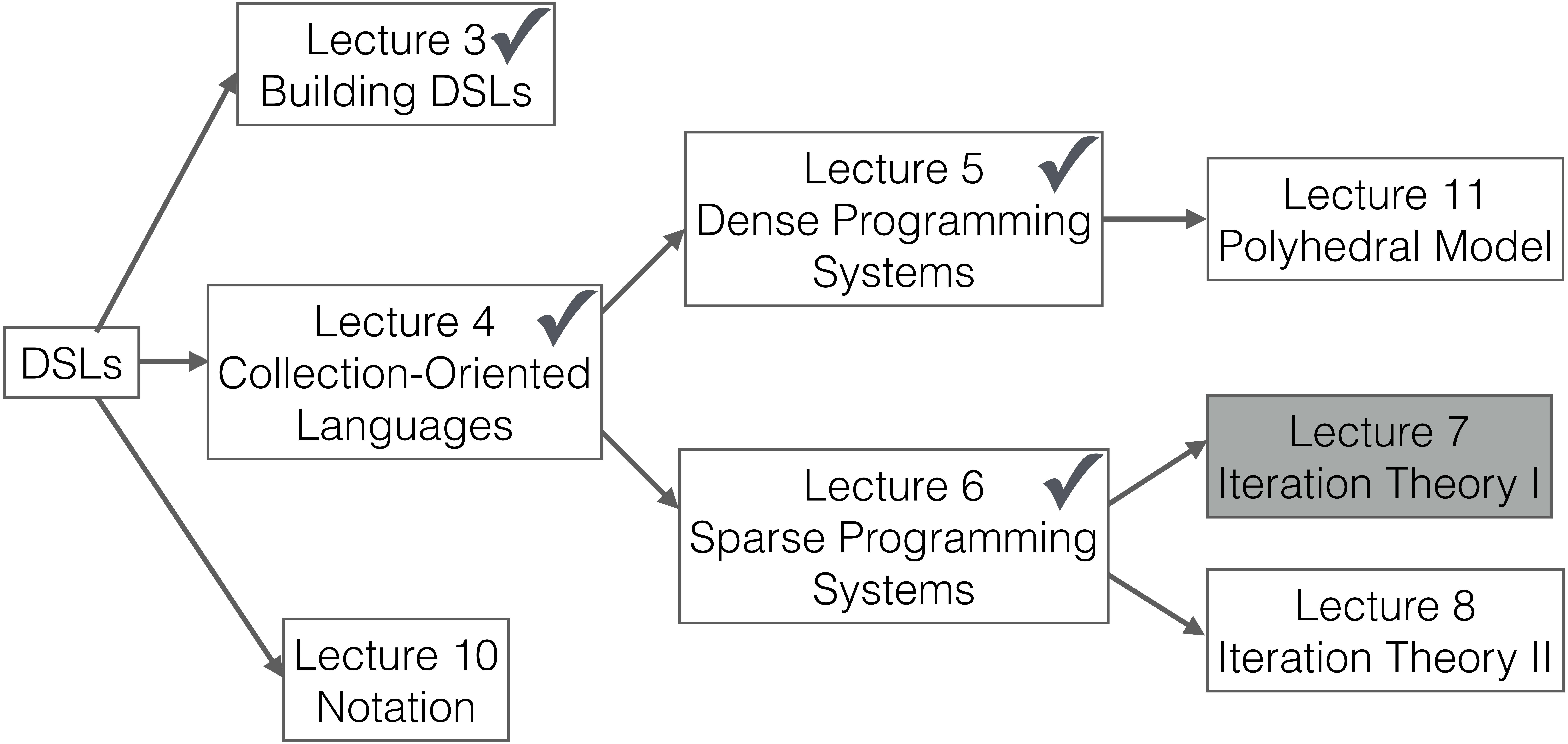
Lecture 7 - Sparse Iteration Theory I

Stanford CS343D (Fall 2020)

Fred Kjolstad and Pat Hanrahan

Course Project

Overview of lectures in the coming weeks



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 Expression

$A = Bc + a$ $a = Bc$

$A = B \odot C$ $A = B + C$ $a = \alpha Bc + \beta a$

$A = BCd$ $A = \alpha B$ $A = 0$ $A = BC$

$a = b \odot c$ $A = B \odot (CD)$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $A = B^T$ $a = B^T Bc$

$A_{ik} = \sum_{j} B_{ijk} c_j$ $A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $A_{ij} = (\sum_{k} B_{ijk} C_{ijk}) + D_{ij}$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$ $\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij})(\sum_{k} z_k \theta_{ik})$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$

Formats

Dense Matrix CSR BCSR DCSR ELLPACK COO CSB DIA Blocked COO CSC Blocked DIA DCSC Sparse vector Hash Maps Dense Tensors CSF Blocked Tensors

Schedule

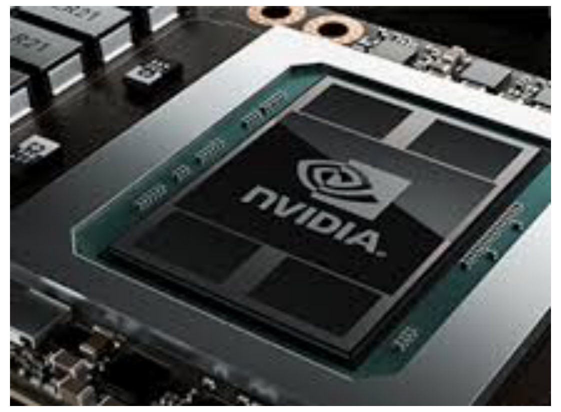
reorder split collapse precmpute unroll parallelize

Sparse Tensor Algebra Compiler (taco)

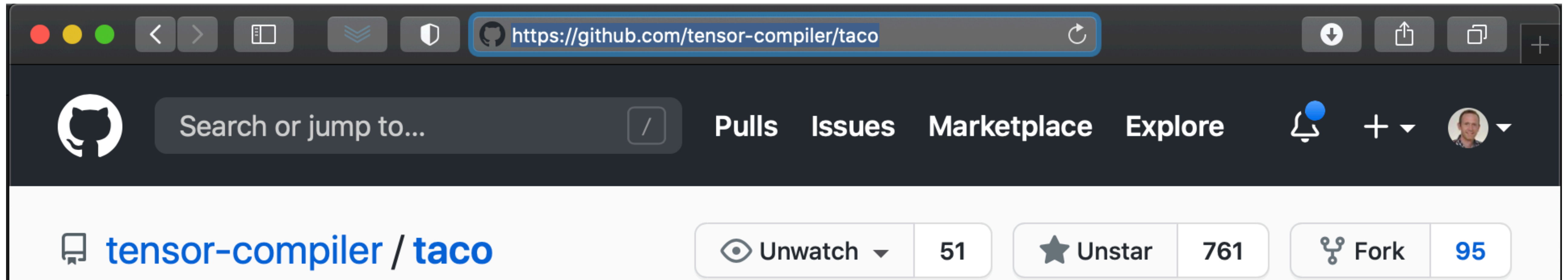
THE C PROGRAMMING LANGUAGE

NVIDIA CUDA

DSAs

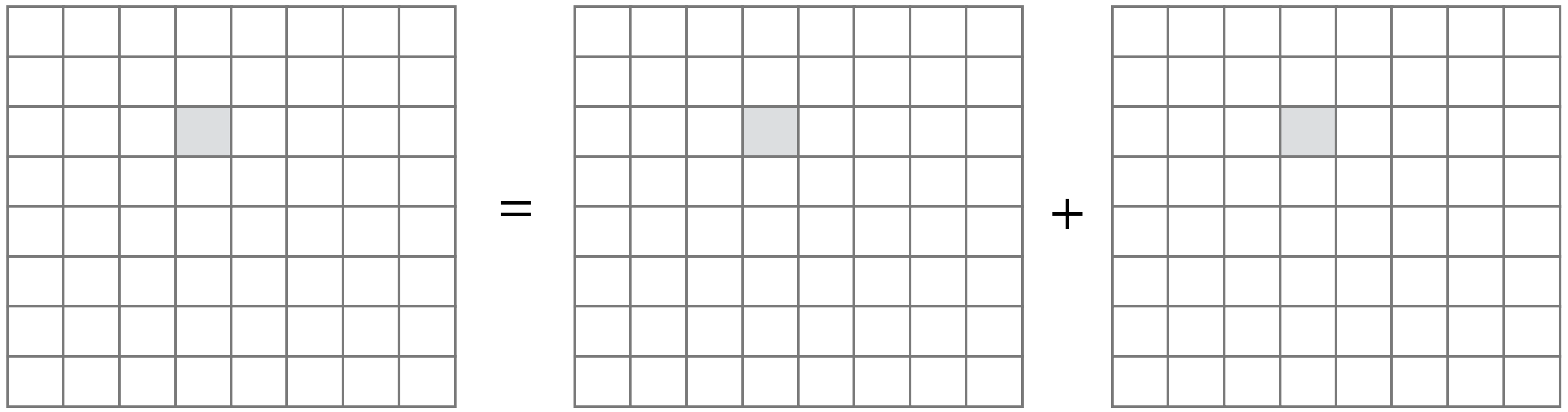


(work in progress)



