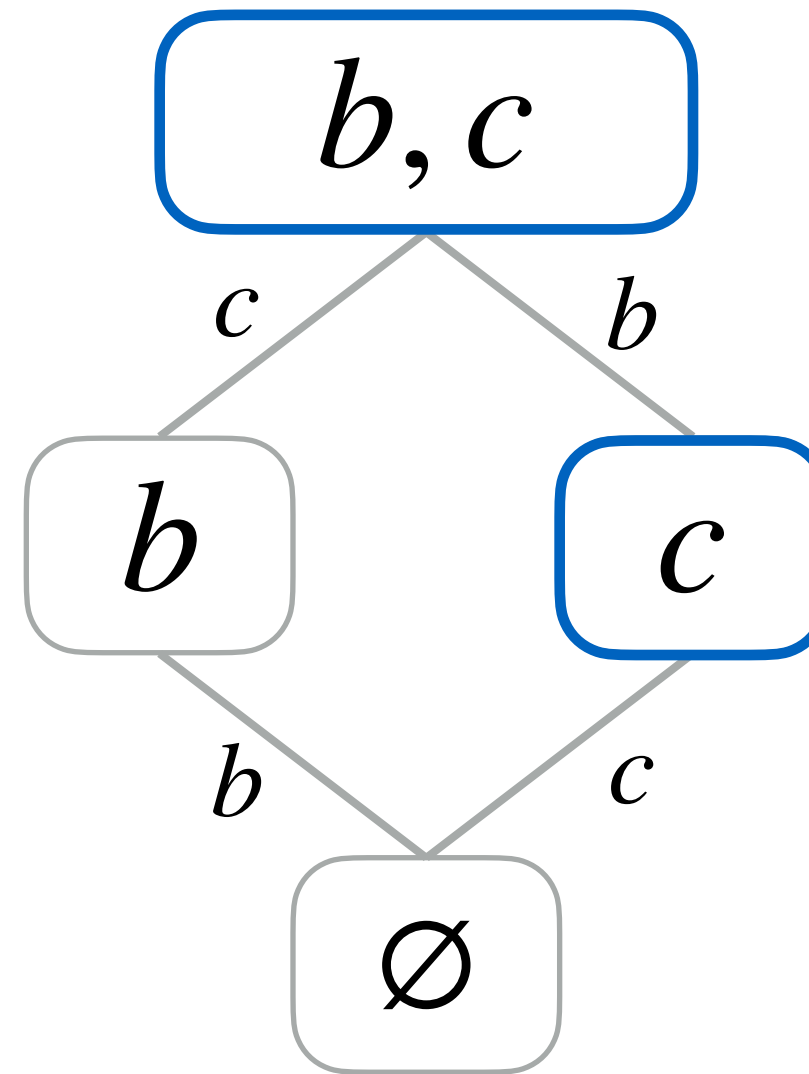
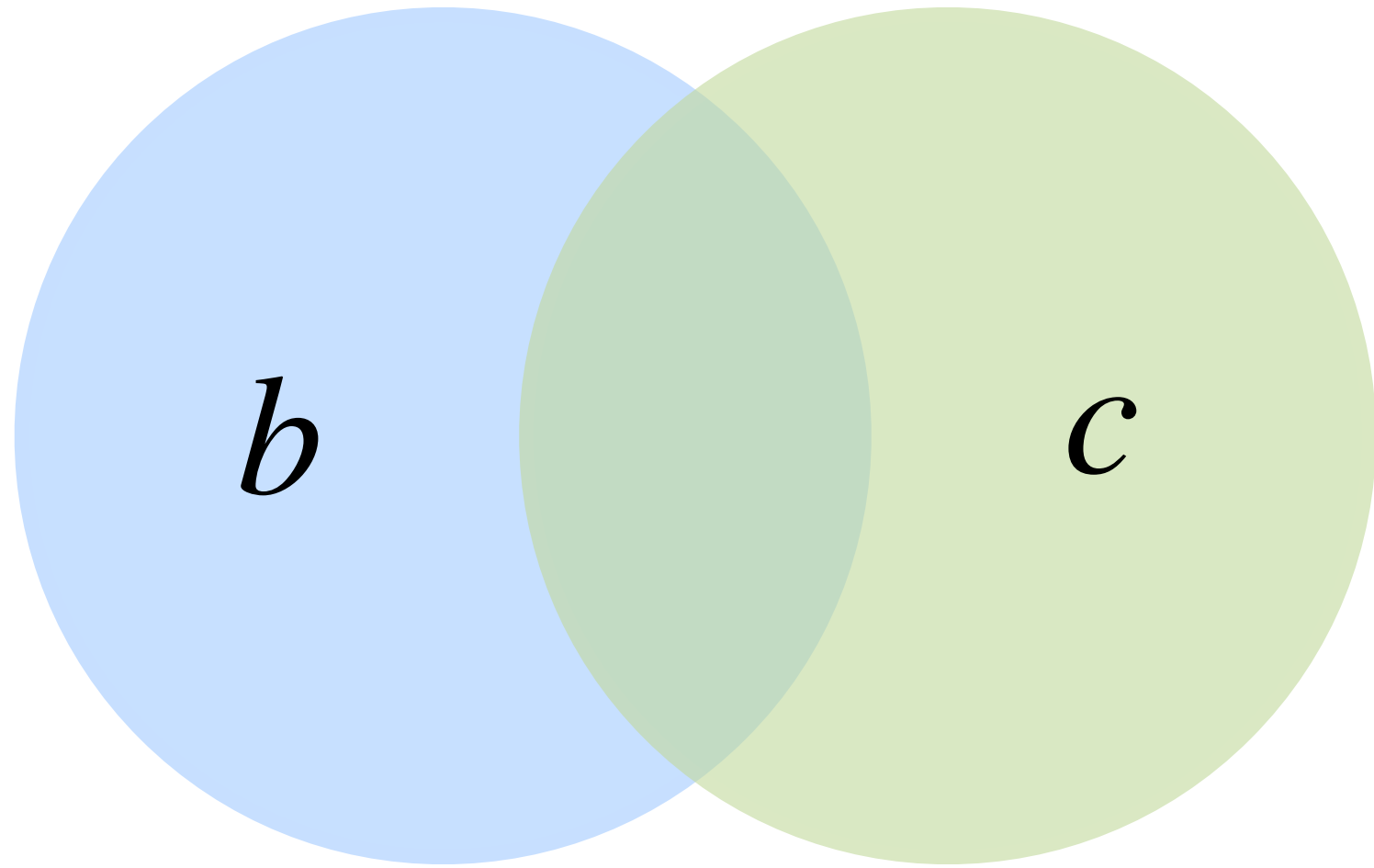
$$a_i = b_i + c_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for additions

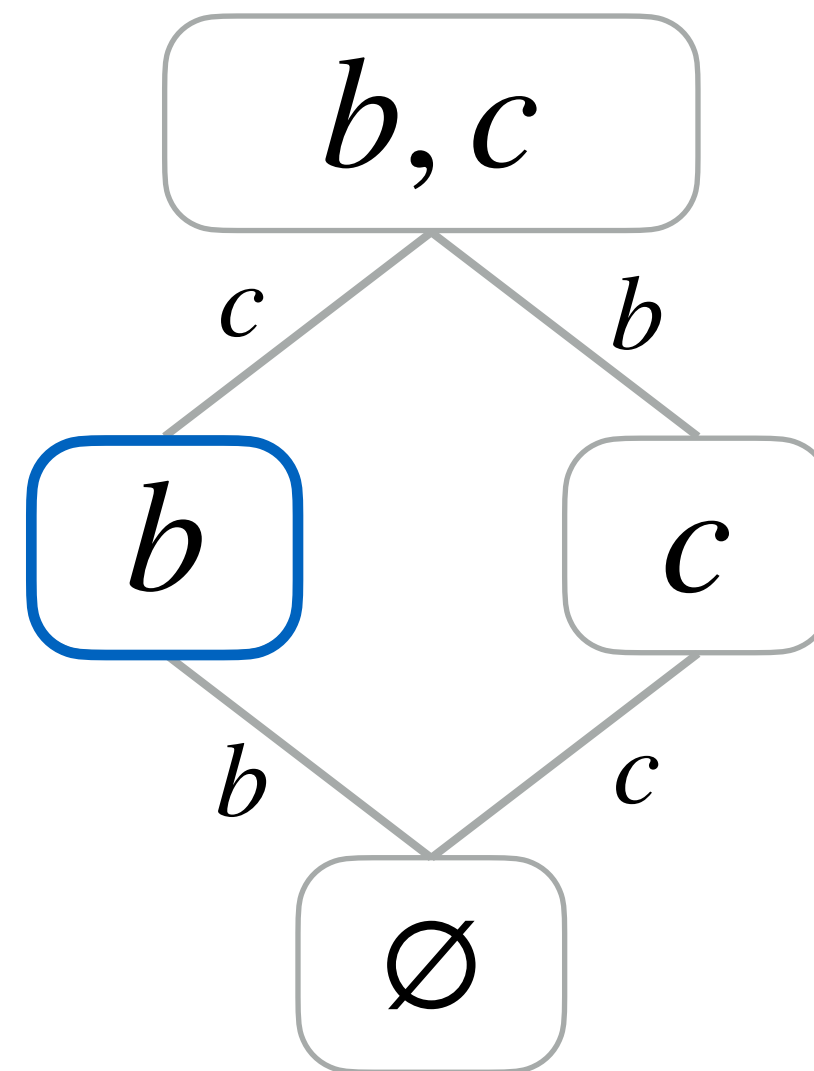
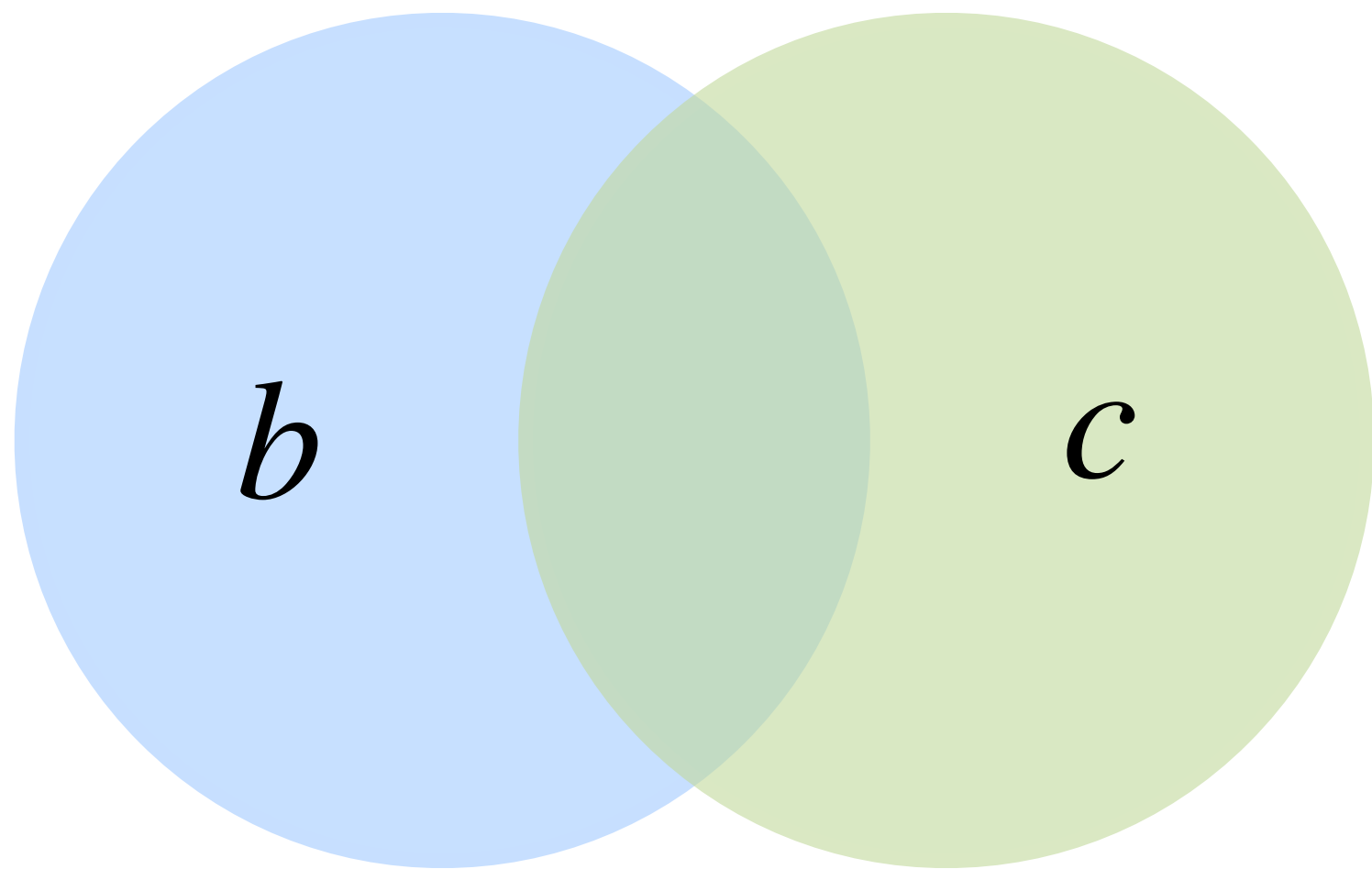
$$a_i = b_i + c_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for additions

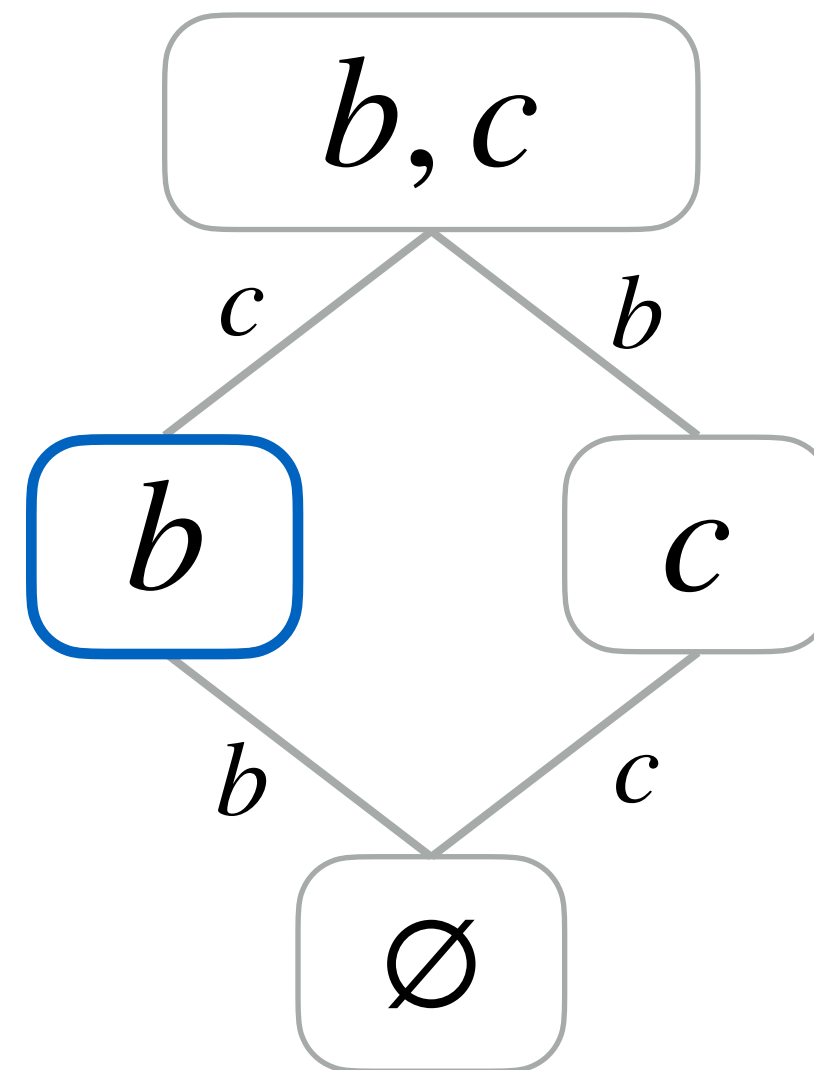
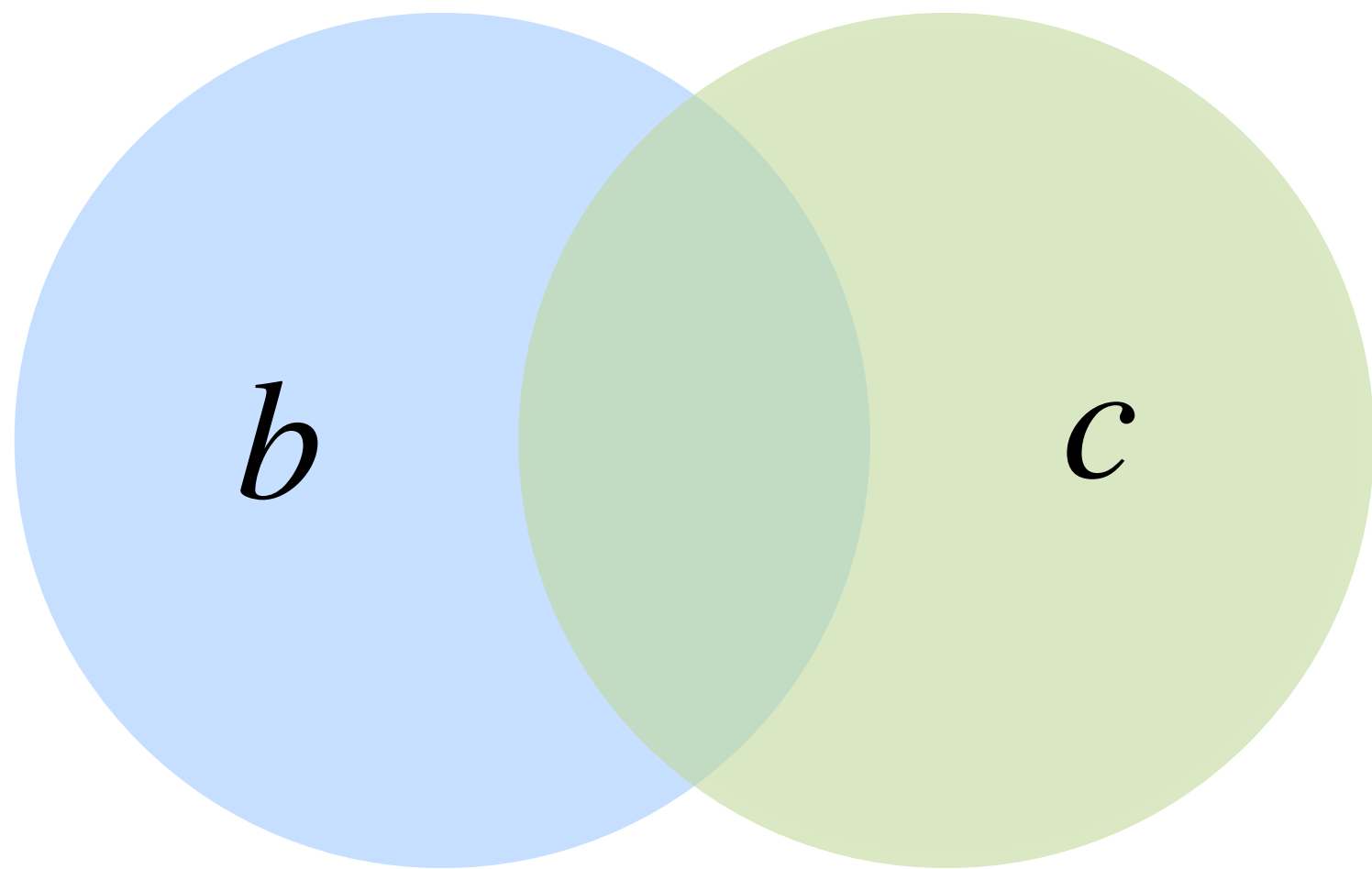
$$a_i = b_i + c_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```