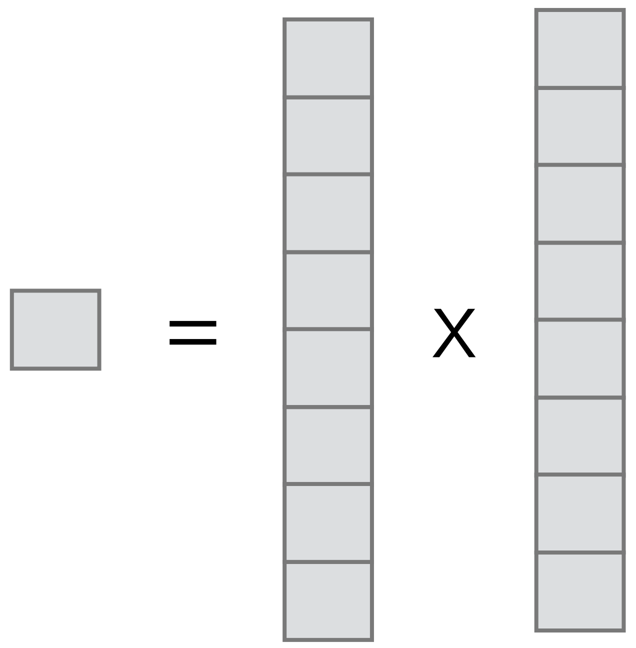
Tensor index notation for expressing functionality

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



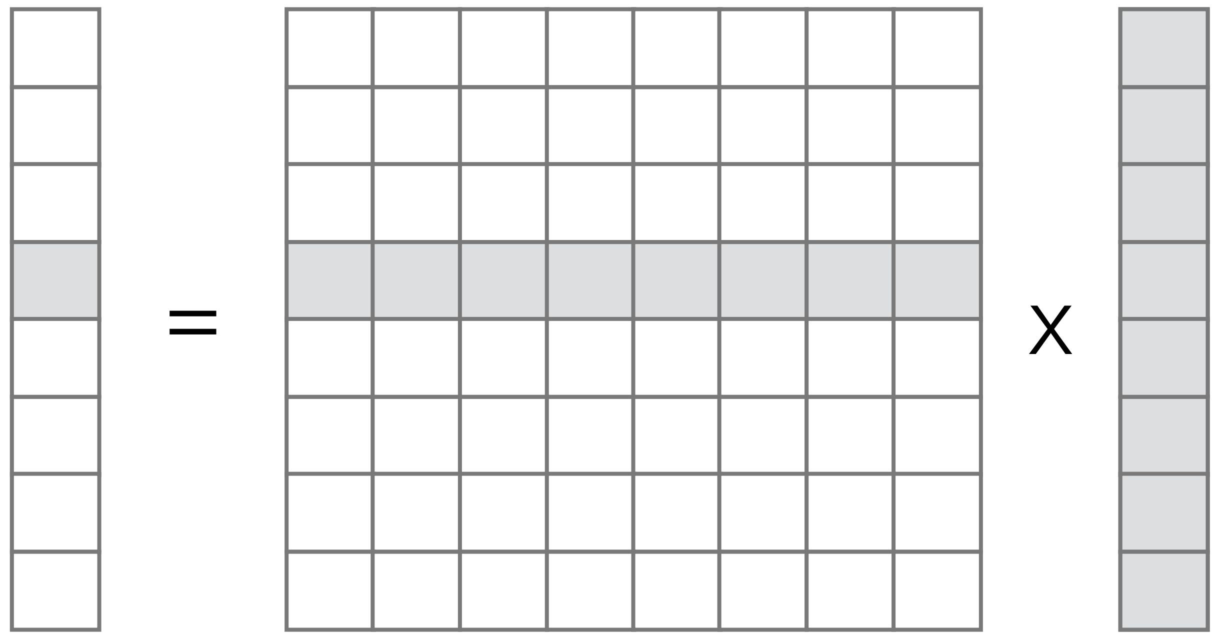
element-wise

$$\alpha = \sum_i b_i c_i$$



reduction over i

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



broadcast c_j over i

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

$a = Bc$

$a = Bc + a$

$a = Bc + b$ $A = B + C$ $a = \alpha Bc + \beta a$

$a = B^T c$ $A = \alpha B$ $a = B(c + d)$

$a = B^T c + d$ $A = B + C + D$ $A = BC$

$A = B \odot C$ $a = b \odot c$ $A = 0$ $A = B \odot (CD)$

$A = BCd$ $A = B^T$ $a = B^T Bc$

$a = b + c$ $A = B$ $K = A^T CA$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj}$ $A_{ij} = \sum_{k} B_{ijk} c_k$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $A_{ik} = \sum_{j} B_{ijk} c_j$

$A_{jk} = \sum_{i} B_{ijk} c_i$ $A_{ijl} = \sum_{k} B_{ikl} C_{kj}$

$\tau = \sum_{i} z_i (\sum_{j} z_j \theta_{ij}) (\sum_{k} z_k \theta_{ik})$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$

$\times$

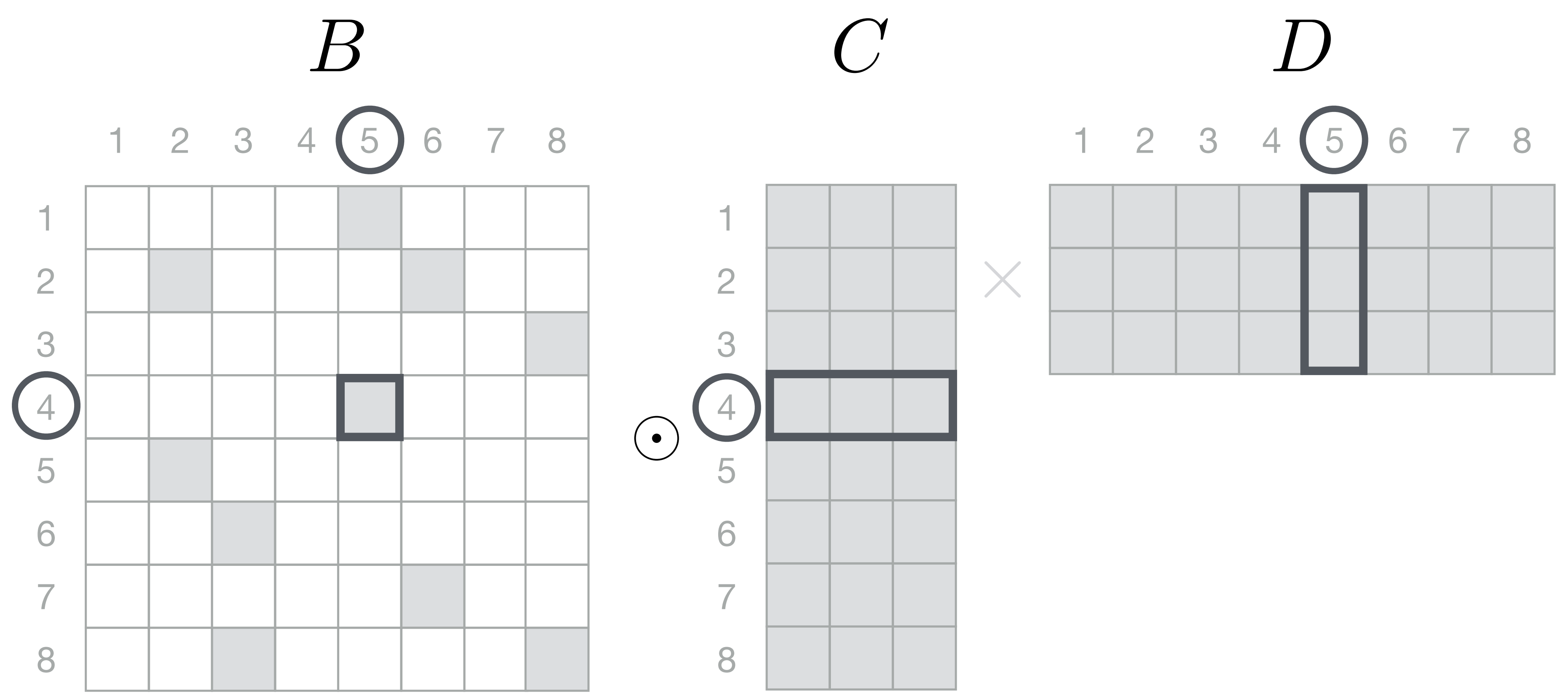
Dense Matrix
CSR DCSR BCSR
COO ELLPACK CSB
Blocked COO CSC
DIA Blocked DIA DCSC
Sparse vector Hash Maps
Coordinates
Dense Tensors
CSF
Blocked Tensors
Linked Lists Database
Compression Schemes
Cloud Storage