Iteration lattice for additions

$$a_i = b_i + c_i$$

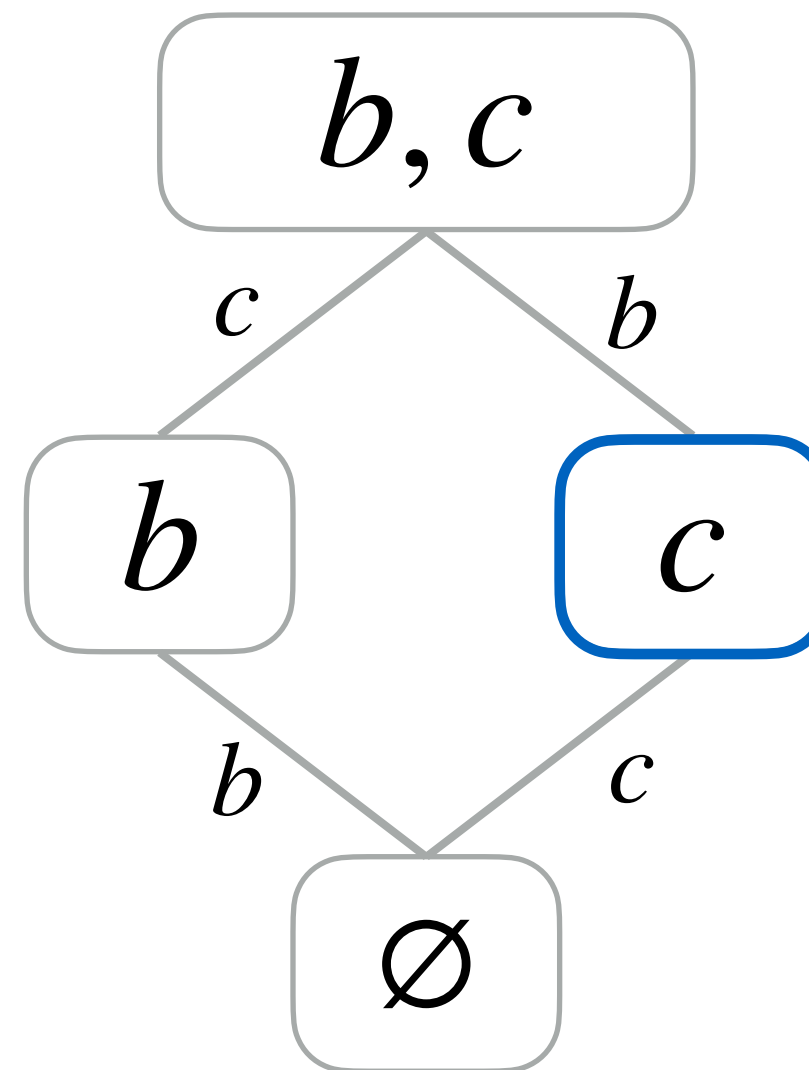
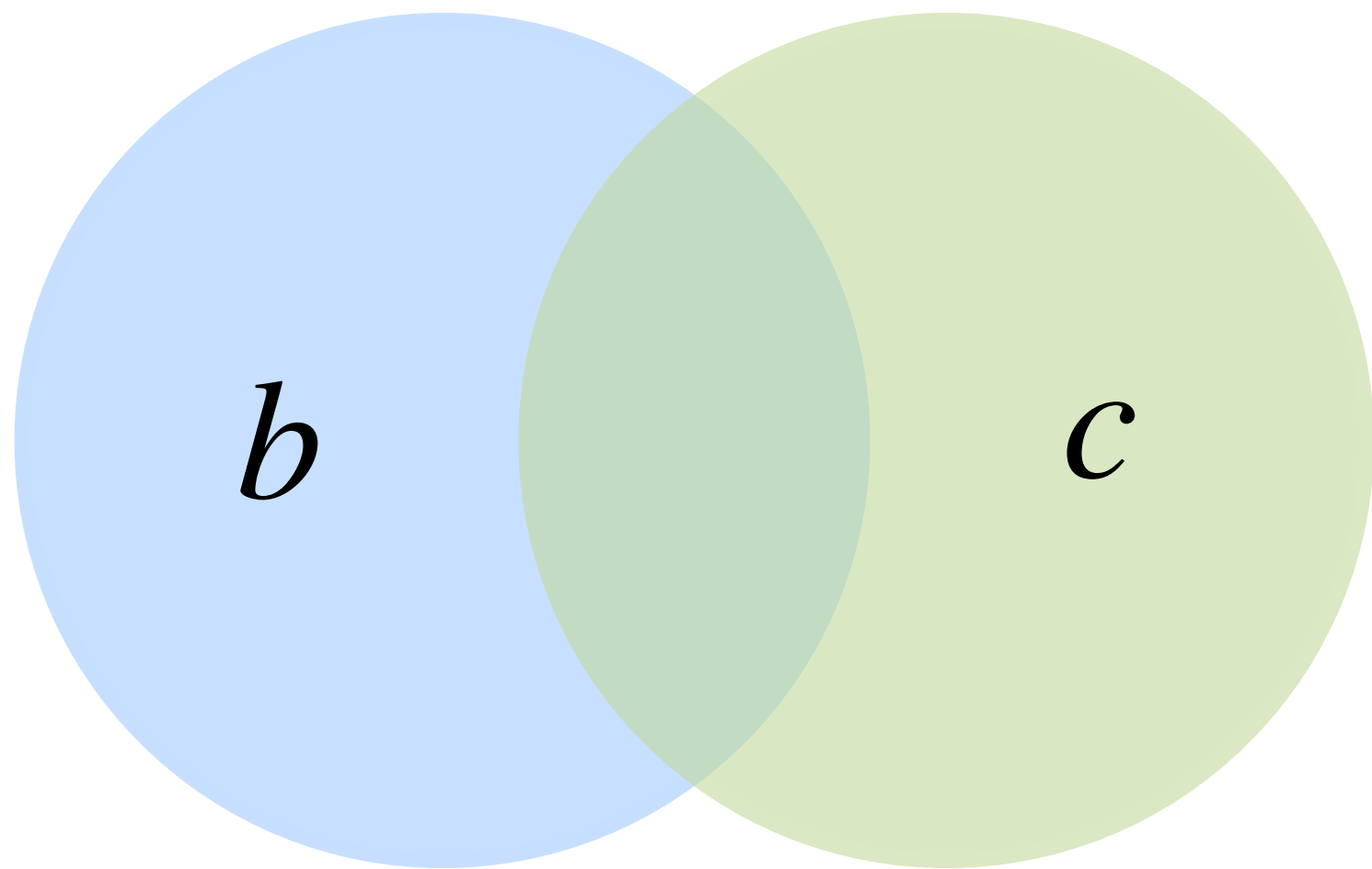


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

while (pb1 < b1_pos[1]) {
    int i = b1_crd[pb1];
    a[i] = b[pb1++];
}
```

Iteration lattice for additions

$$a_i = b_i + c_i$$



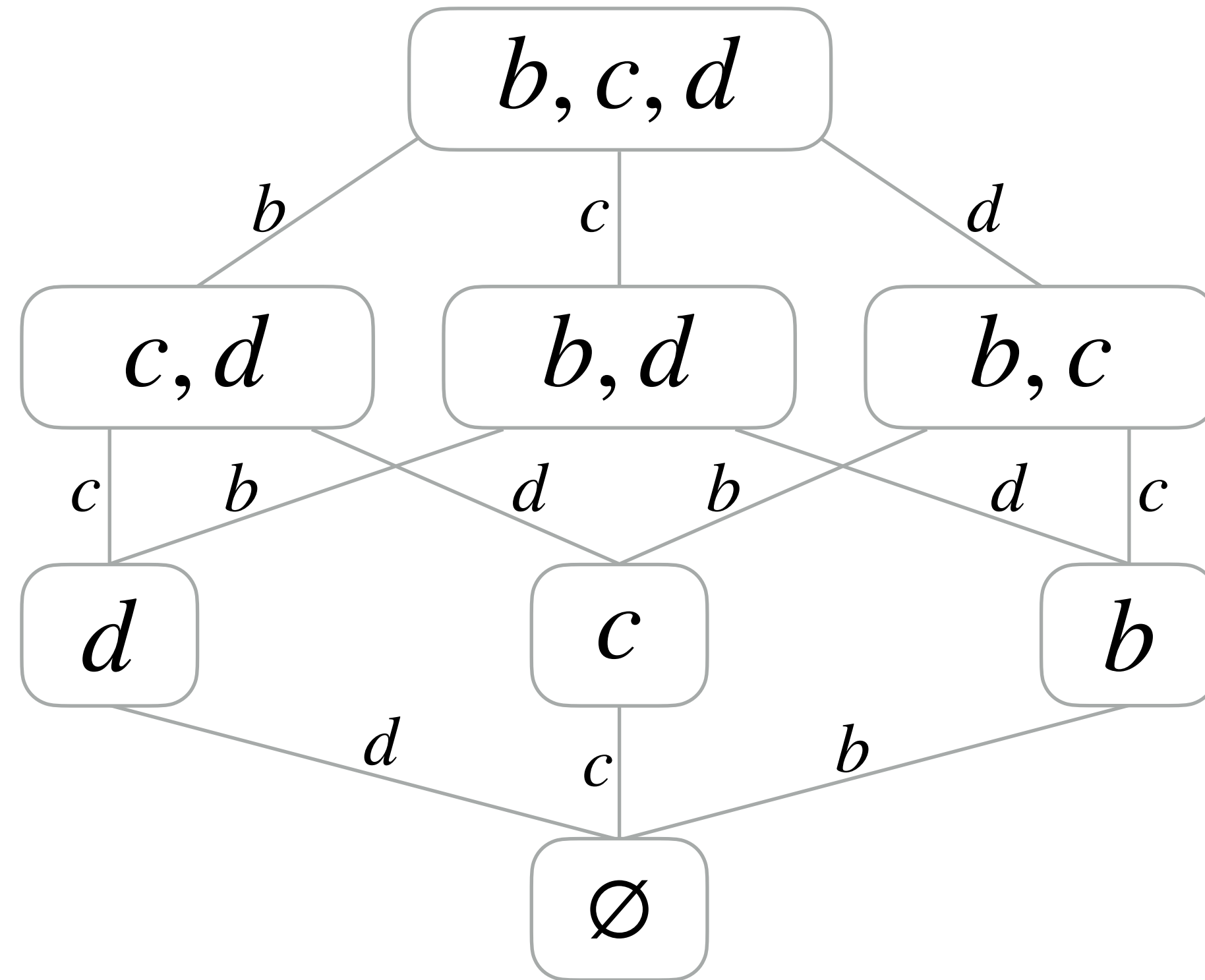
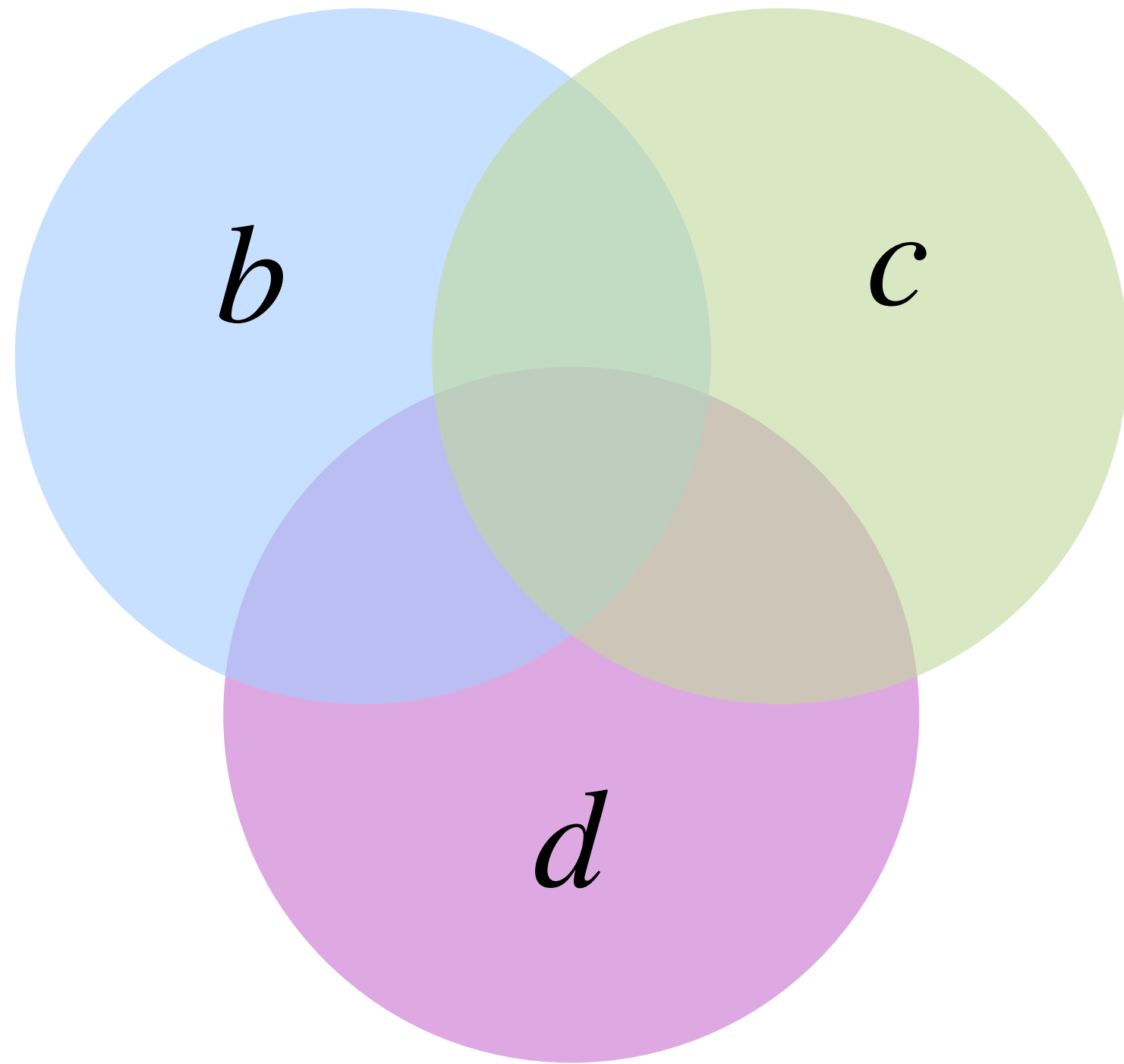
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

while (pb1 < b1_pos[1]) {
    int i = b1_crd[pb1];
    a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
    int i = c1_crd[pc1];
    a[i] = c[pc1++];
}
```

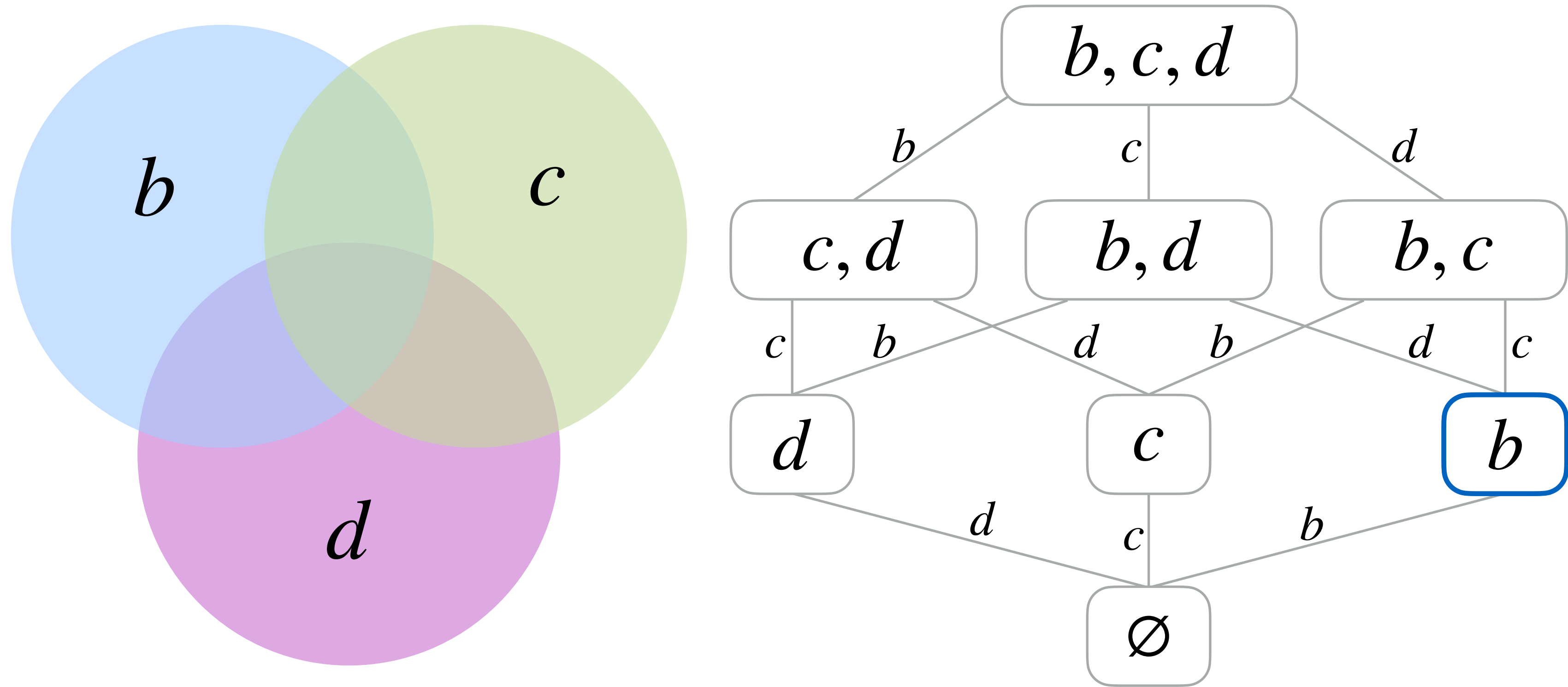

Iteration lattice for a compound expression

$$a_i = b_i + c_i + d_i$$



Iteration lattice for a compound expression

$$a_i = b_i + c_i + d_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = b[pb1] + c[pc1] + d[pd1];
    }
    else if (ib == i && id == i) {
        a[i] = b[pb1] + d[pd1];
    }
    else if (ic == i && id == i) {
        a[i] = c[pc1] + d[pd1];
    }
    else if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    else if (ic == i) {
        a[i] = c[pc1];
    }
    else {
        a[i] = d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] + d[pd1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    else {
        a[i] = d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