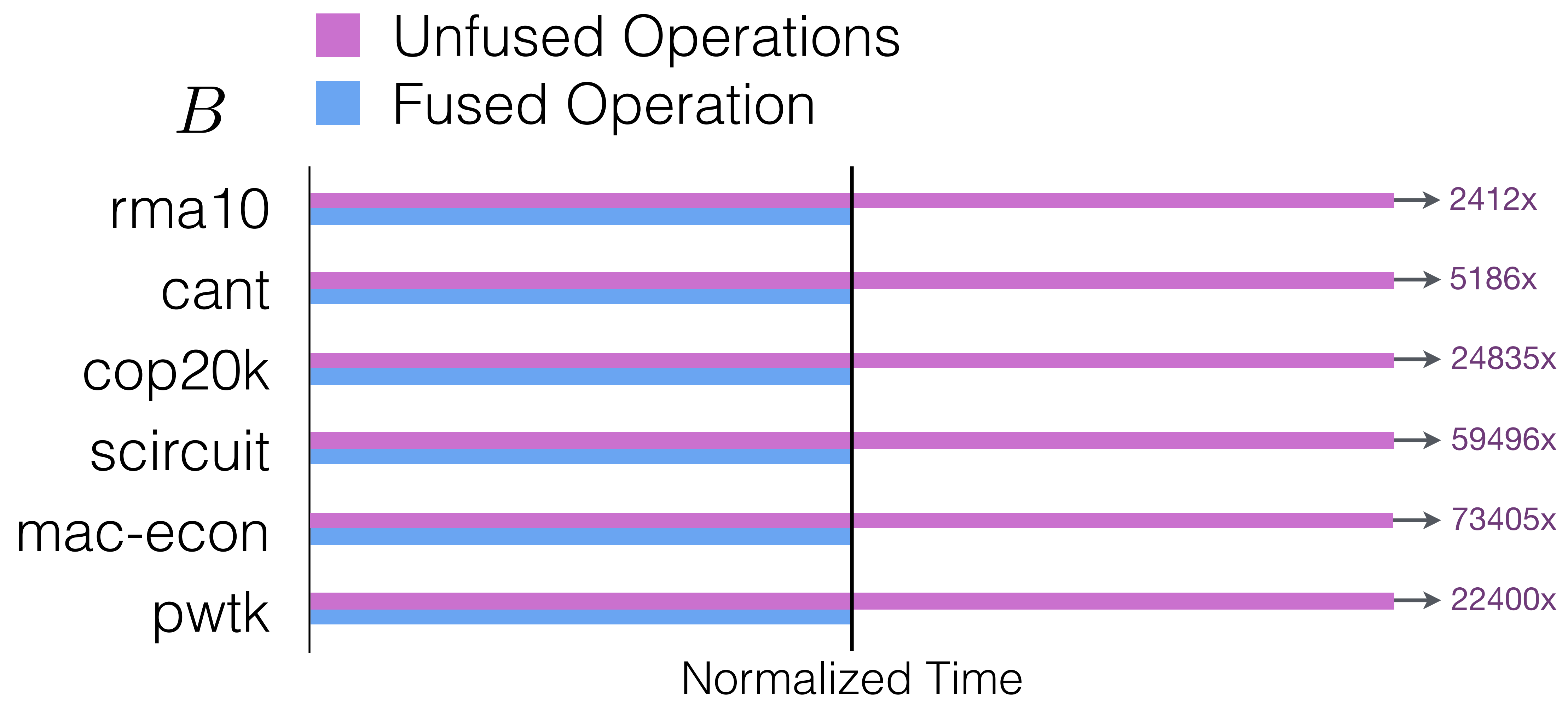
$\times$

CPU
GPUs TPUs
FPGA
Sparse Tensor Hardware
Cloud Computers
Supercomputers

Expressions matter for performance

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



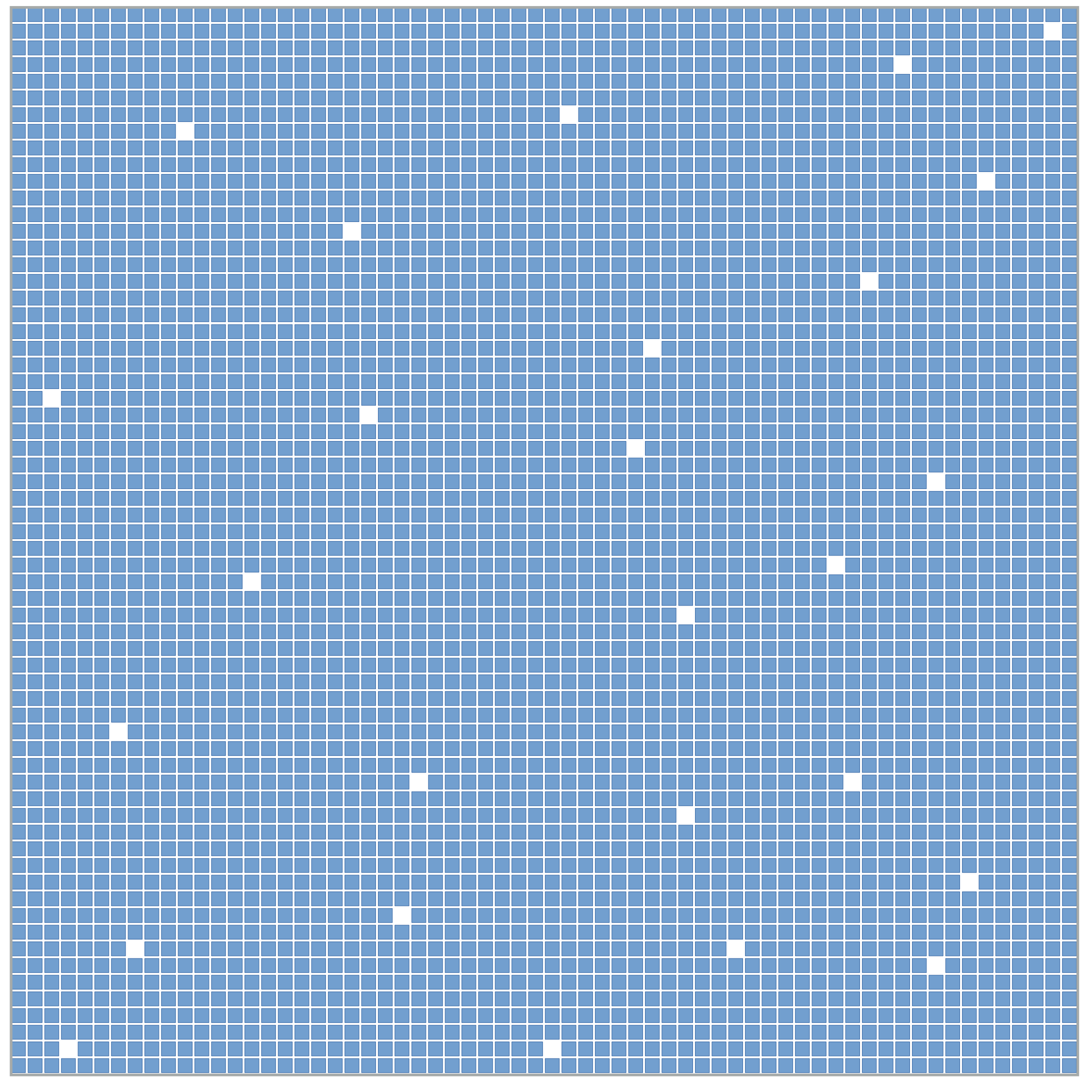


Unfused: $\Theta(n^2 k)$

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

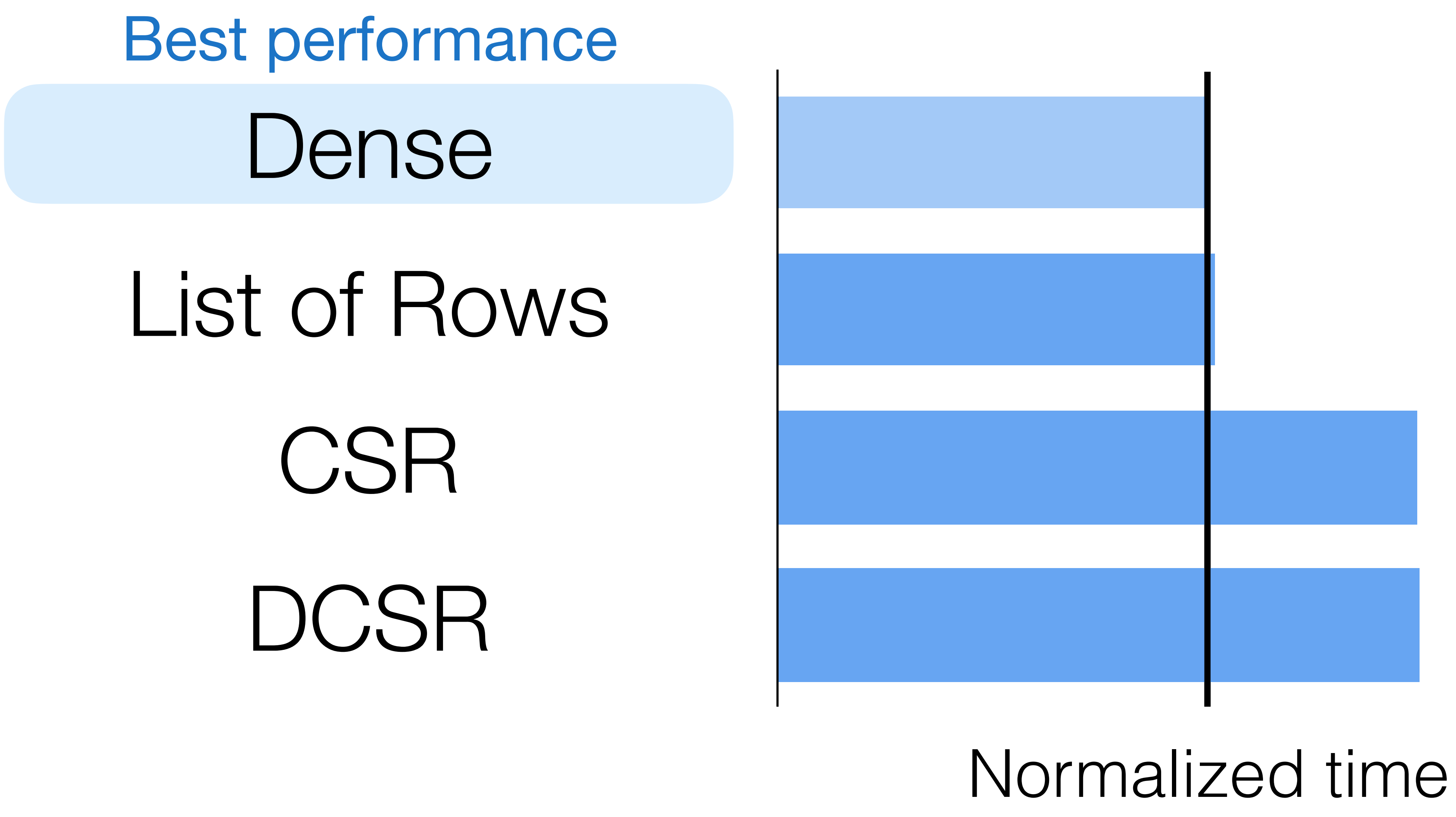
Formats matter for performance

Dense Matrix



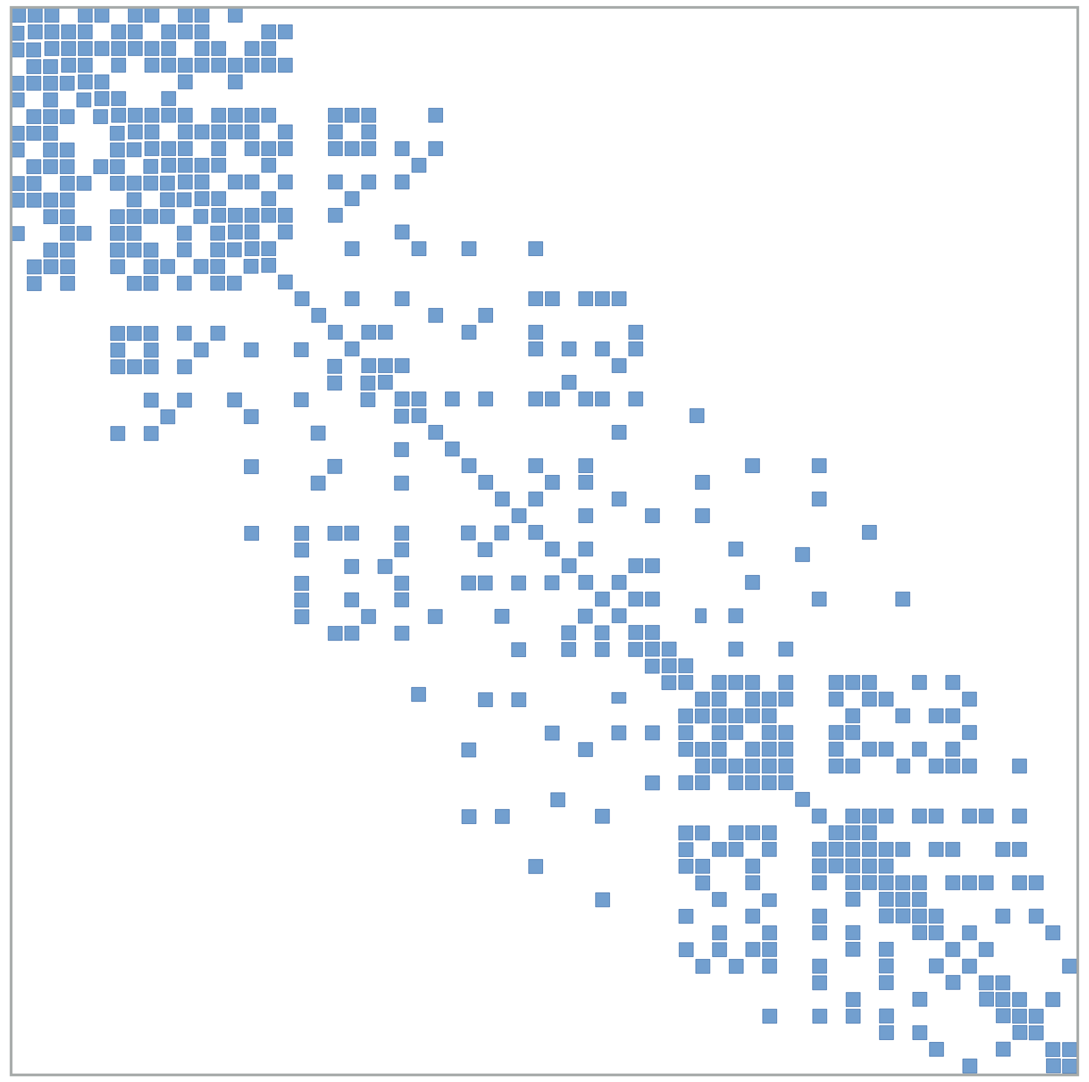
Formats