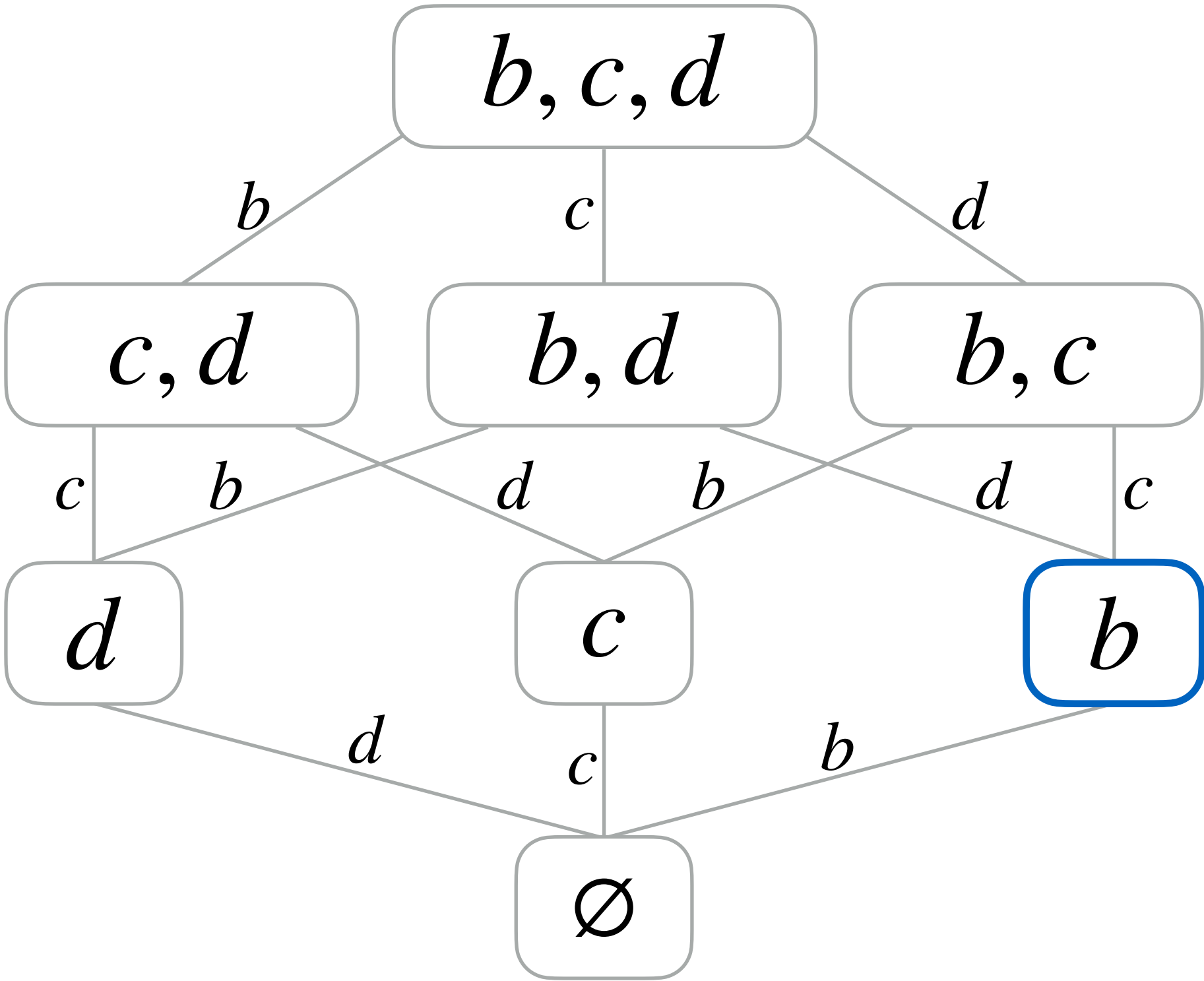
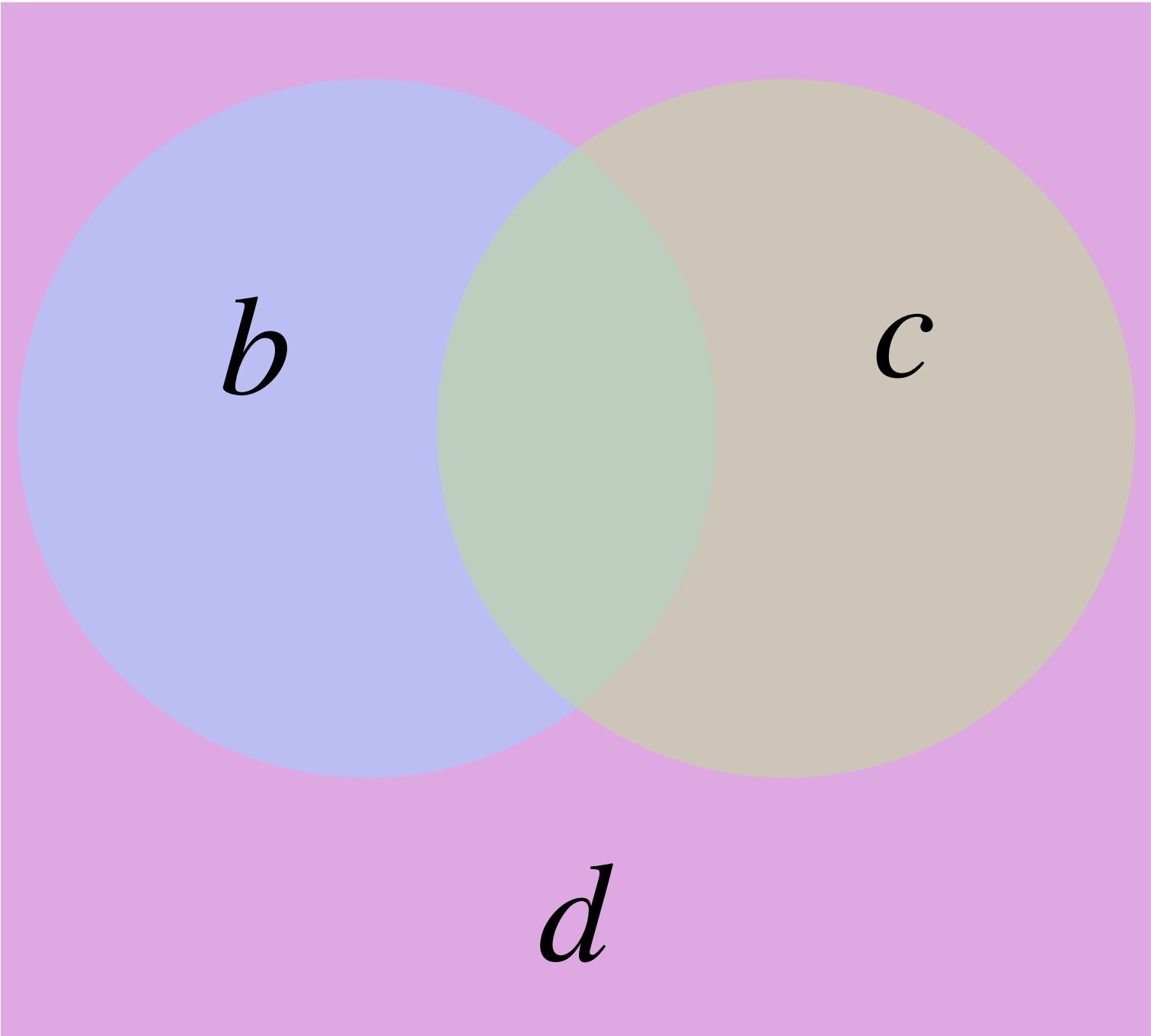
while (pd1 < d1_pos[1]) {
    int id = d1_crd[pd1];
    a[id] = d[pd1];
    pd1++;
}

while (pc1 < c1_pos[1]) {
    int ic = c1_crd[pc1];
    a[ic] = c[pc1];
    pc1++;
}

while (pb1 < b1_pos[1]) {
    int ib = b1_crd[pb1];
    a[ib] = b[pb1];
    pb1++;
}
```

Iteration lattice for a compound expression

$$a_i = b_i + c_i + d_i \leftarrow \text{Dense}$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int id = d1_crd[pd1];
  int i = min(ib, ic, id);
  if (ib == i && ic == i && id == i) {
    a[i] = b[pb1] + c[pc1] + d[pd1];
  }
  else if (ib == i && id == i) {
    a[i] = b[pb1] + d[pd1];
  }
  else if (ic == i && id == i) {
    a[i] = c[pc1] + d[pd1];
  }
  else if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else if (ic == i) {
    a[i] = c[pc1];
  }
  else {
    a[i] = d[pd1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
  if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
  int ib = b1_crd[pb1];
  int id = d1_crd[pd1];
  int i = min(ib, id);
  if (ib == i && id == i) {
    a[i] = b[pb1] + d[pd1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = d[pd1];
  }
  if (ib == i) pb1++;
  if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = c[pc1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}

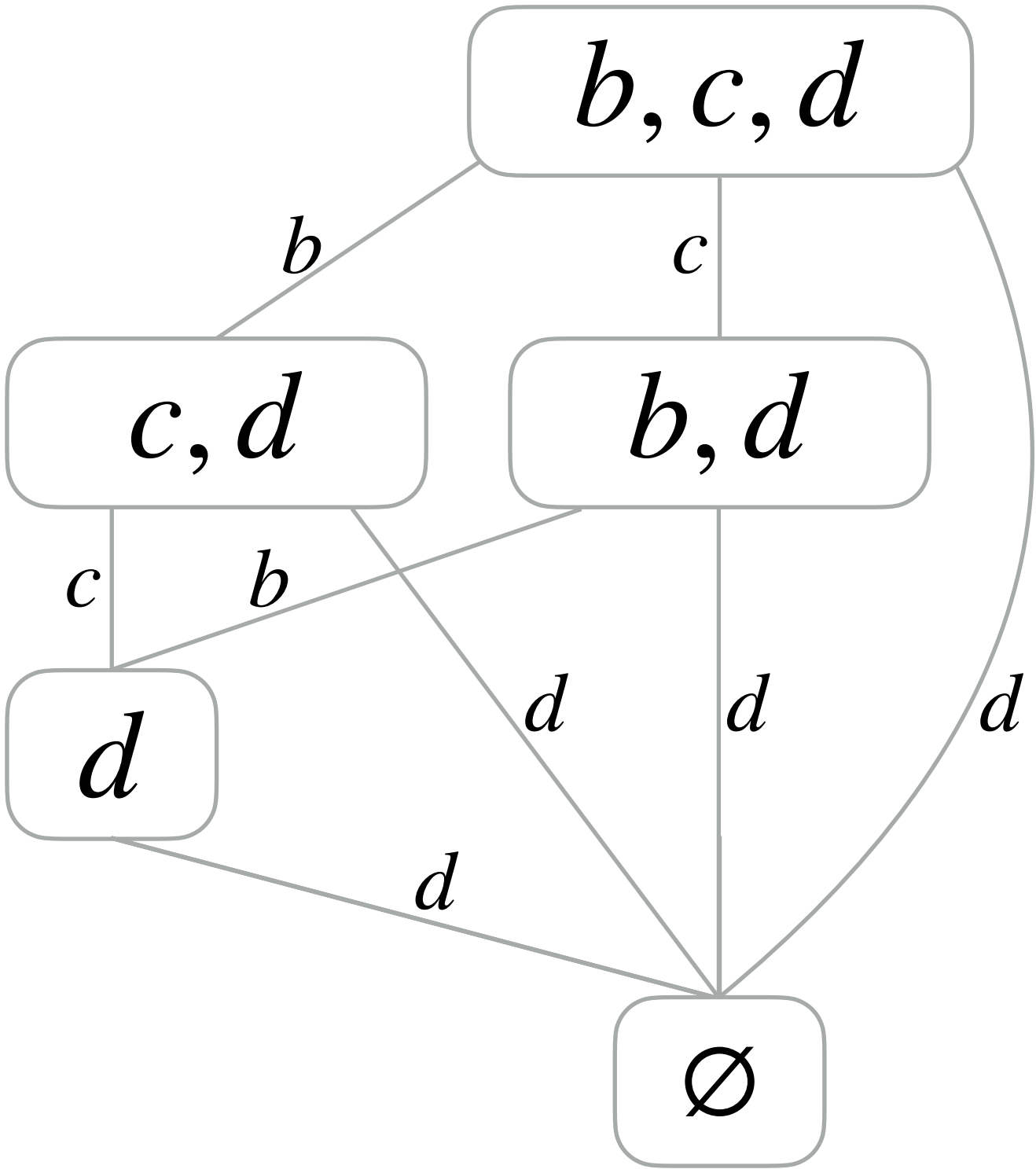
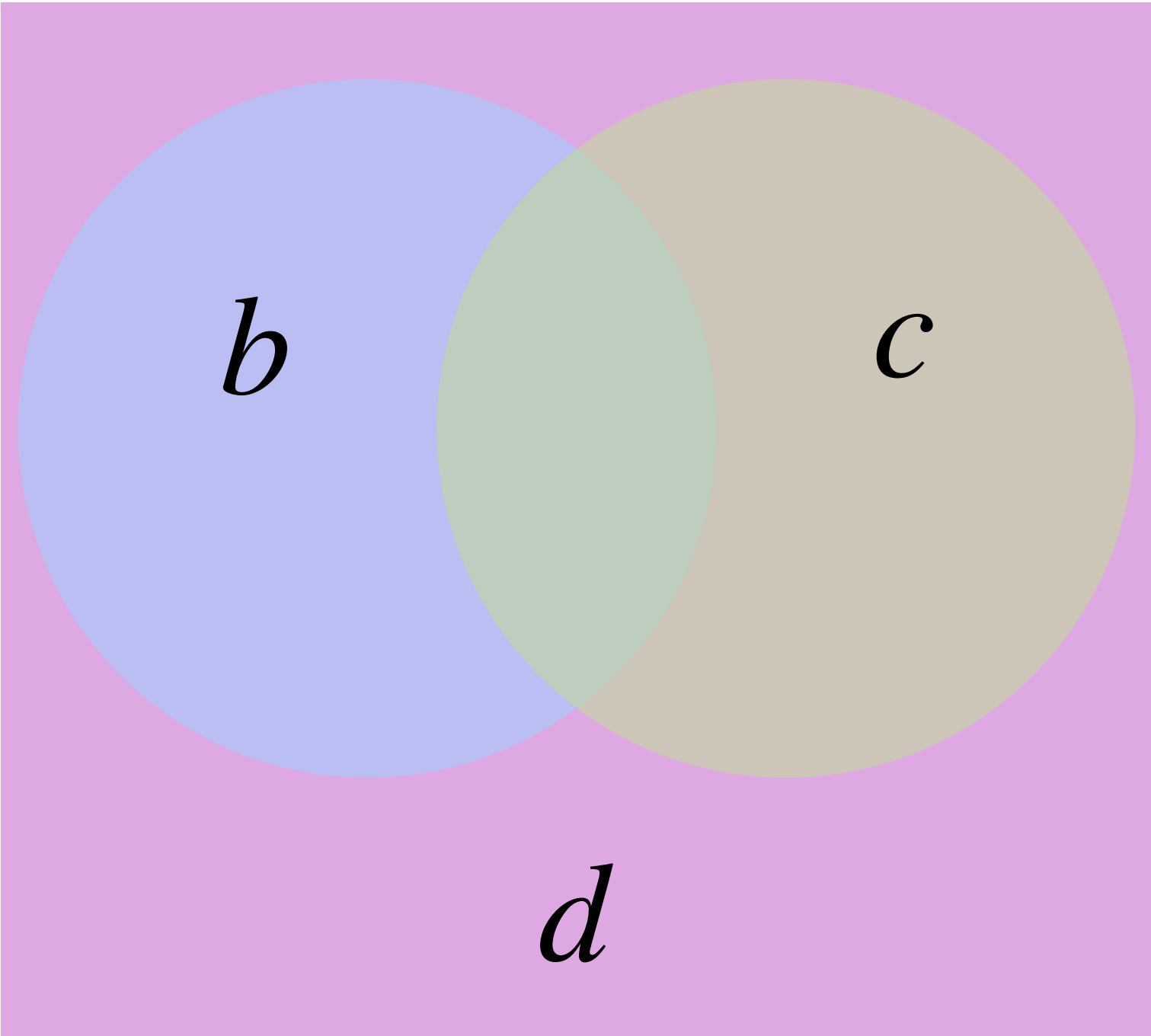
while (pd1 < d1_pos[1]) {
  int id = d1_crd[pd1];
  a[id] = d[pd1];
  pd1++;
}

while (pc1 < c1_pos[1]) {
  int ic = c1_crd[pc1];
  a[ic] = c[pc1];
  pc1++;
}

while (pb1 < b1_pos[1]) {
  int ib = b1_crd[pb1];
  a[ib] = b[pb1];
  pb1++;
}
```


Iteration lattice for a compound expression

$$a_i = b_i + c_i + d_i \leftarrow \text{Dense}$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int id = 0;
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int pd1 = id;
  int pa1 = id;
  if (ib == id && ic == id) {
    a[pa1] = b[pb1] + c[pc1] + d[pd1];
  }
  else if (ib == id) {
    a[pa1] = b[pb1] + d[pd1];
  }
  else if (ic == id) {
    a[pa1] = c[pc1] + d[pd1];
  }
  else {
    a[pa1] = d[pd1];
  }
  if (ib == id) pb1++;
  if (ic == id) pc1++;
  id++;
}
```

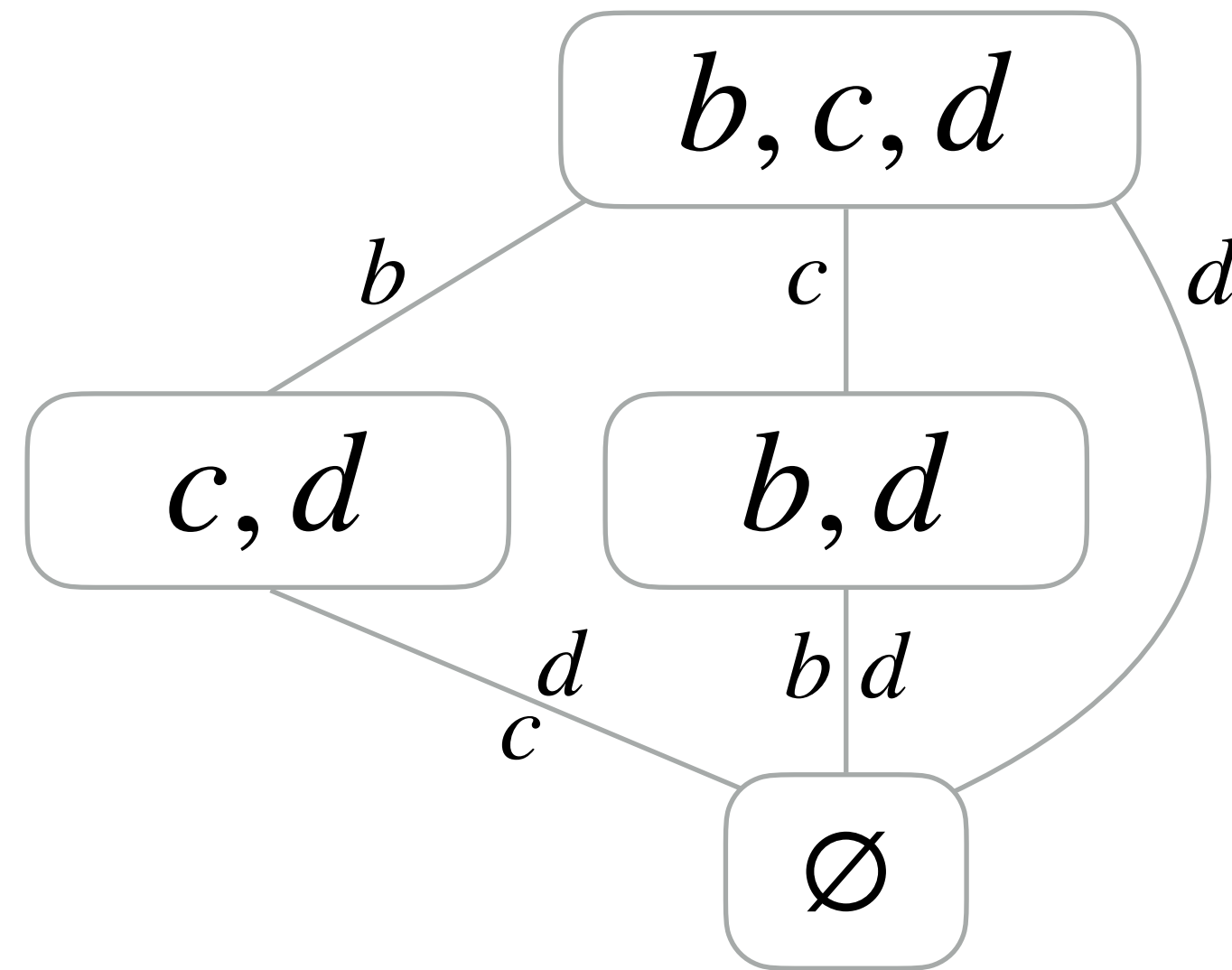
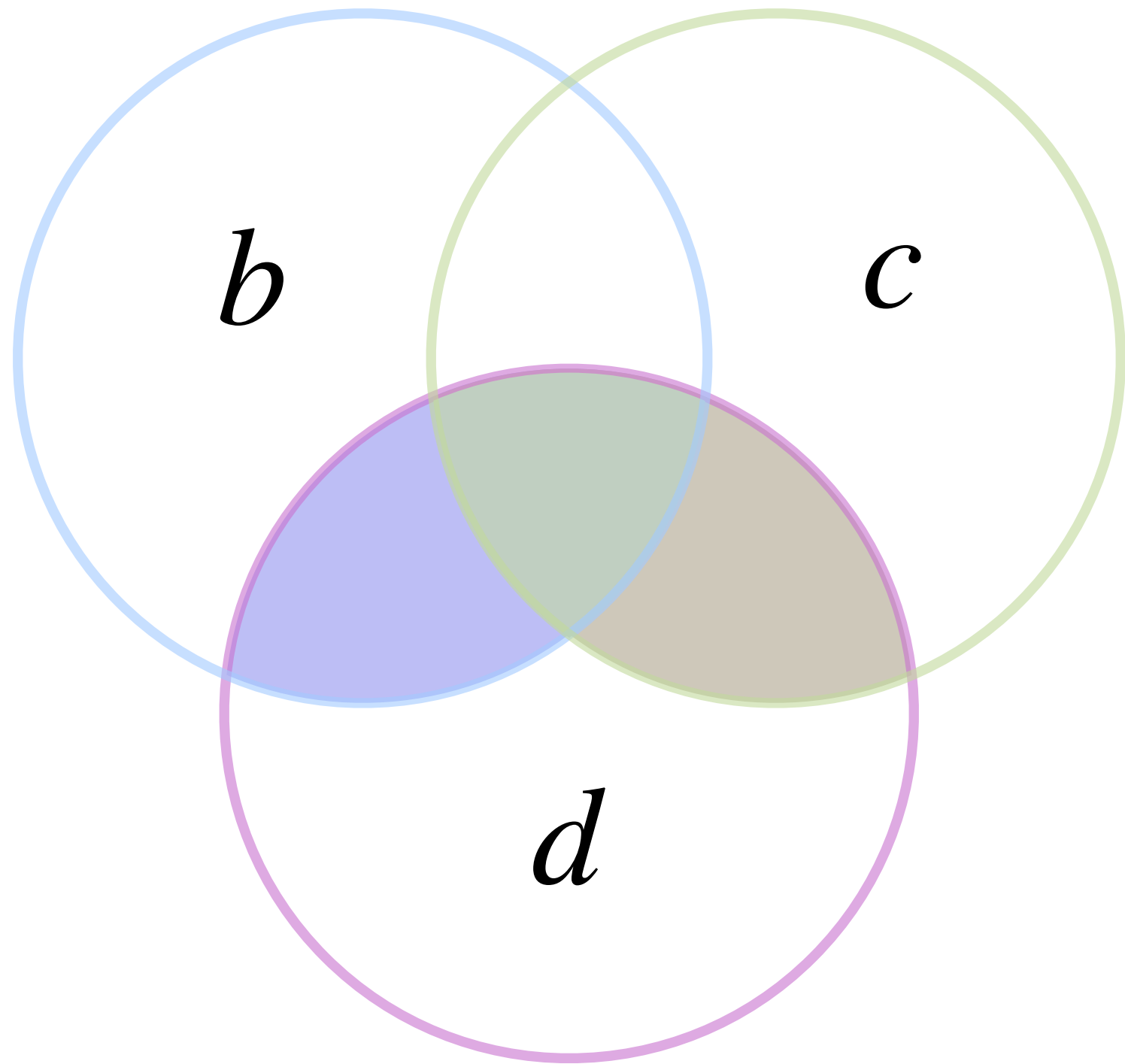
```
while (pc1 < c1_pos[1]) {
  int ic = c1_crd[pc1];
  int pd1 = id;
  int pa1 = id;
  if (ic == id) {
    a[pa1] = c[pc1] + d[pd1];
  }
  else {
    a[pa1] = d[pd1];
  }
  if (ic == id) pc1++;
  id++;
}
```