$y = Ax$



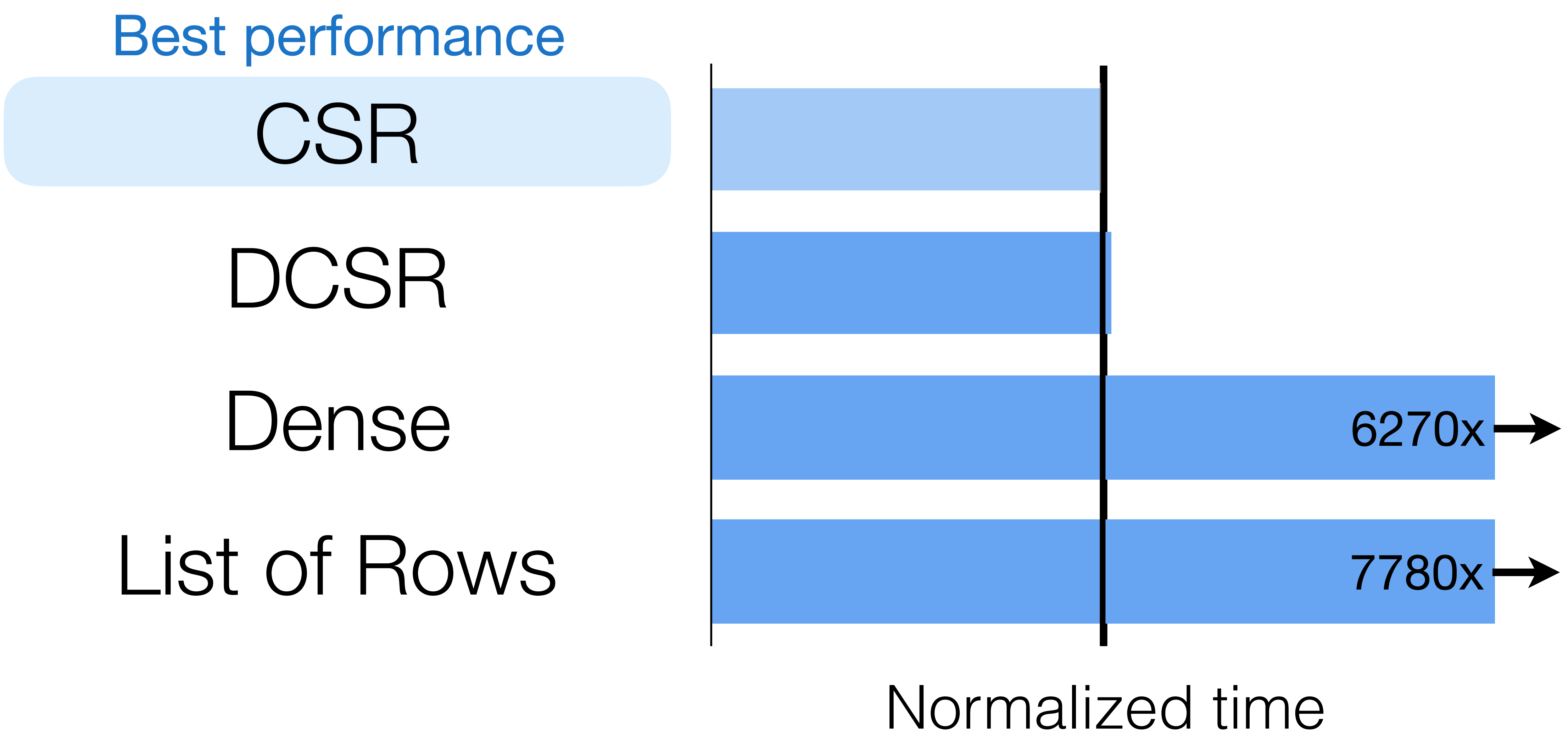
Formats matter for performance

Thermal Matrix



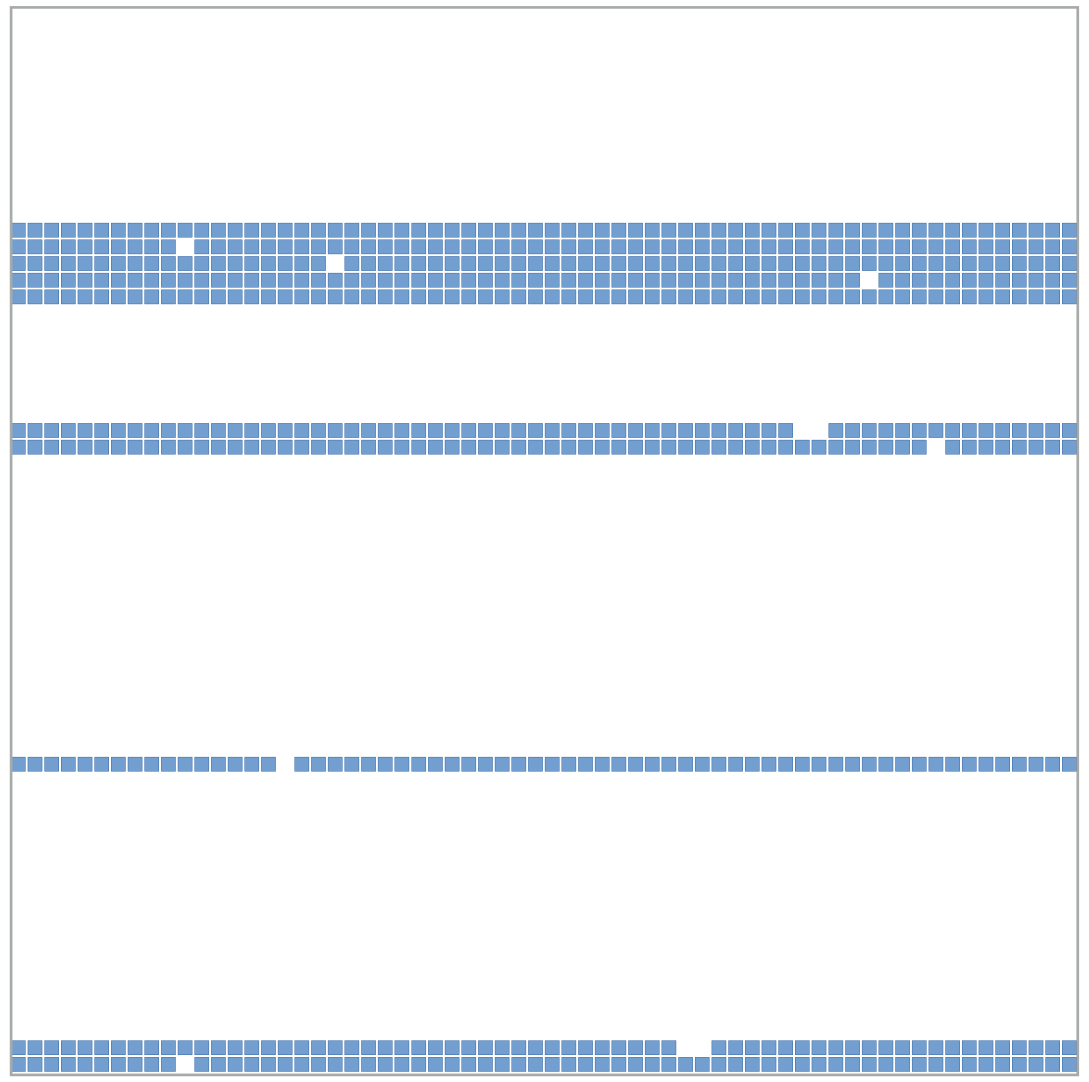
Formats

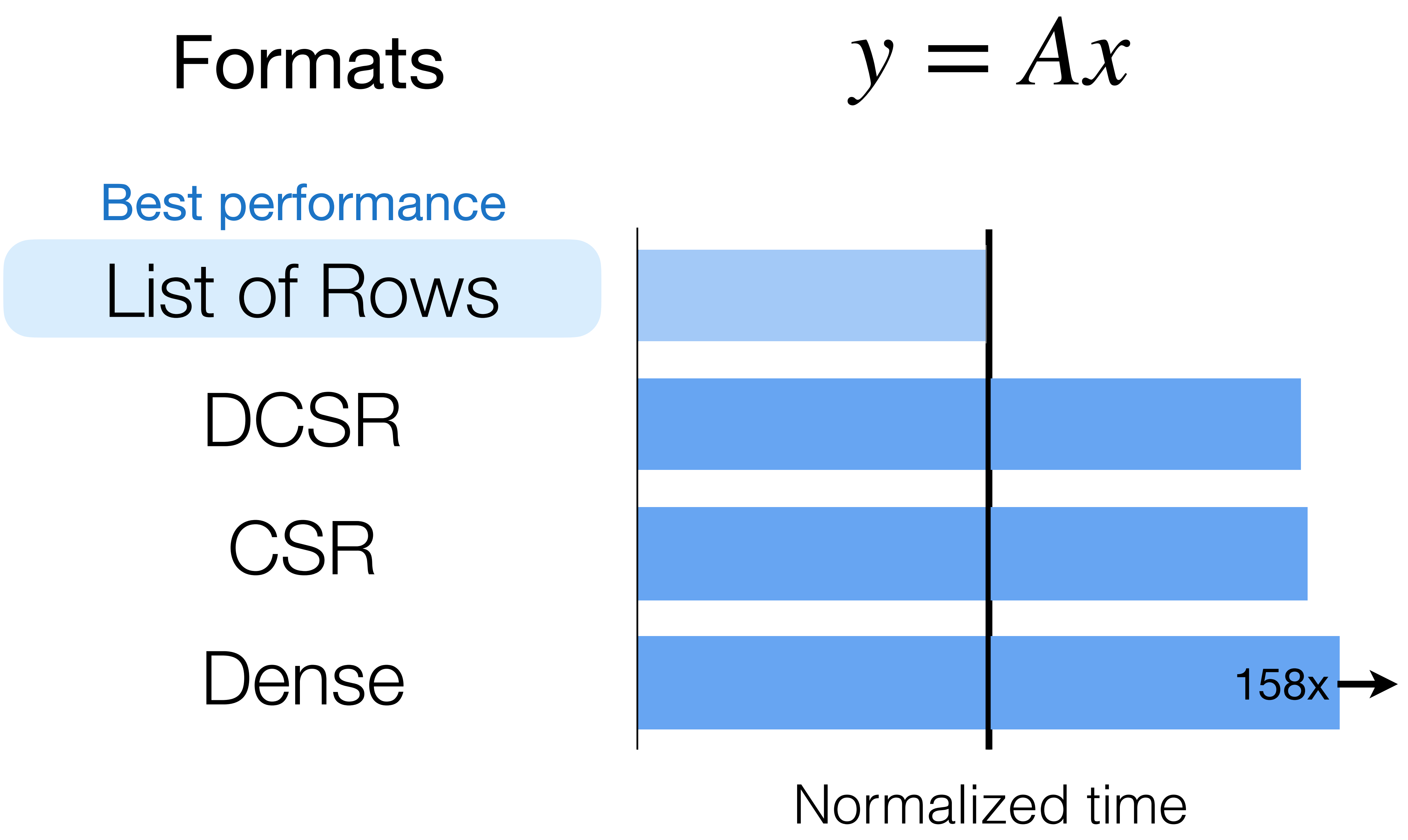
$y = Ax$



Formats matter for performance