```
while (pb1 < b1_pos[1]) {
  int ib = b1_crd[pb1];
  int pd1 = id;
  int pa1 = id;
  if (ib == id) {
    a[pa1] = b[pb1] + d[pd1];
  }
  else {
    a[pa1] = d[pd1];
  }
  if (ib == id) pb1++;
  id++;
}
```

```
while (id < d1_dimension) {
  int pd1 = id;
  int pa1 = id;
  a[pa1] = d[pd1];
  id++;
}
```

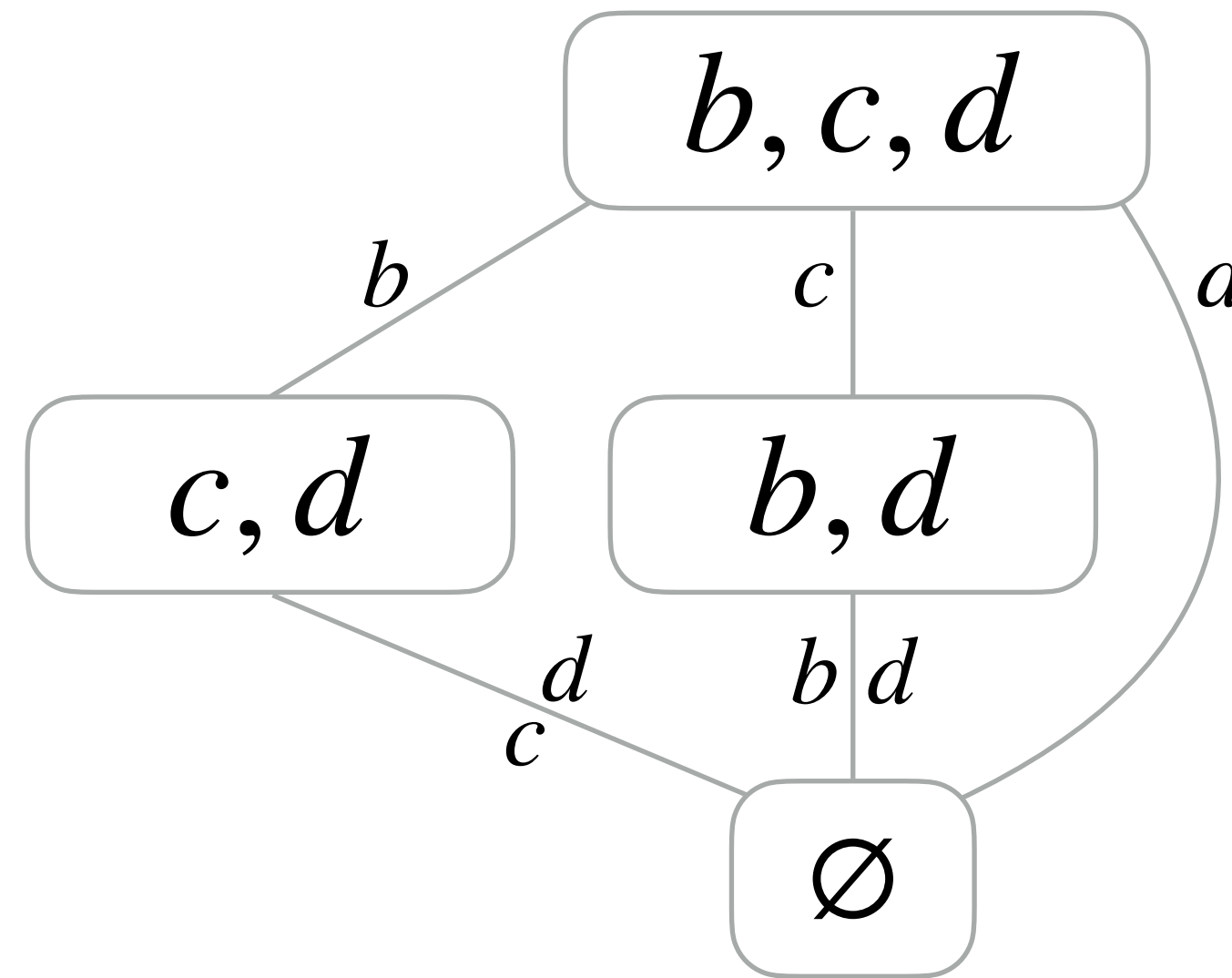
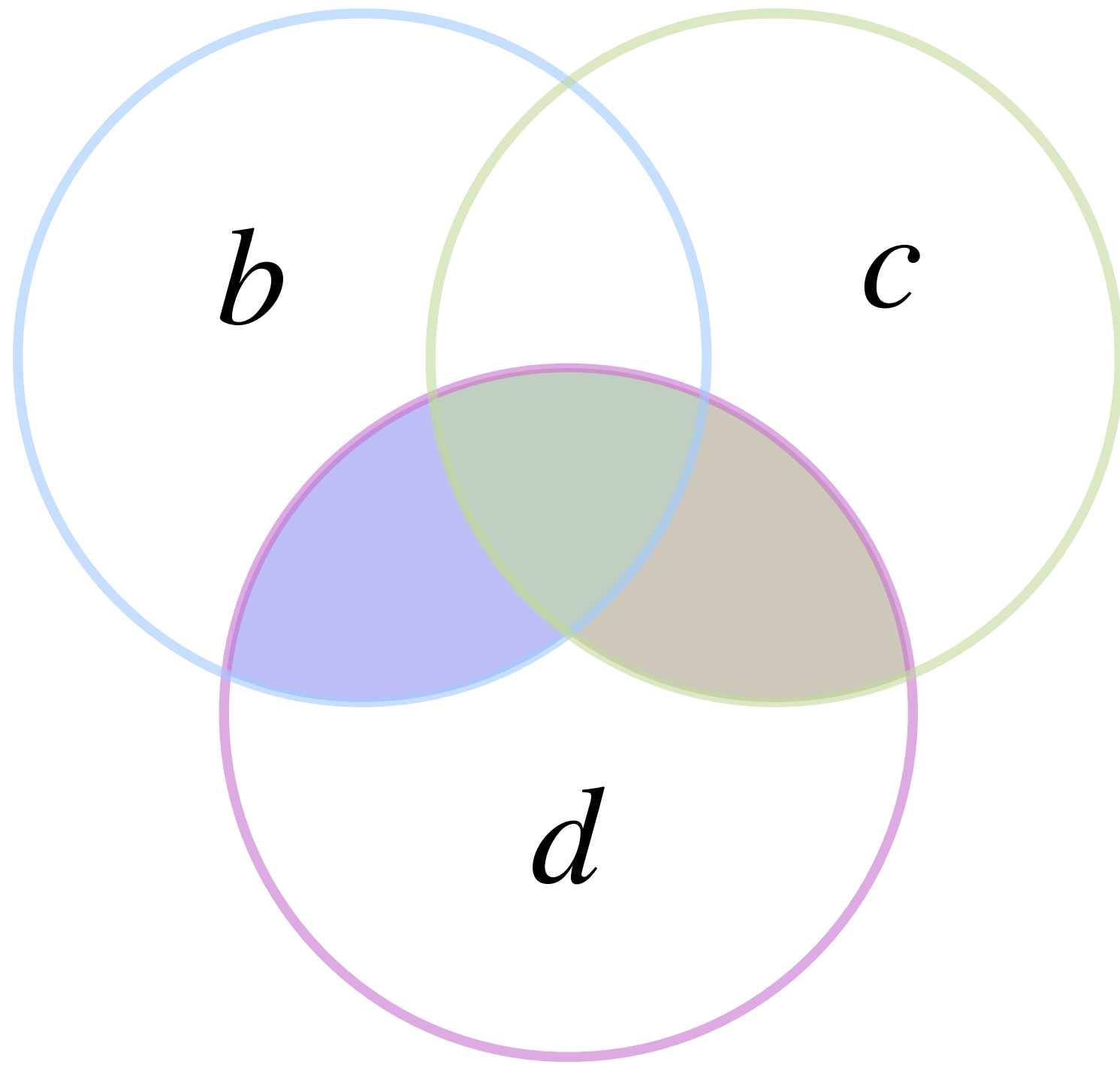
Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i$$



Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i$$



```

int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    }
    else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

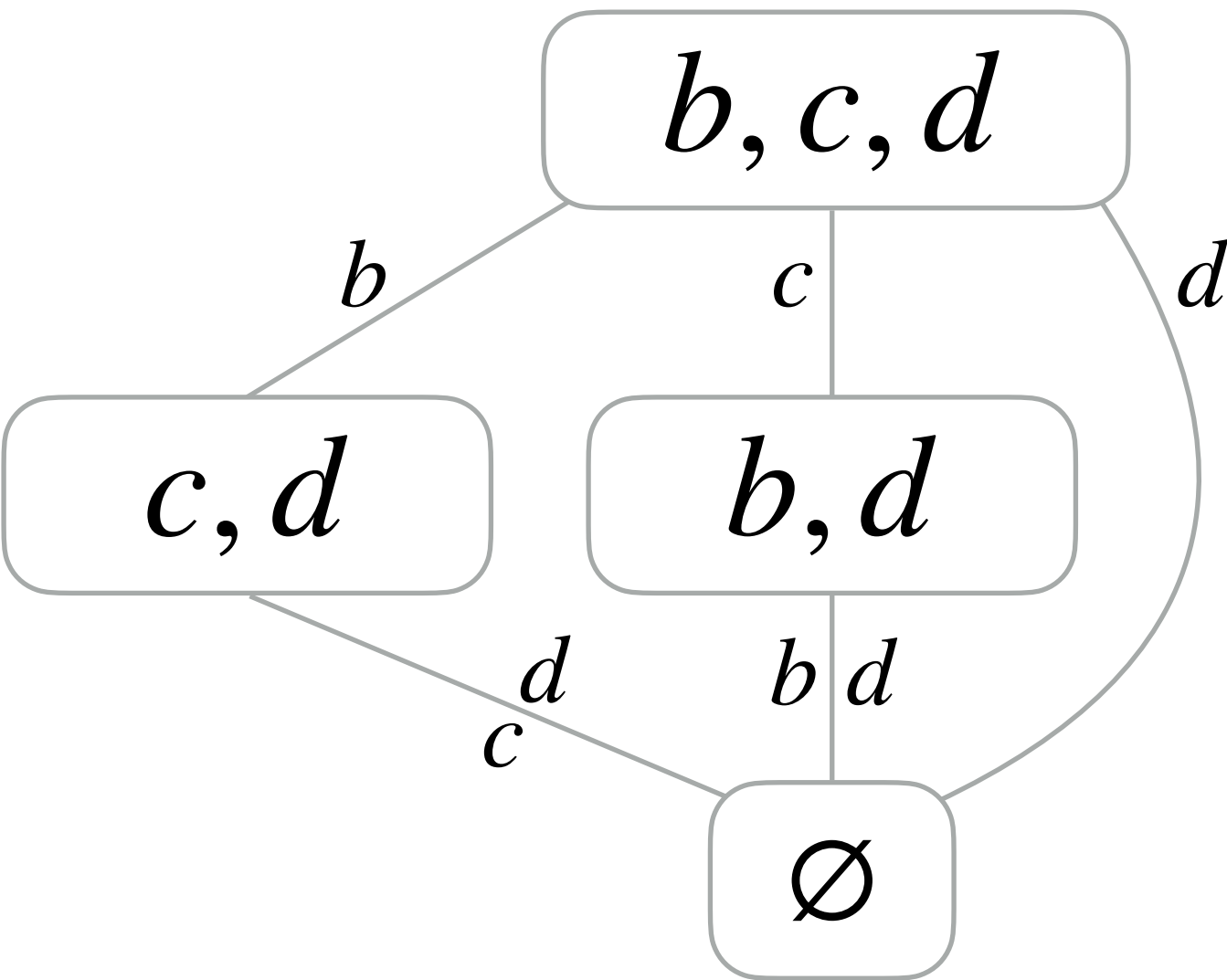
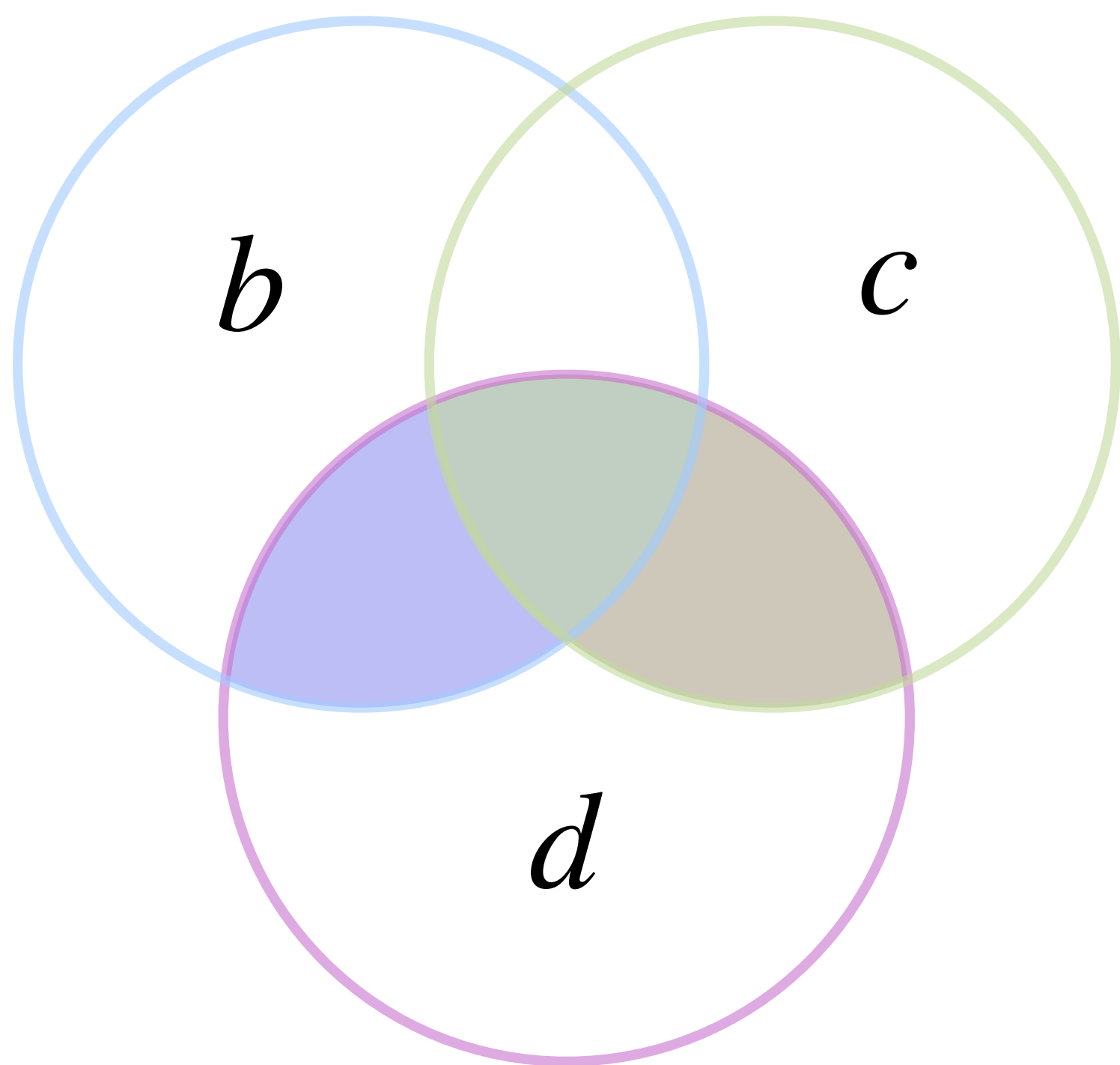
while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}

```


Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \leftarrow \text{Dense}$$



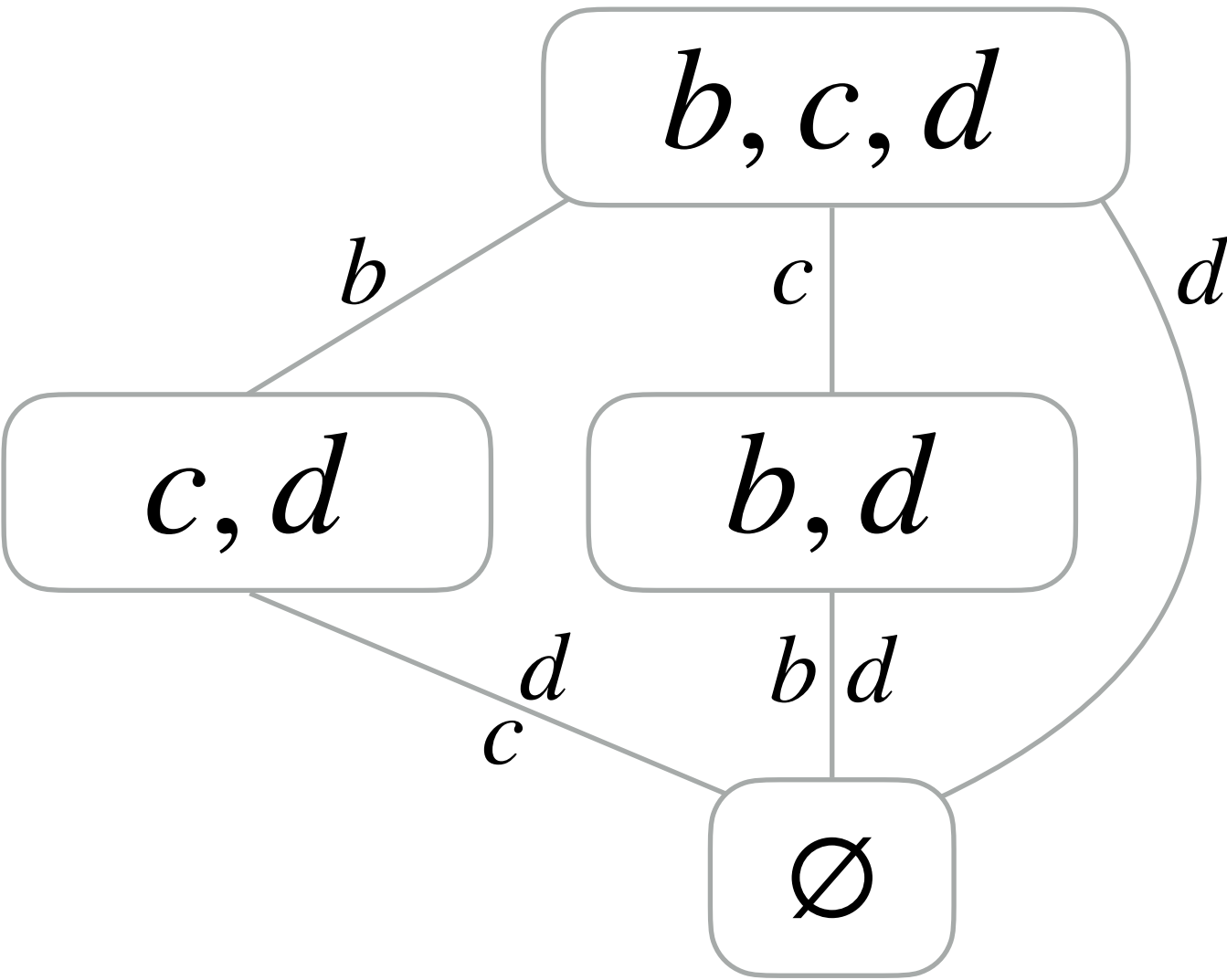
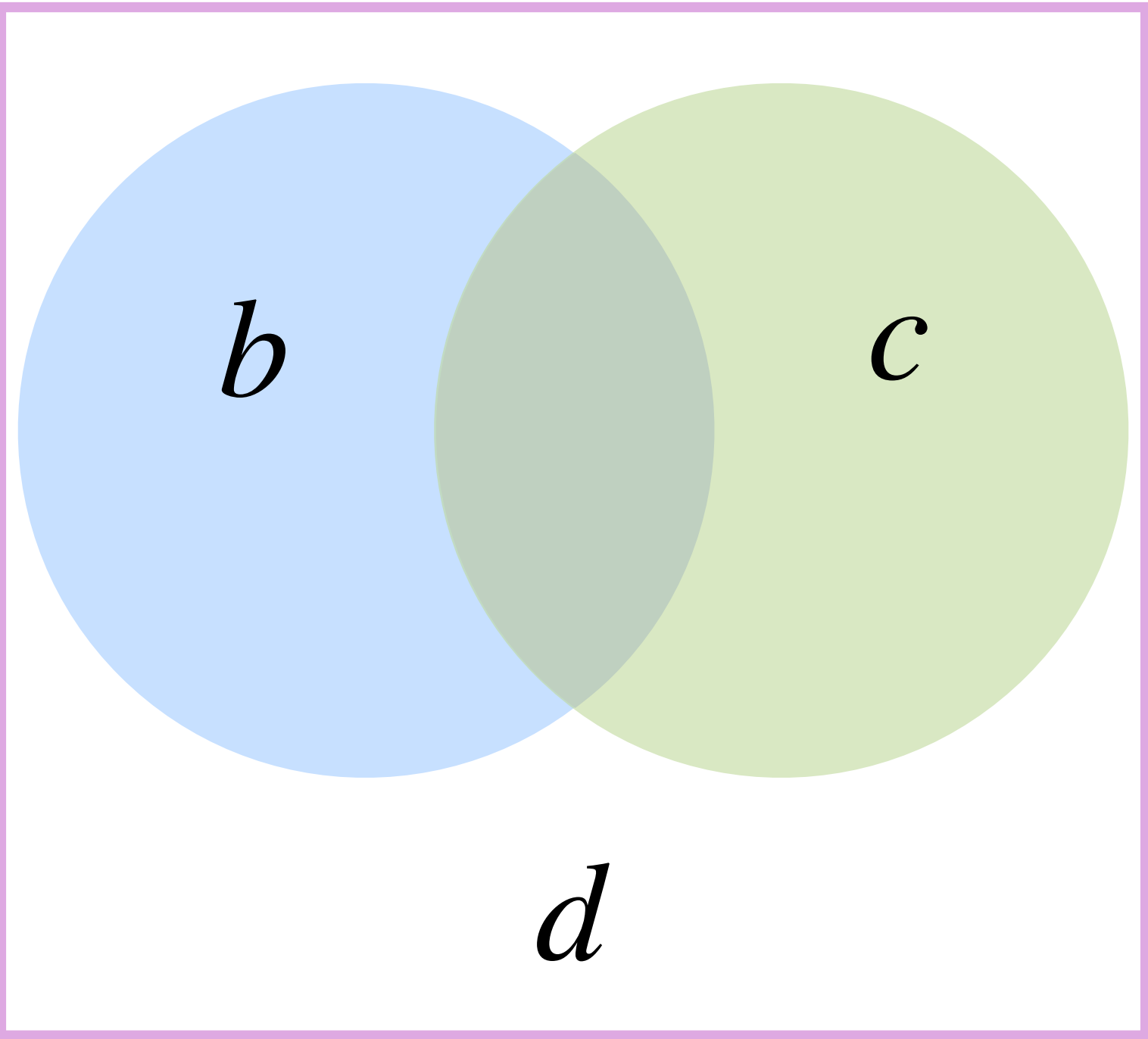
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    }
    else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}
```

Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \leftarrow \text{Dense}$$



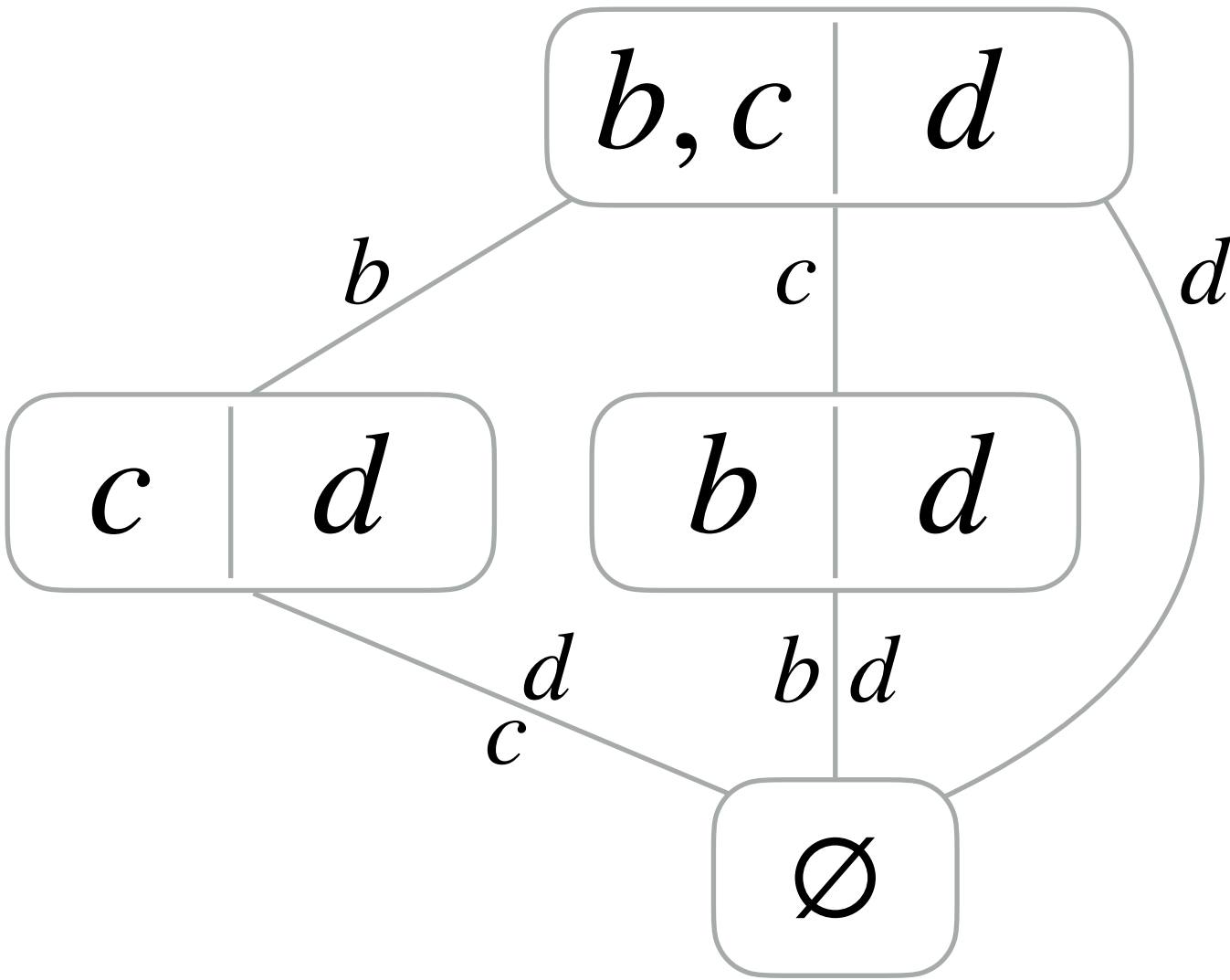
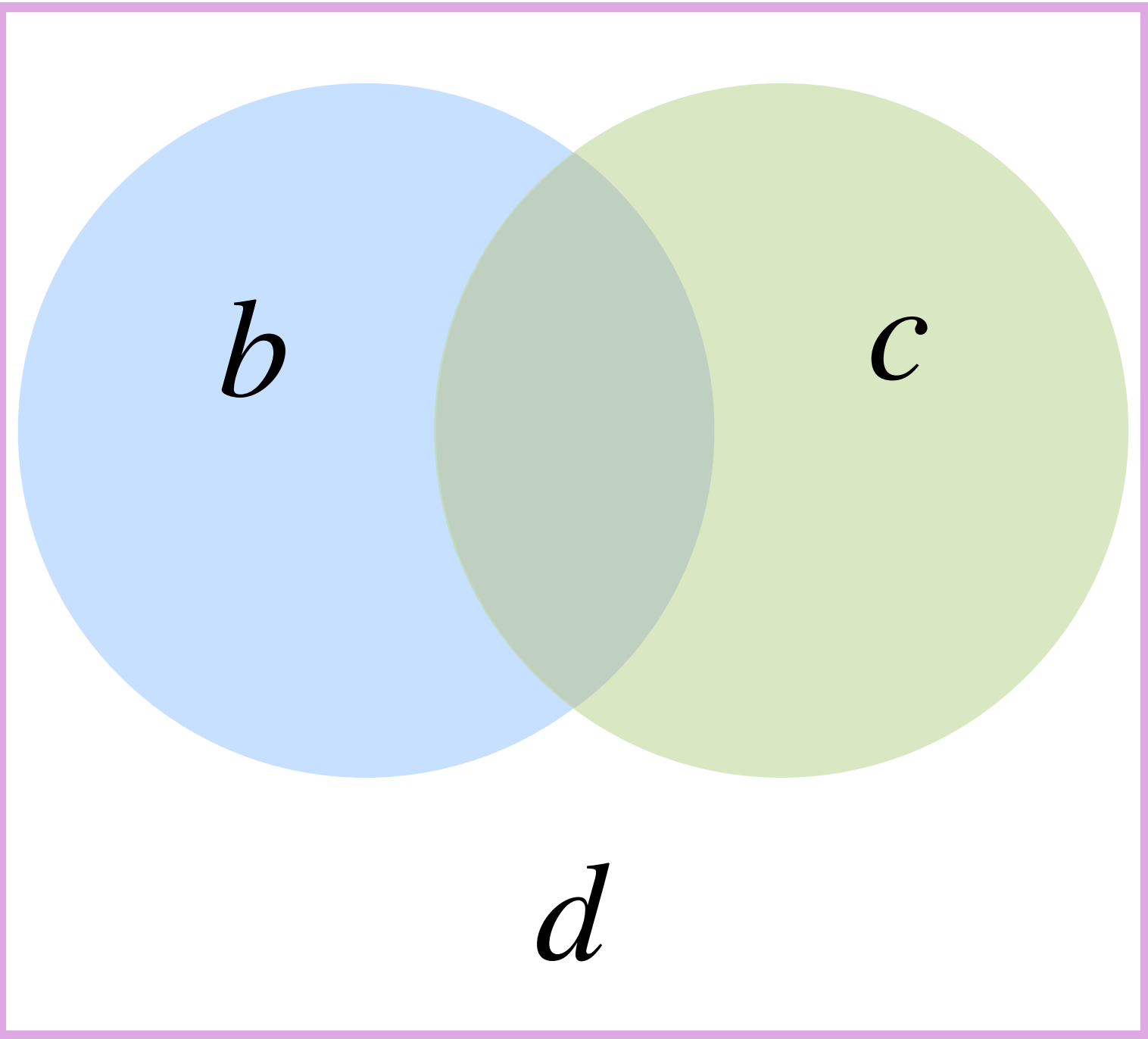
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int id = d1_crd[pd1];
  int i = min(ib, ic, id);
  if (ib == i && ic == i && id == i) {
    a[i] = (b[pb1] + c[pc1]) * d[pd1];
  }
  else if (ib == i && id == i) {
    a[i] = b[pb1] * d[pd1];
  }
  else if (ic == i && id == i) {
    a[i] = c[pc1] * d[pd1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
  if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
  int ic = c1_crd[pc1];
  int id = d1_crd[pd1];
  int i = min(ic, id);
  if (ic == i && id == i) {
    a[i] = c[pc1] * d[pd1];
  }
  if (ic == i) pc1++;
  if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
  int ib = b1_crd[pb1];
  int id = d1_crd[pd1];
  int i = min(ib, id);
  if (ib == i && id == i) {
    a[i] = b[pb1] * d[pd1];
  }
  if (ib == i) pb1++;
  if (id == i) pd1++;
}
```


Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \leftarrow \text{Dense}$$



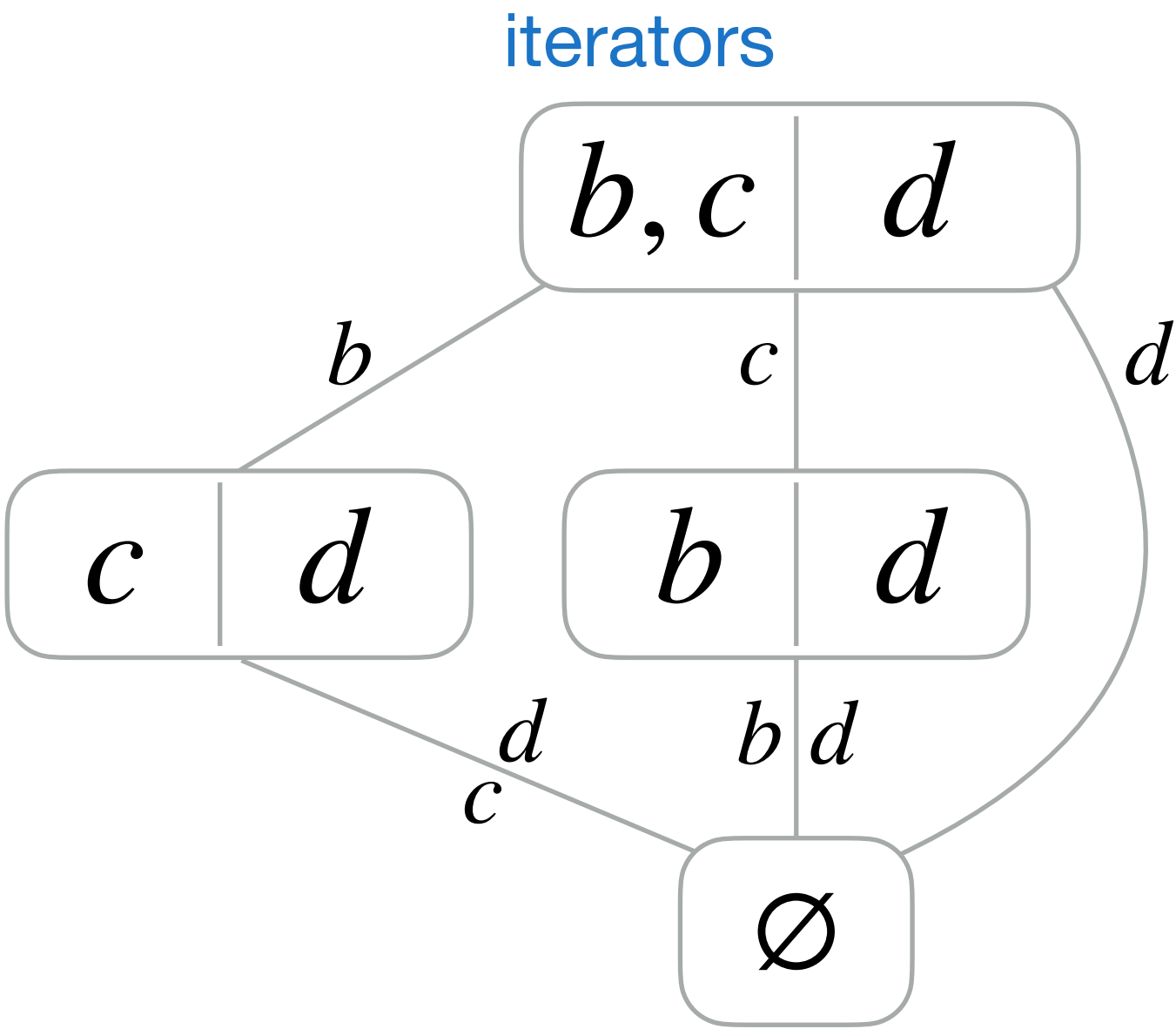
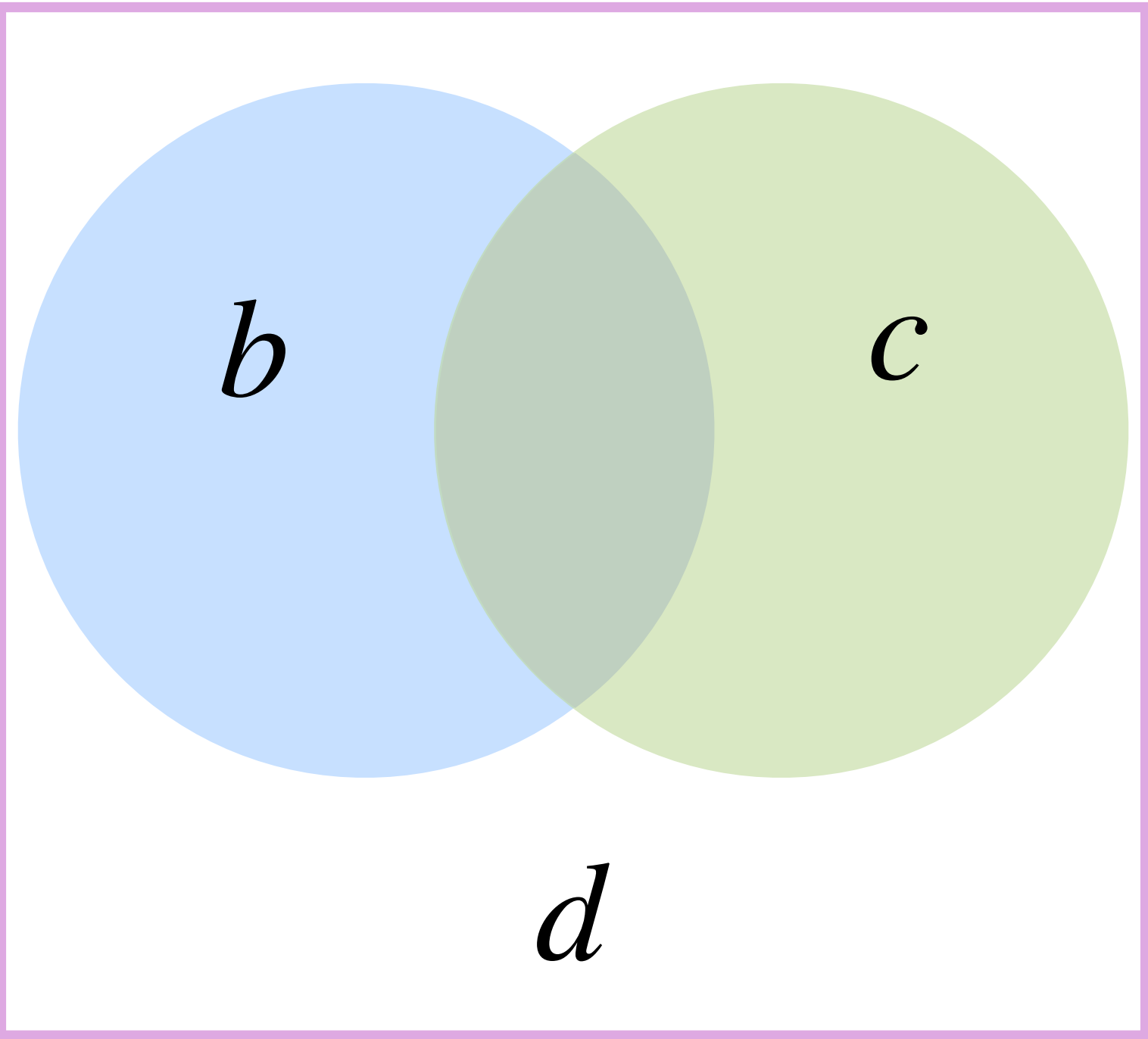
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    }
    else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}
```

Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \leftarrow \text{Dense}$$



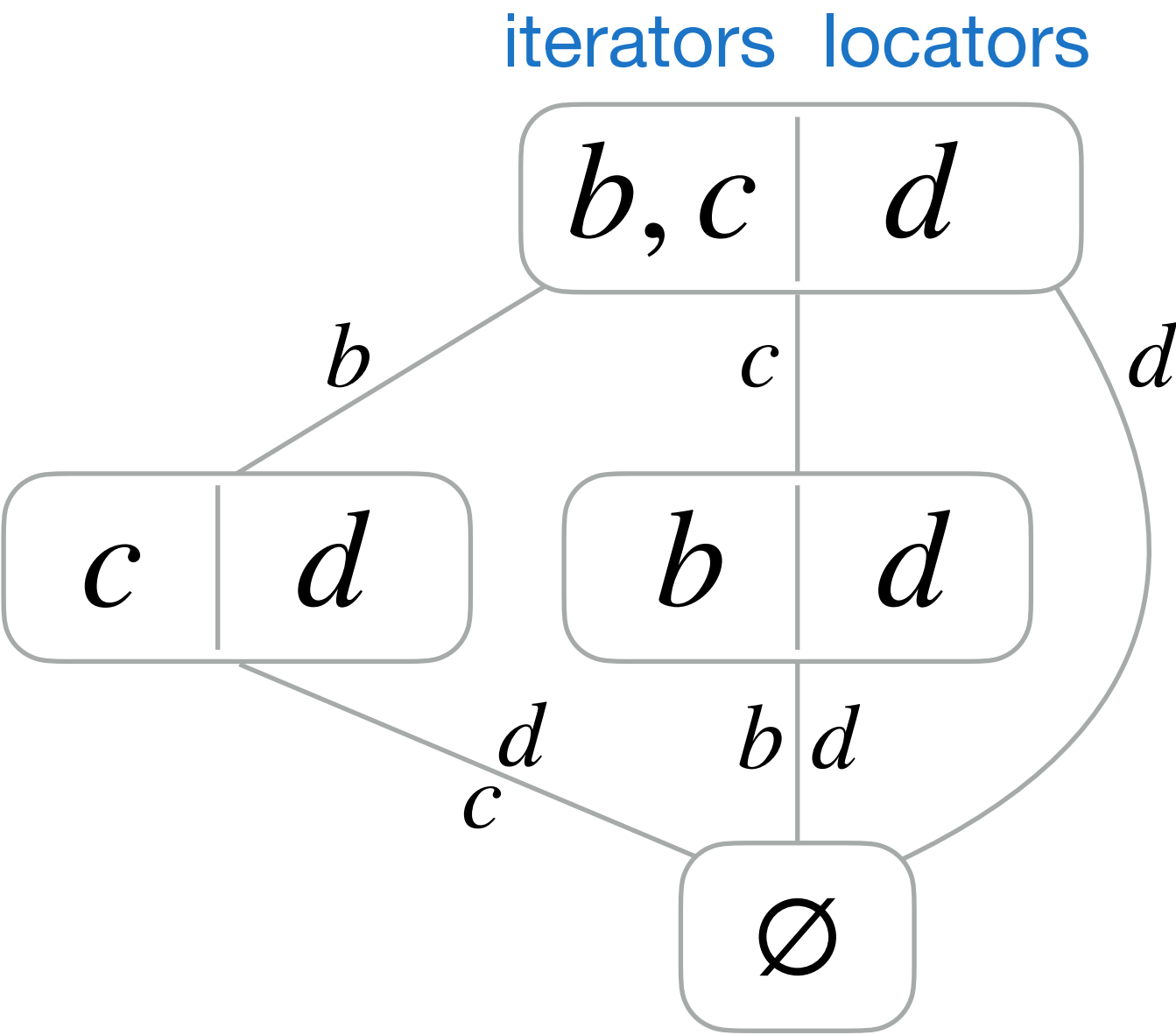
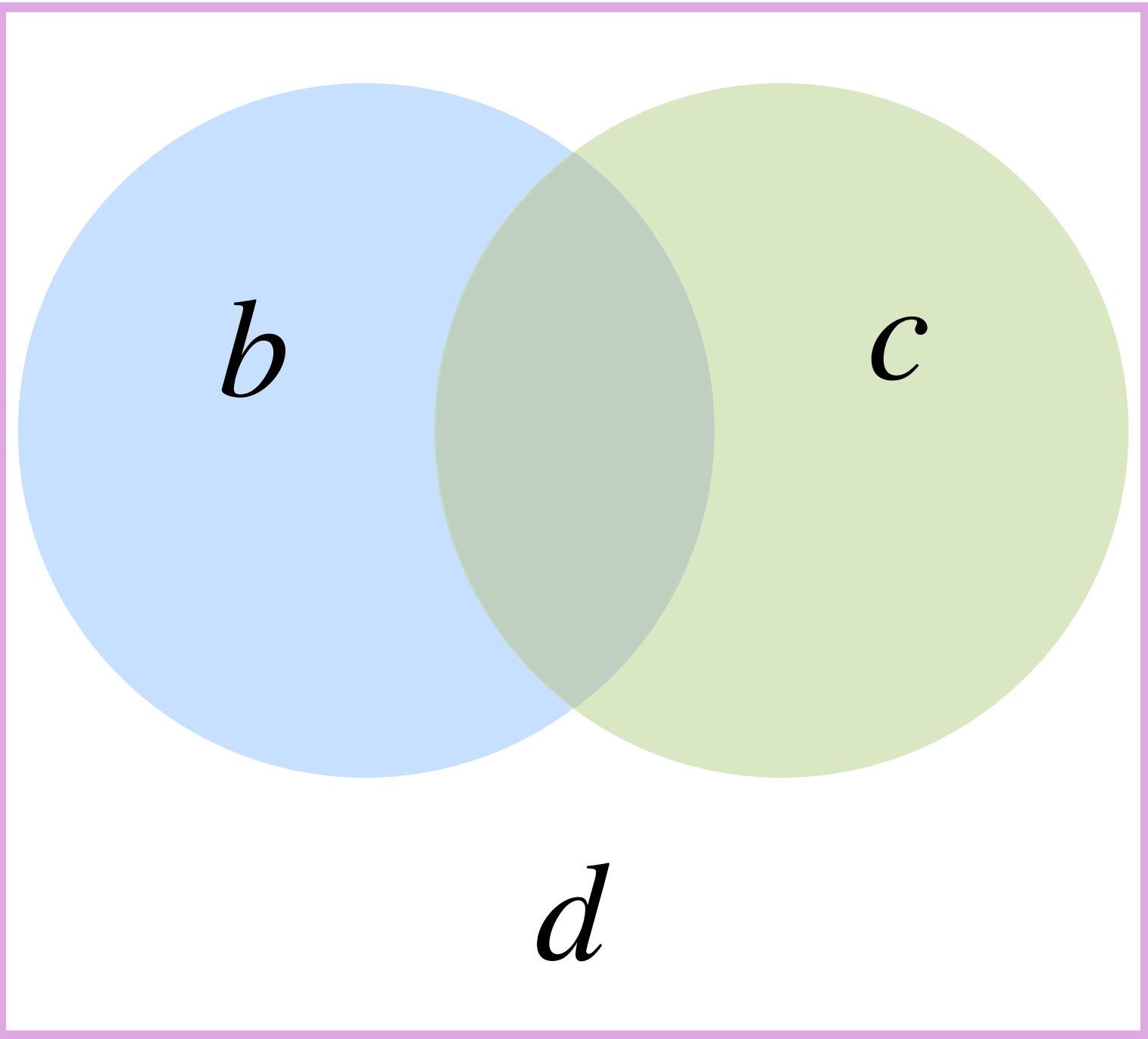
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    }
    else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}
```

Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \leftarrow \text{Dense}$$



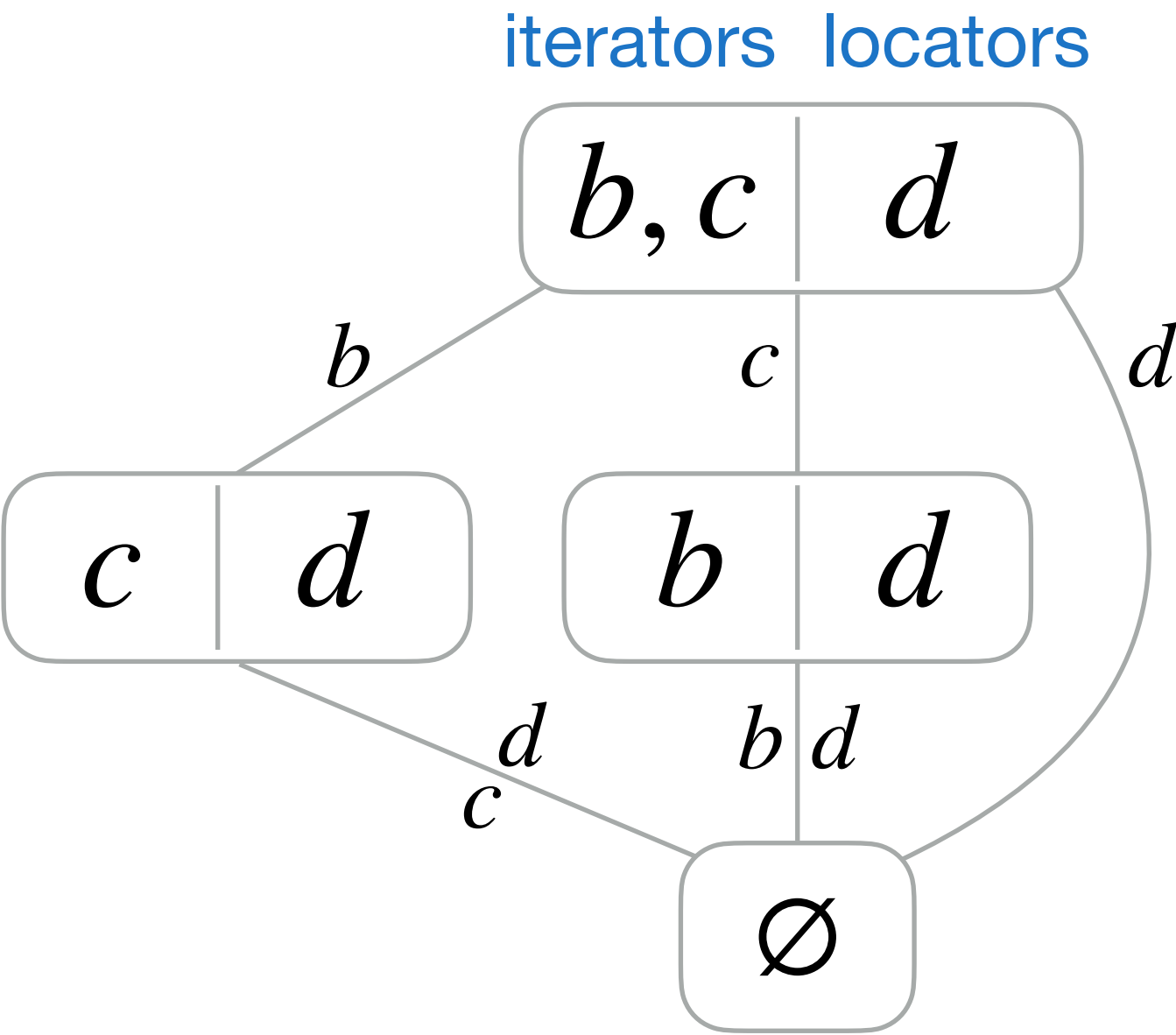
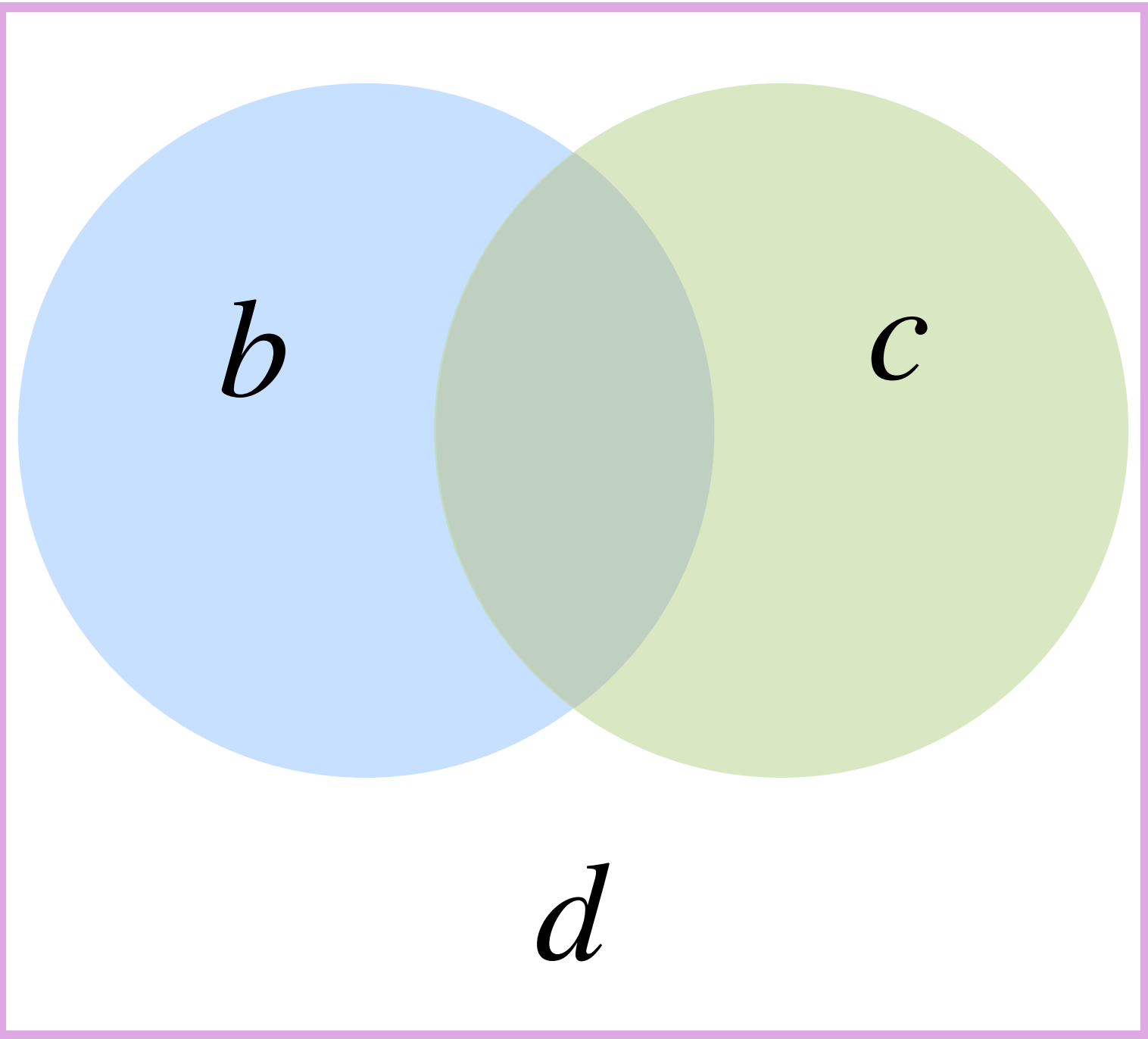
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    }
    else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}
```


Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \leftarrow \text{Dense}$$

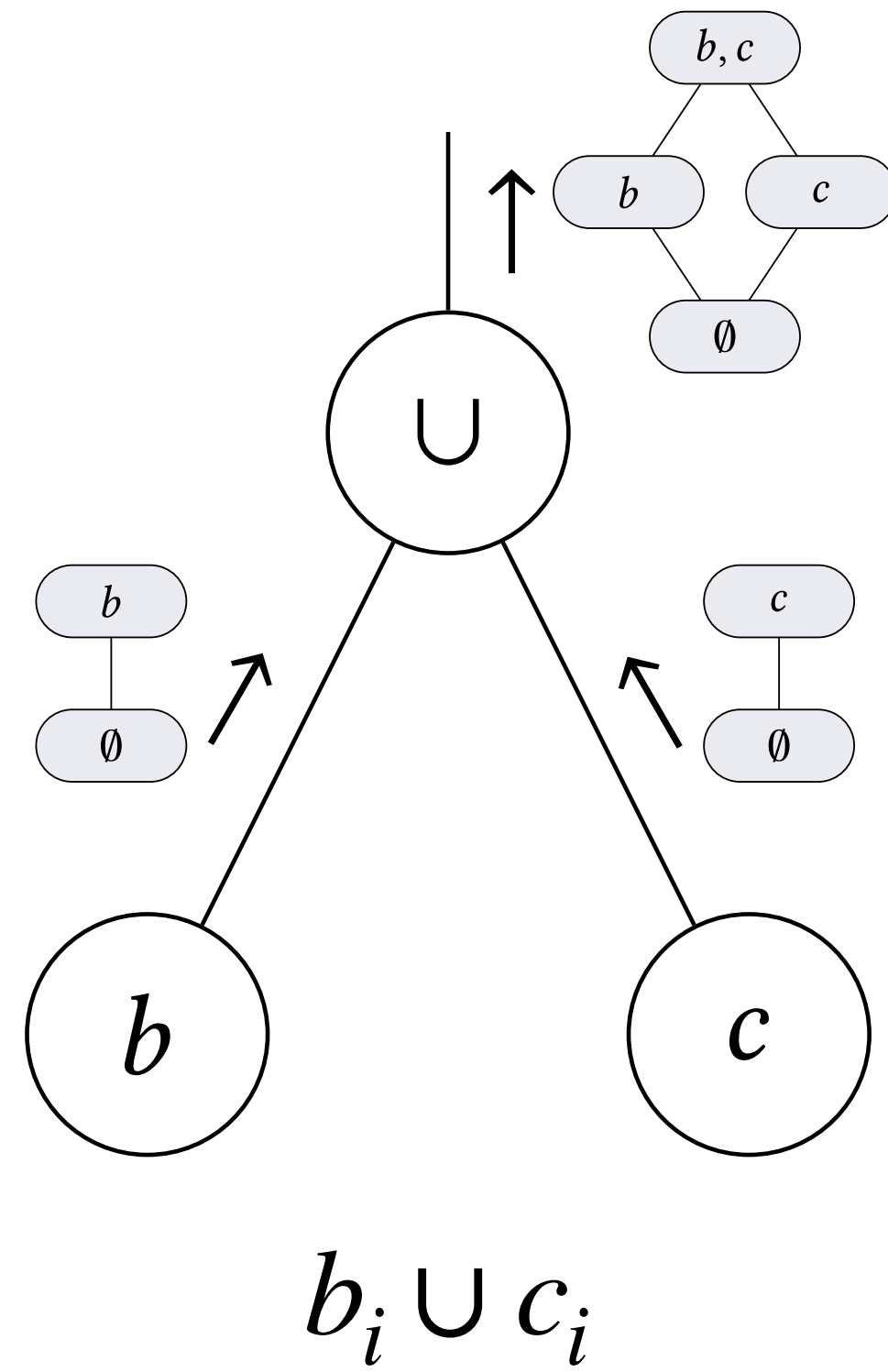


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = (b[pb1] + c[pc1]) * d[i];
  }
  else if (ib == i) {
    a[i] = b[pb1] * d[i];
  }
  else if (ic == i) {
    a[i] = c[pc1] * d[i];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}

while (pb1 < b1_pos[1]) {
  int i = b1_crd[pb1];
  a[i] = b[pb1] * d[i];
  pb1++;
}