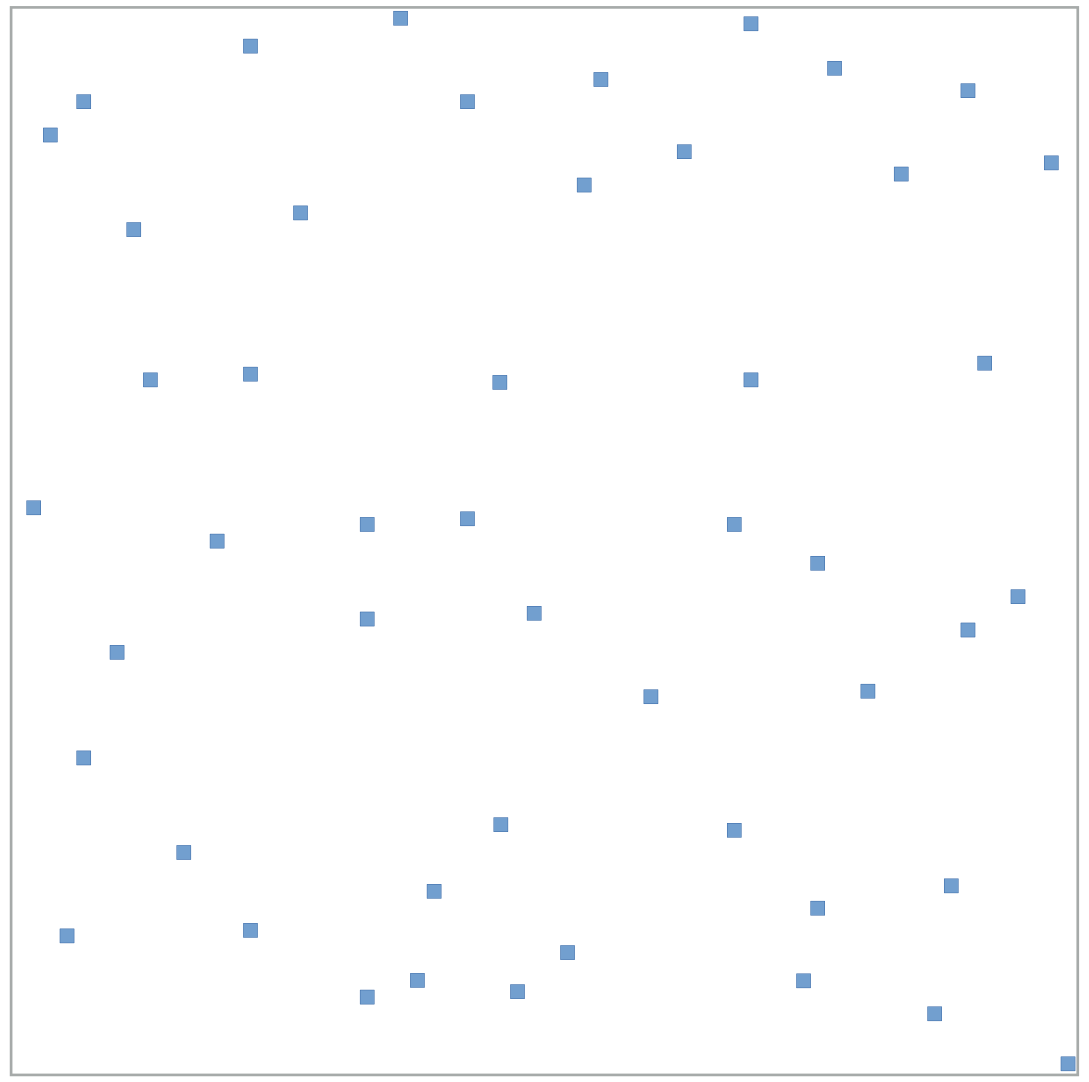
Row-sliced Matrix

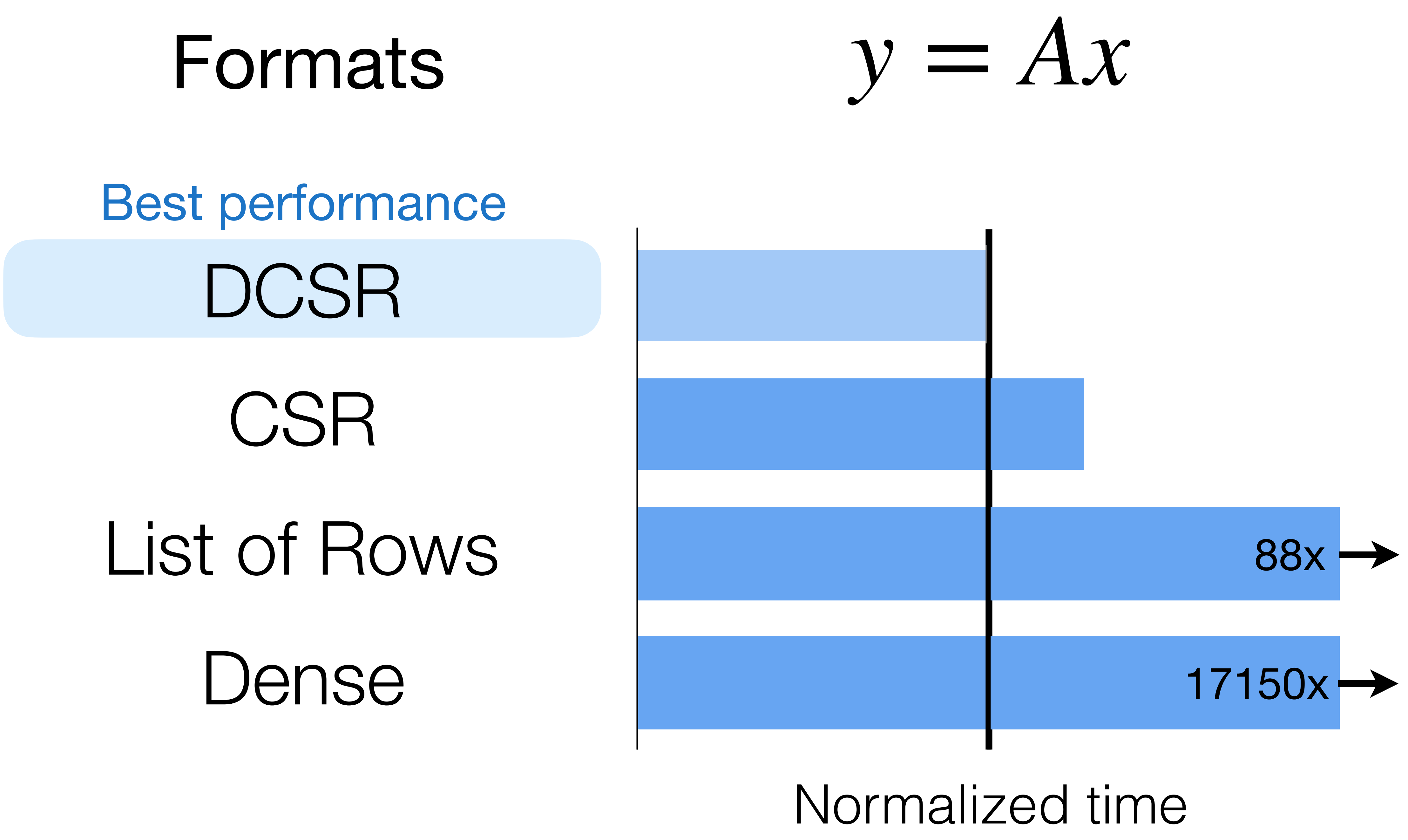




Formats matter for performance

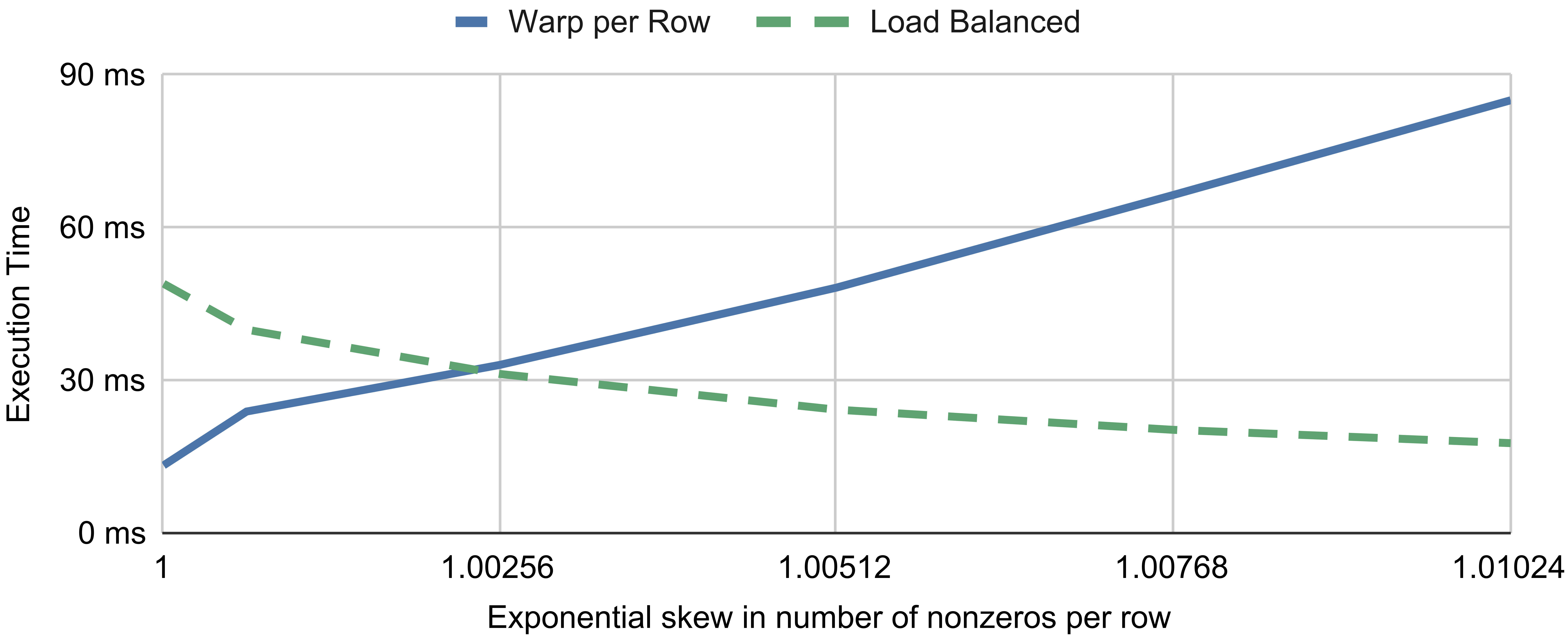
Hypersparse Matrix





Schedules matter for performance

$$y = Ax$$



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)

(v_1,v_5)
(v_4,v_3)
(v_5,v_3)
(v_3,v_1)
(v_3,v_5)

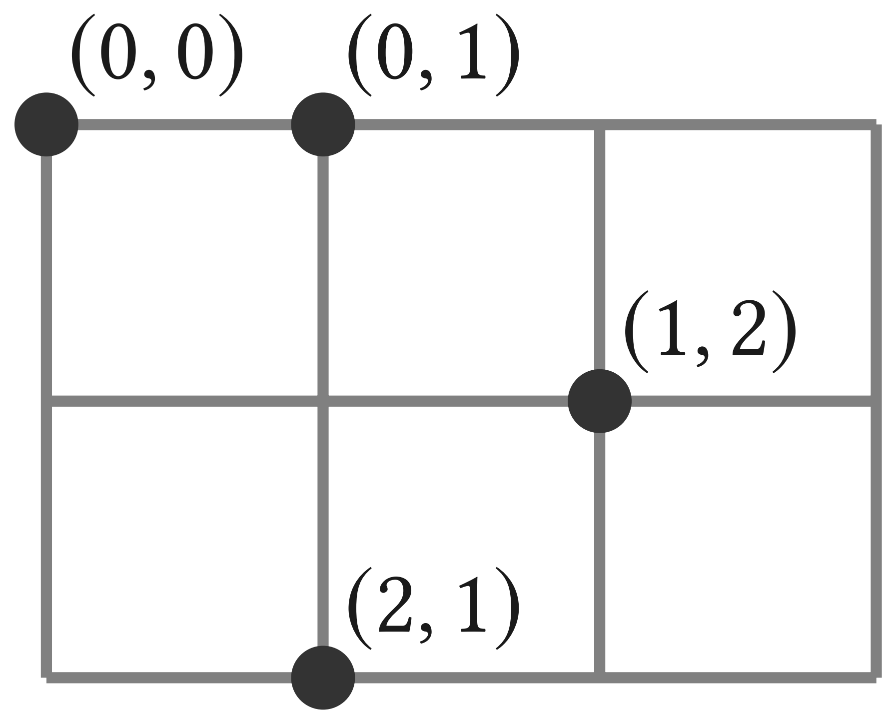
Values may be attached to these coordinates: e.g., nonzero values, edge attributes