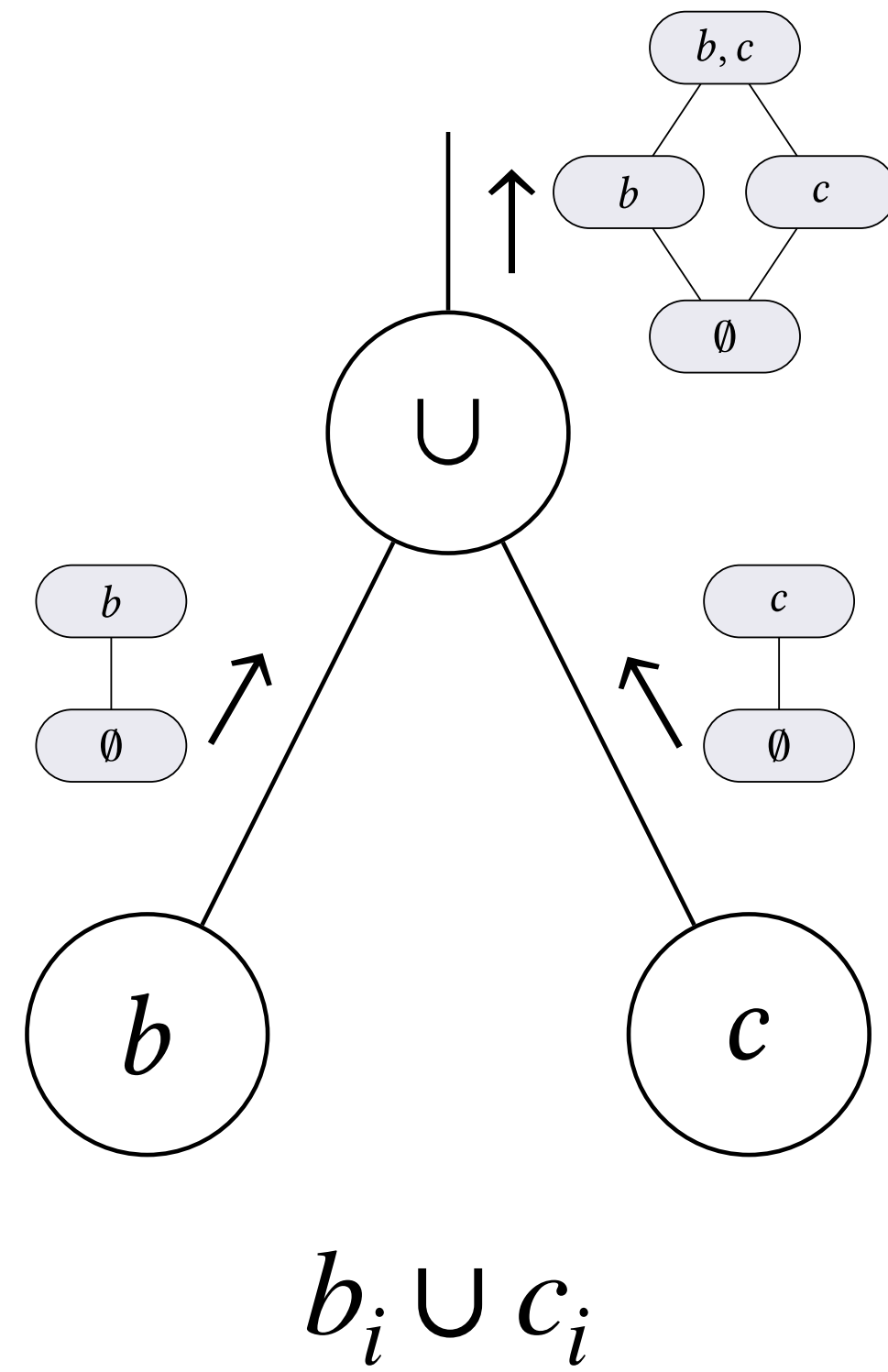
while (pc1 < c1_pos[1]) {
  int i = c1_crd[pc1];
  a[i] = c[pc1] * d[i];
  pc1++;
}
```

Iteration lattice construction



Bottom-up construction from set expression:
create and merge iteration lattices

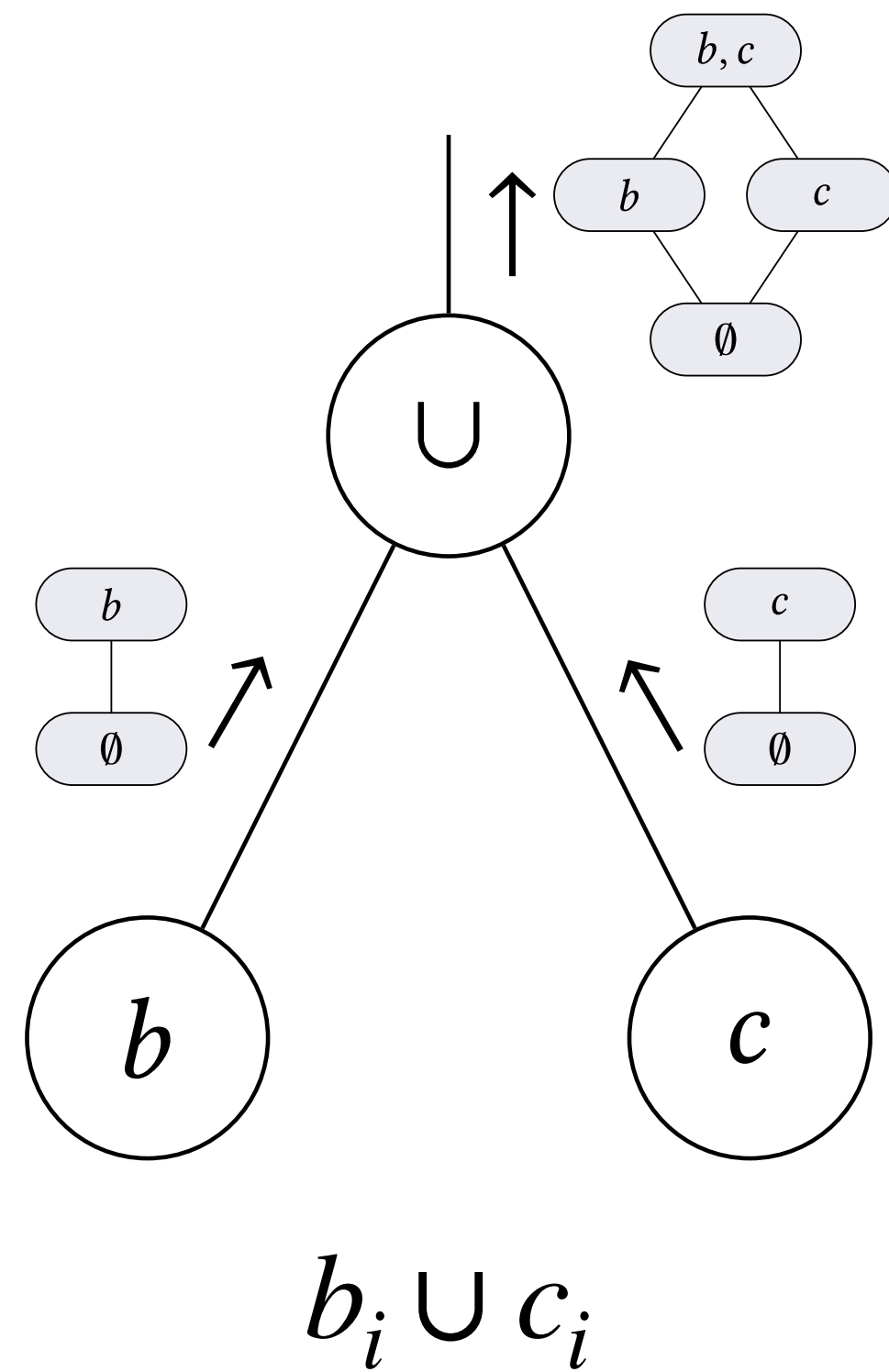
Iteration lattice construction



Base cases:

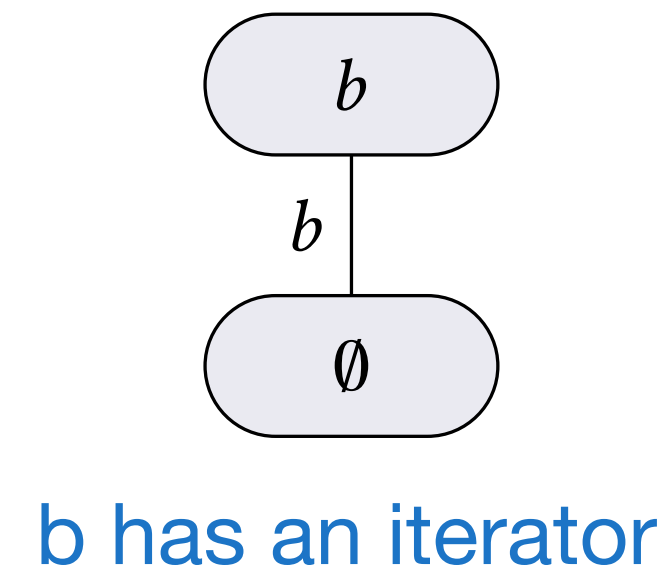
Bottom-up construction from set expression:
create and merge iteration lattices

Iteration lattice construction

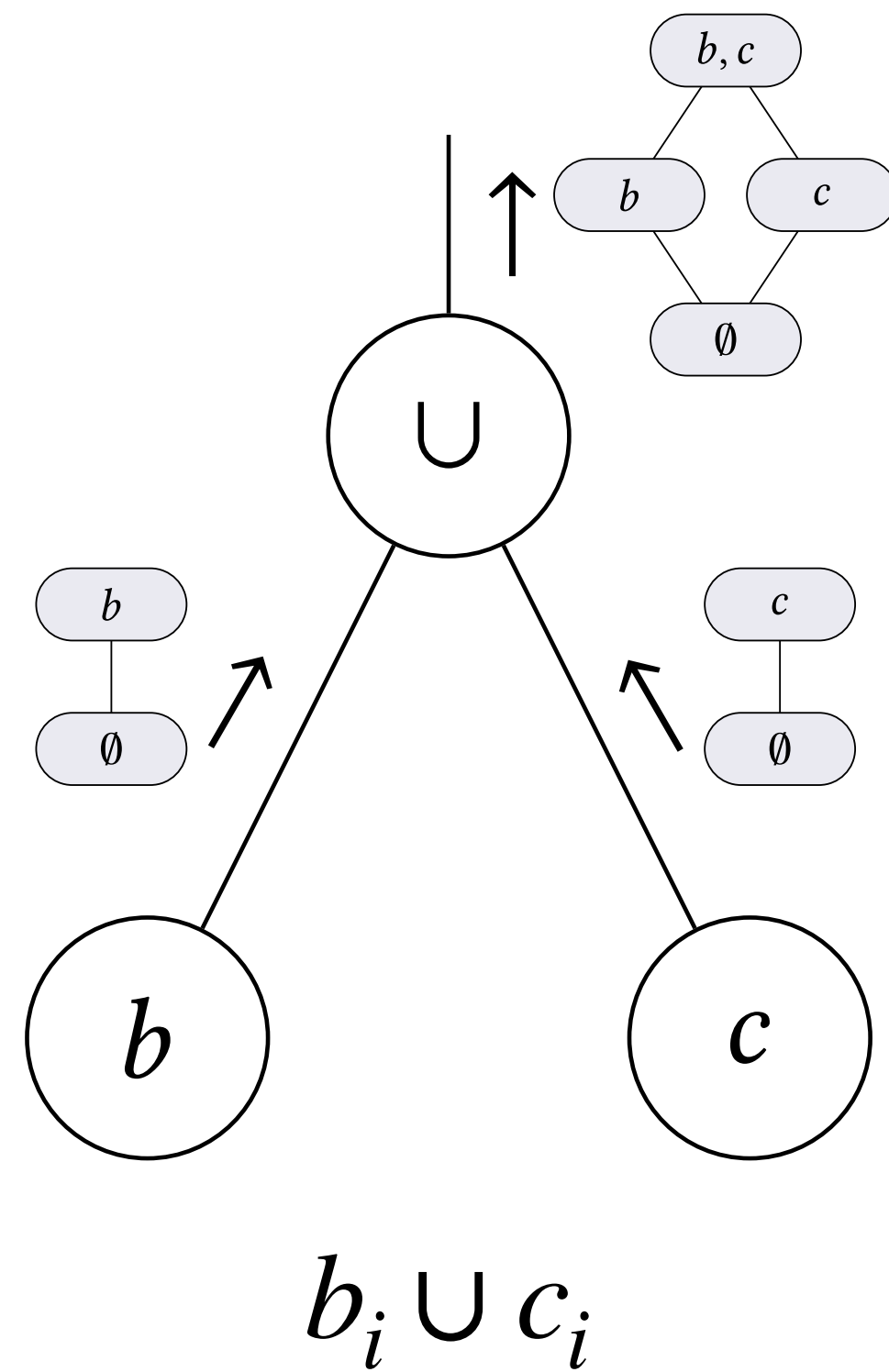


Bottom-up construction from set expression:
create and merge iteration lattices

Base cases:

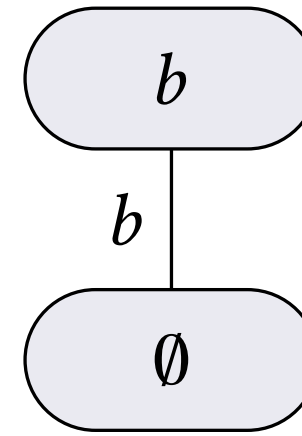


Iteration lattice construction

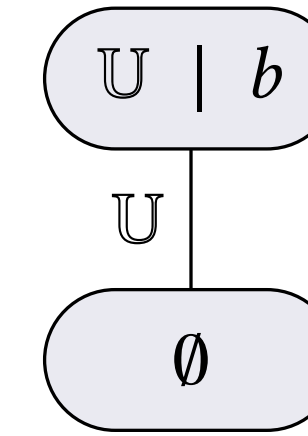


Bottom-up construction from set expression:
create and merge iteration lattices

Base cases:

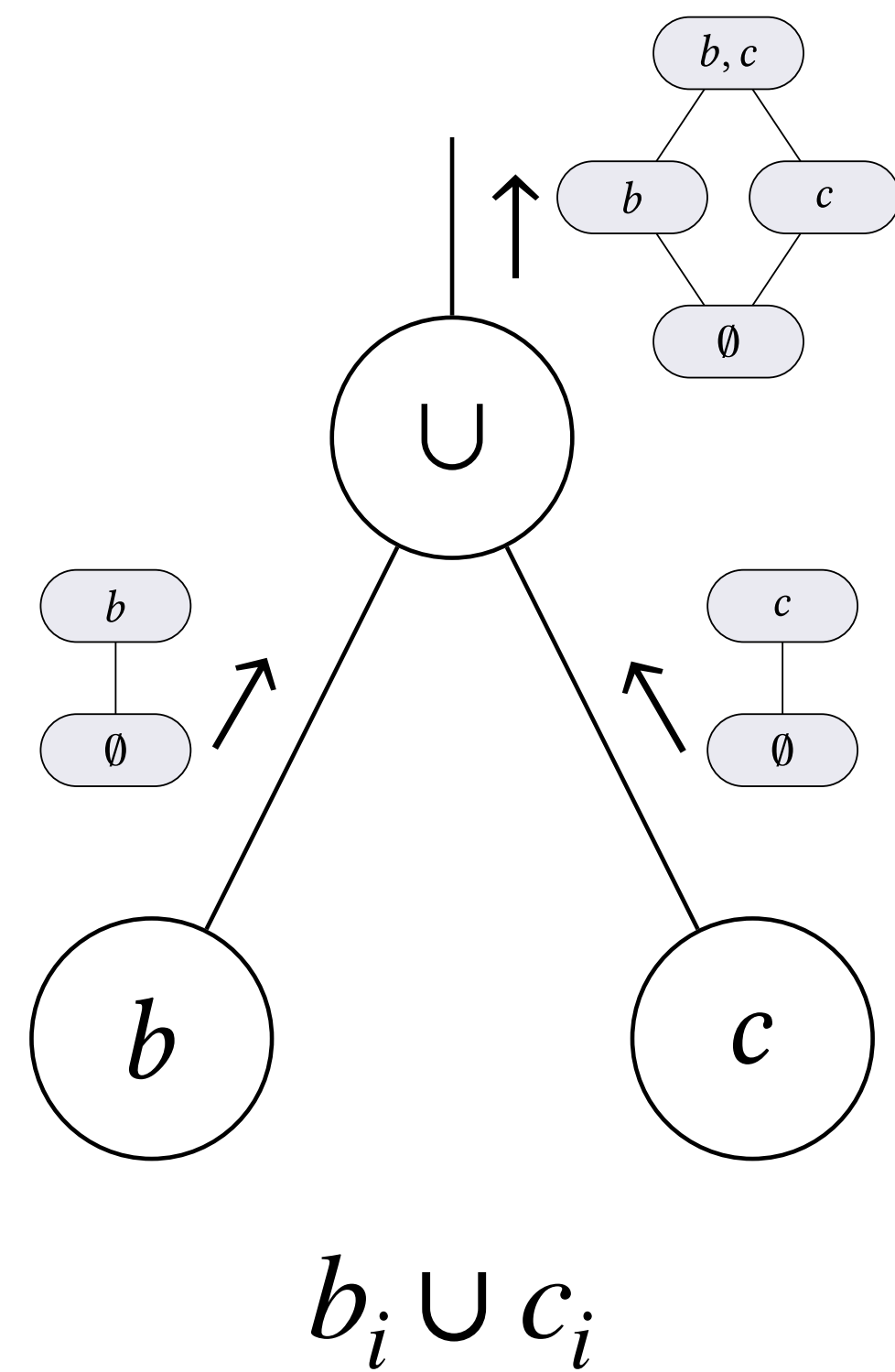


b has an iterator



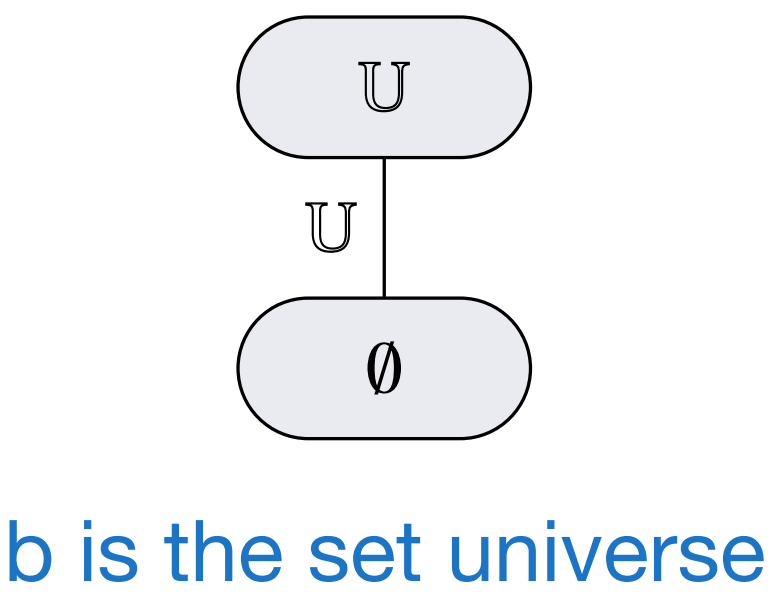
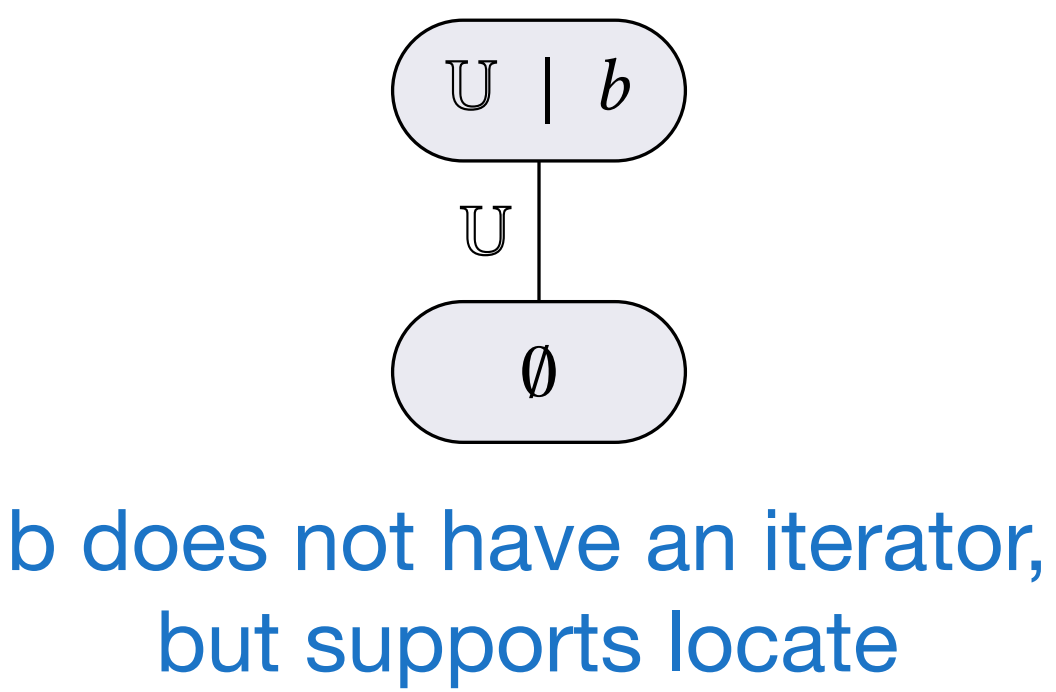
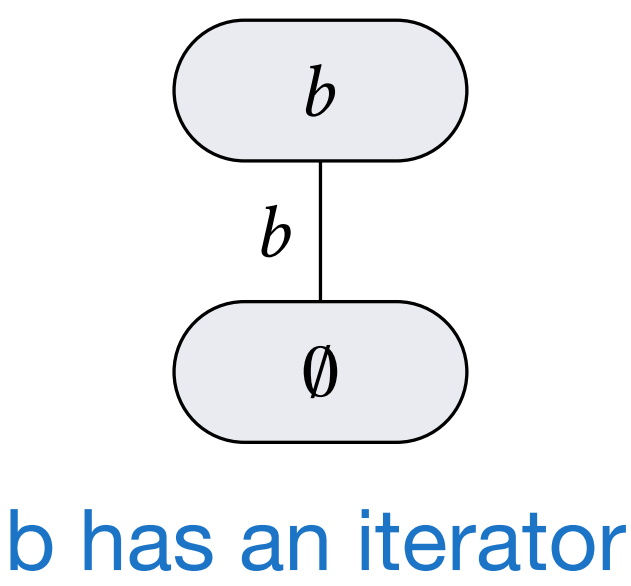
b does not have an iterator,
but supports locate

Iteration lattice construction

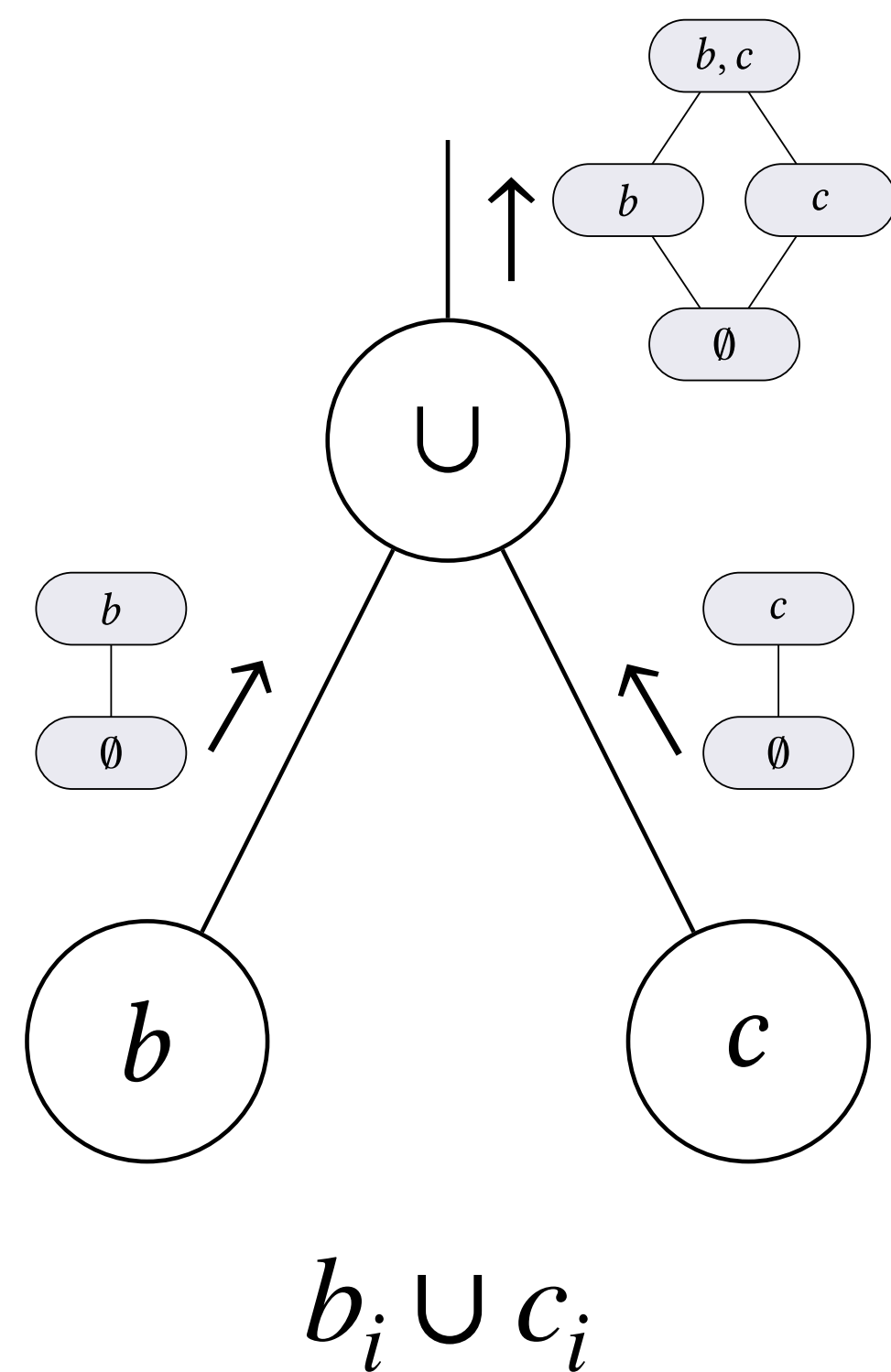


Bottom-up construction from set expression:
create and merge iteration lattices

Base cases:

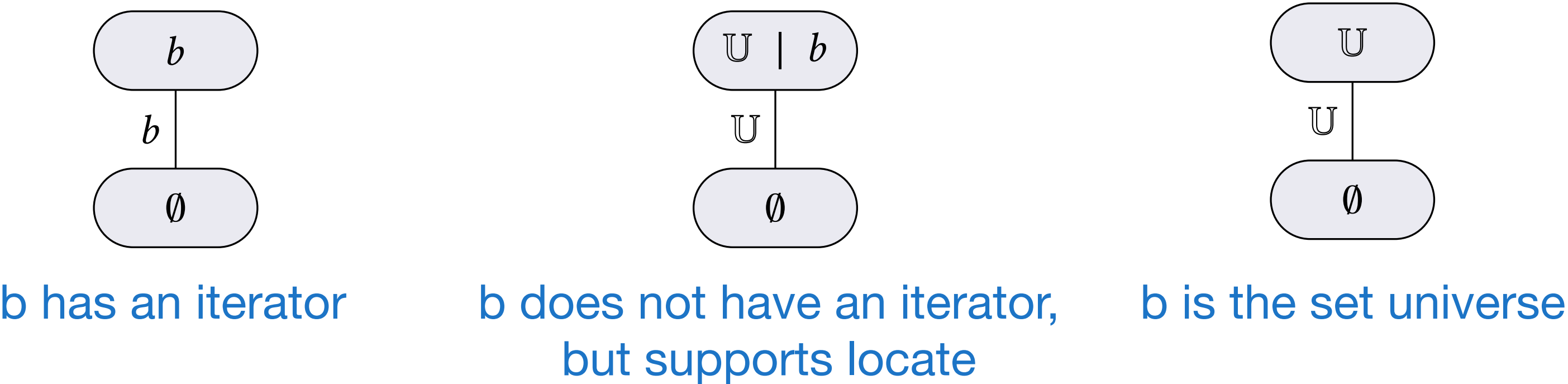


Iteration lattice construction

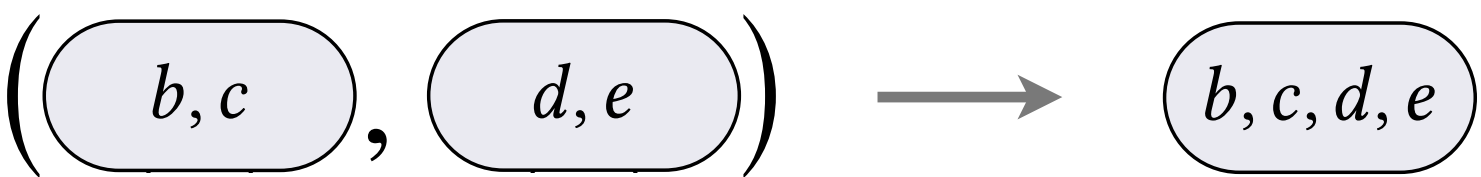


Bottom-up construction from set expression:
create and merge iteration lattices

Base cases:



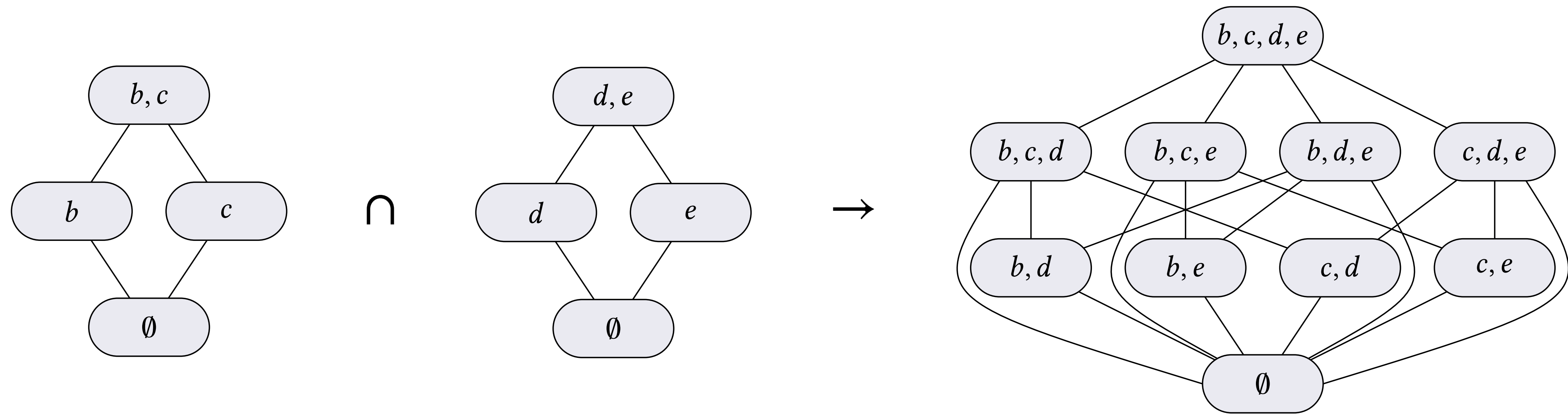
Lattice point merging:



Lattice points are merged by
taking the union of their iterator
and locator sets respectively

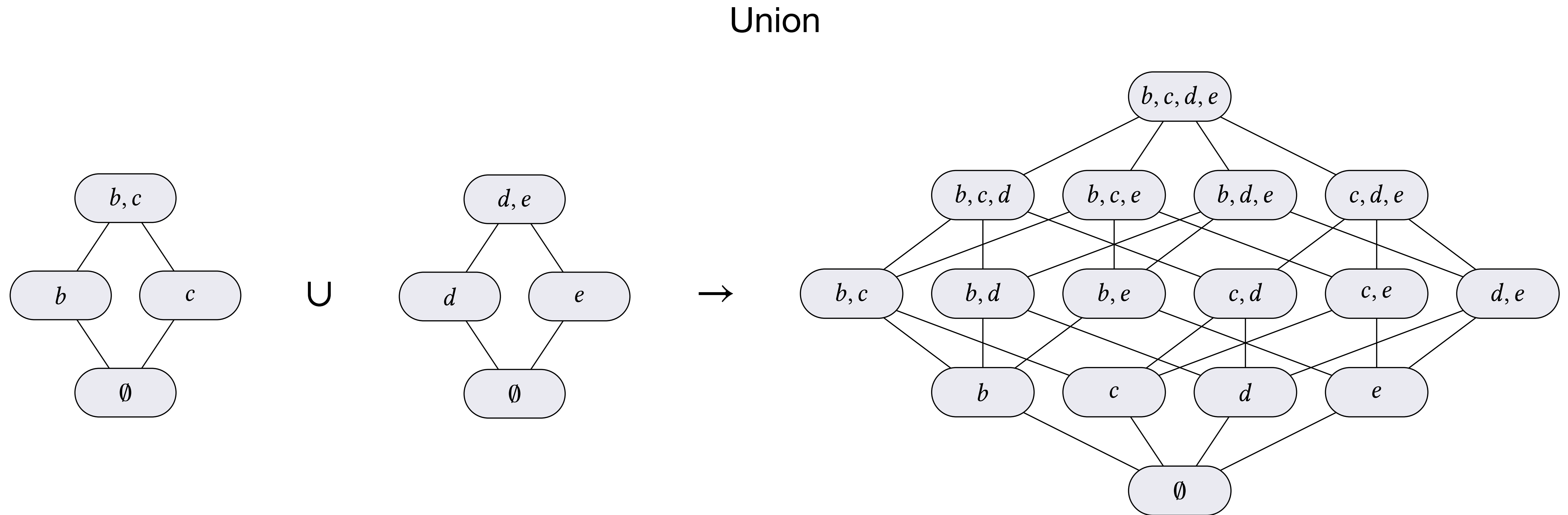
Iteration lattice construction

Intersection



The intersection of two lattices is computed by merging the lattice point pairs in the Cartesian combination of their lattice points.

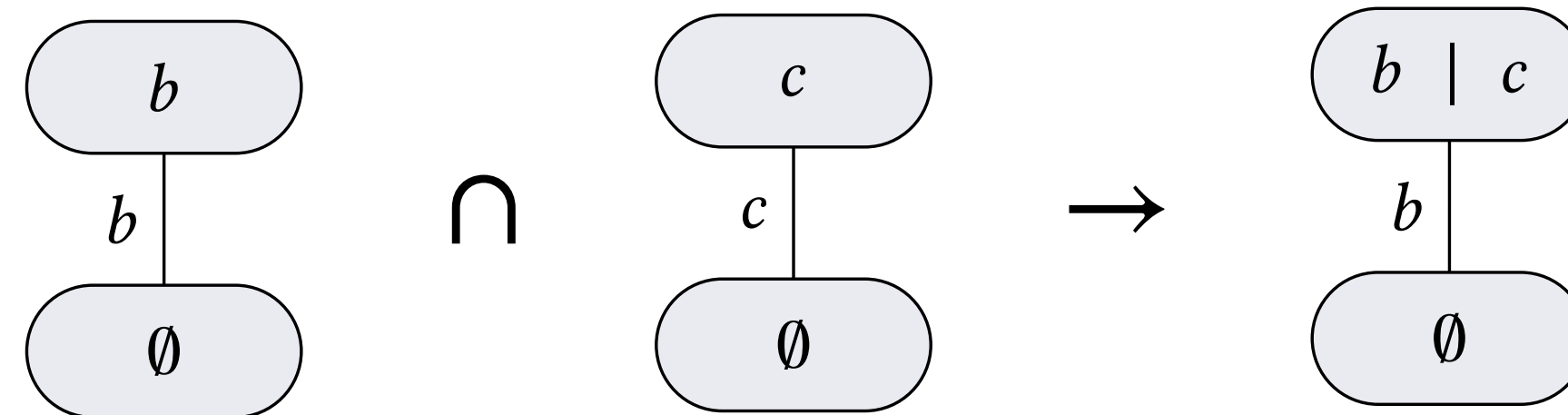
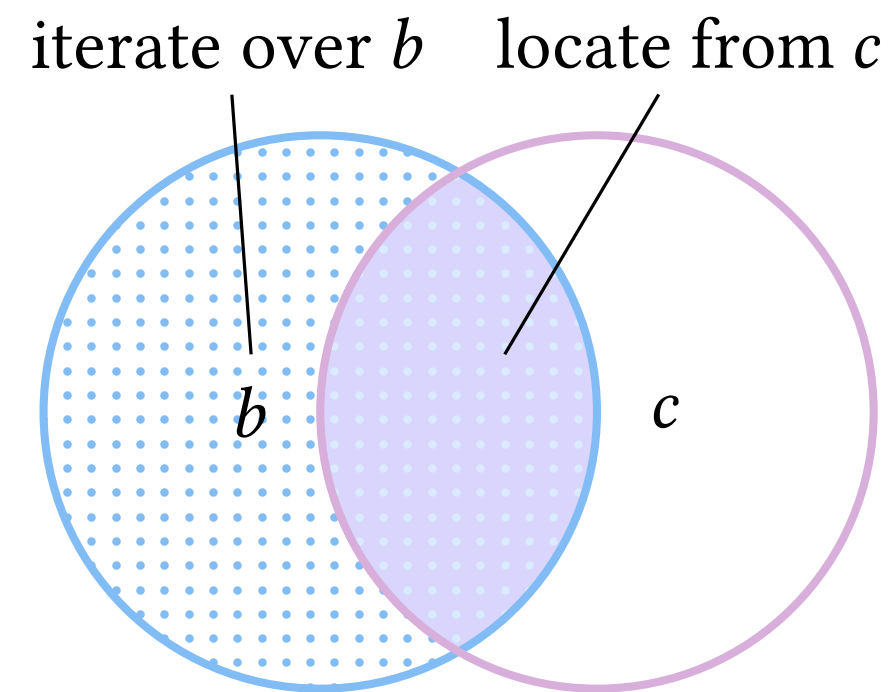
Iteration lattice construction



The union of two lattices is computed by first merging the lattice point pairs in the Cartesian combination of their lattice points. The union of the lattices is then the union of the result and the two initial lattices.

Iteration lattice optimization example

Intersection Optimization



When intersecting two lattices, move the operands with the locate capability from one side of the intersection from the iterators to the locators set.