Iteration spaces from coordinate relations

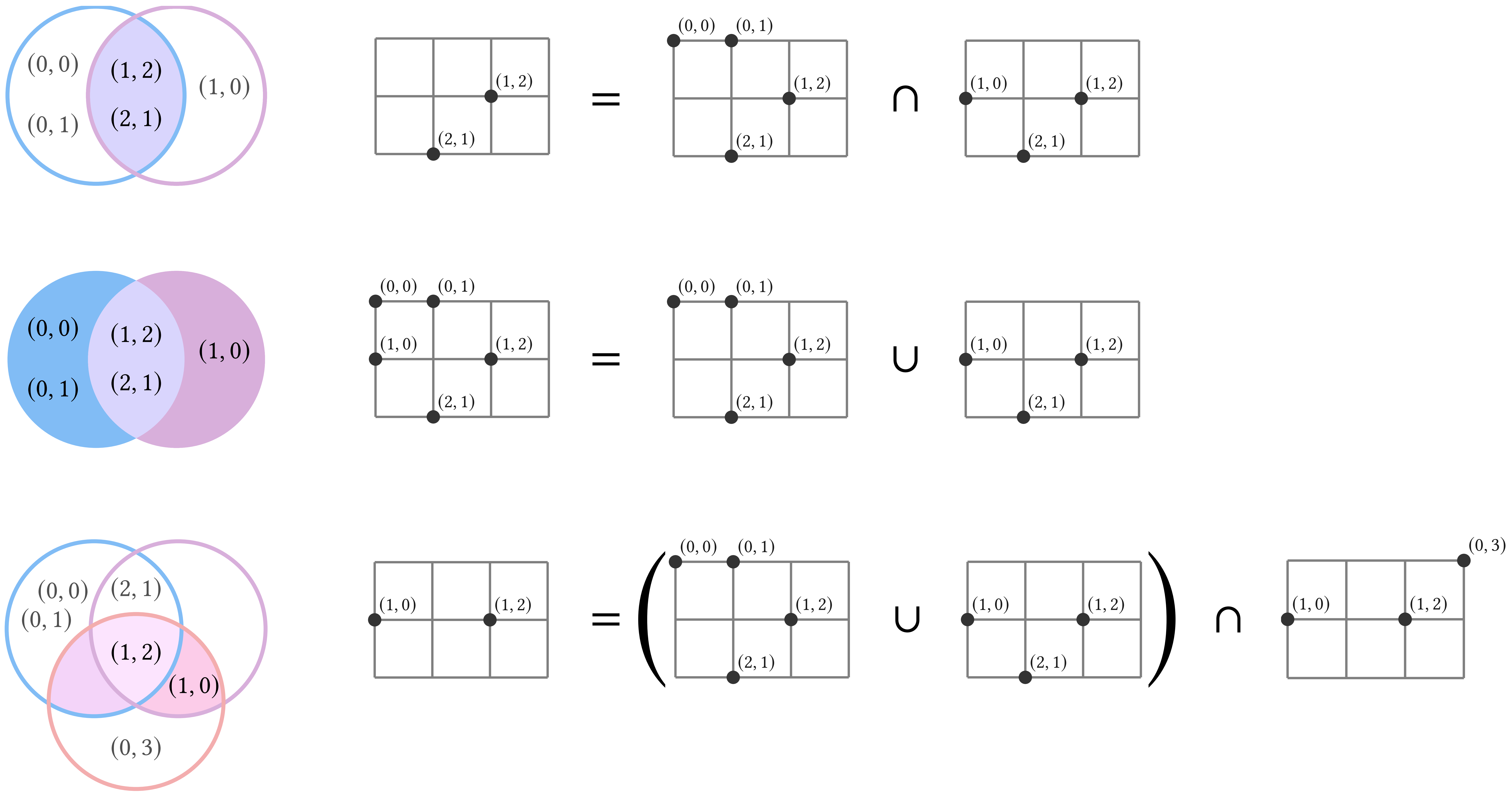
$$(0, 0)$$

$$(0, 1)$$

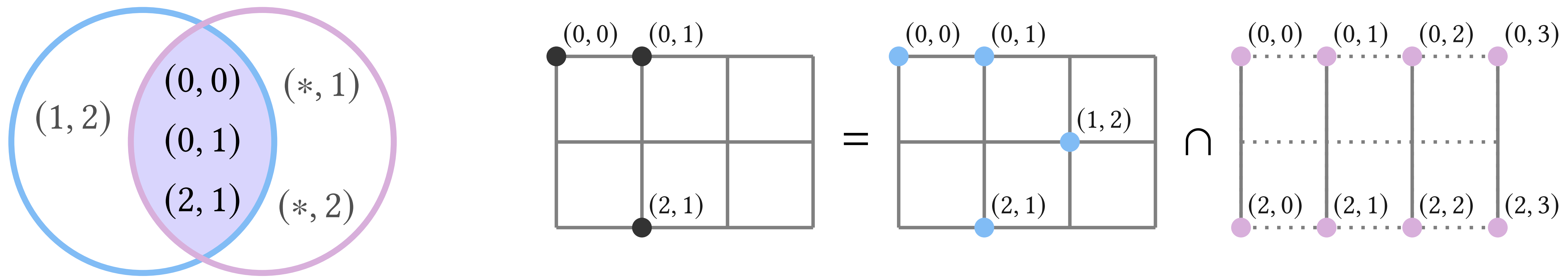
$$(1, 2) \quad (2, 1)$$

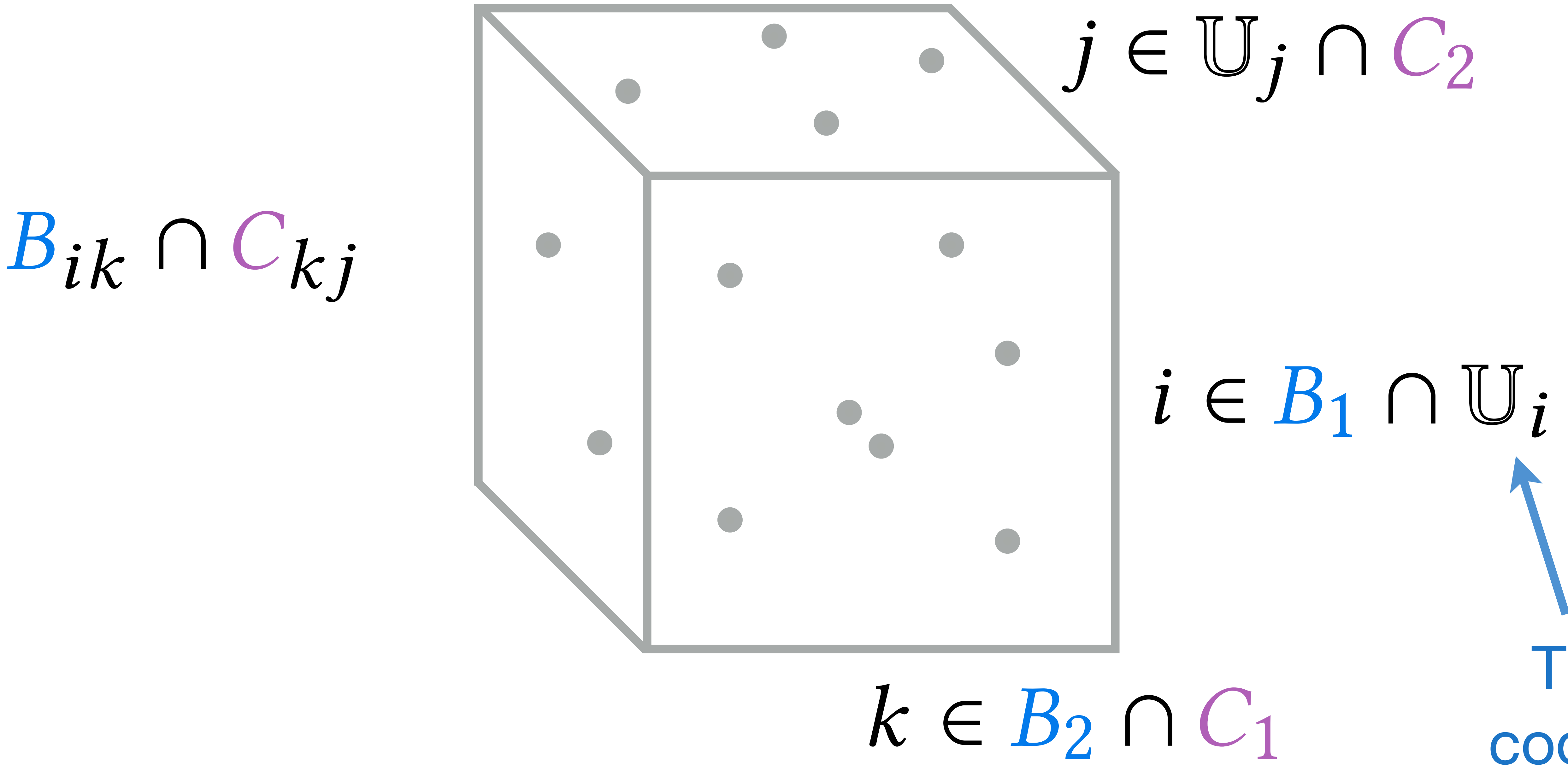


Iteration spaces from set operations



Iteration spaces from broadcast operations





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

Coordinate relations → coordinate trees

Matrix

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

Coordinate Relation

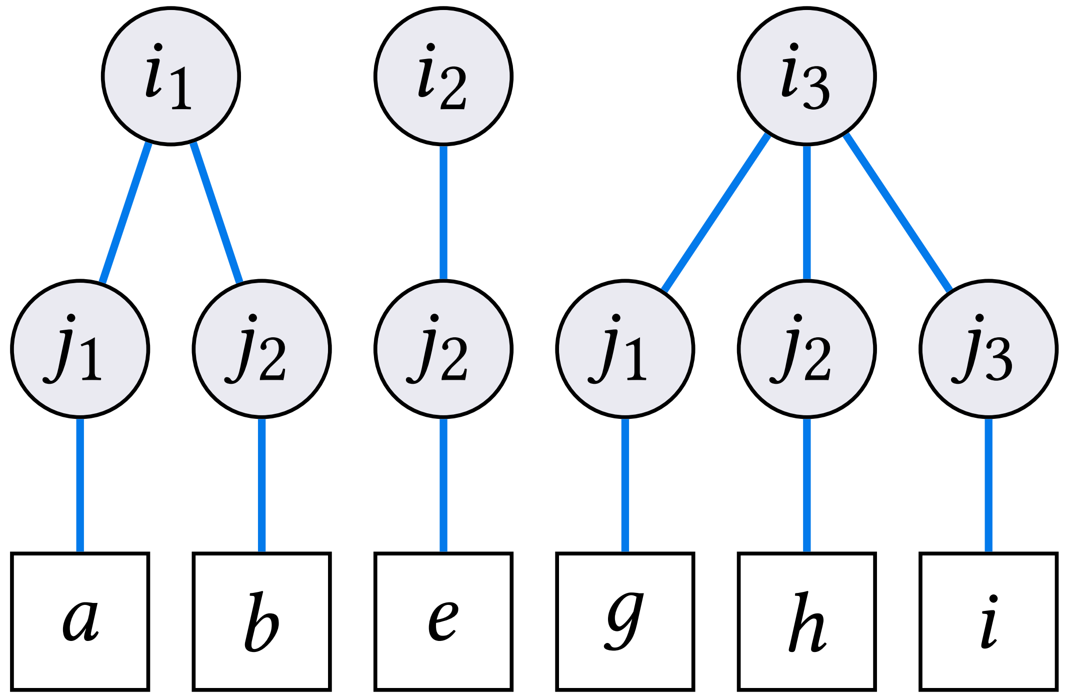
$(i_1, j_1) \to a$ $\quad (i_1, j_2) \to b$

$(i_3, j_3) \to i$ $\quad (i_2, j_2) \to e$

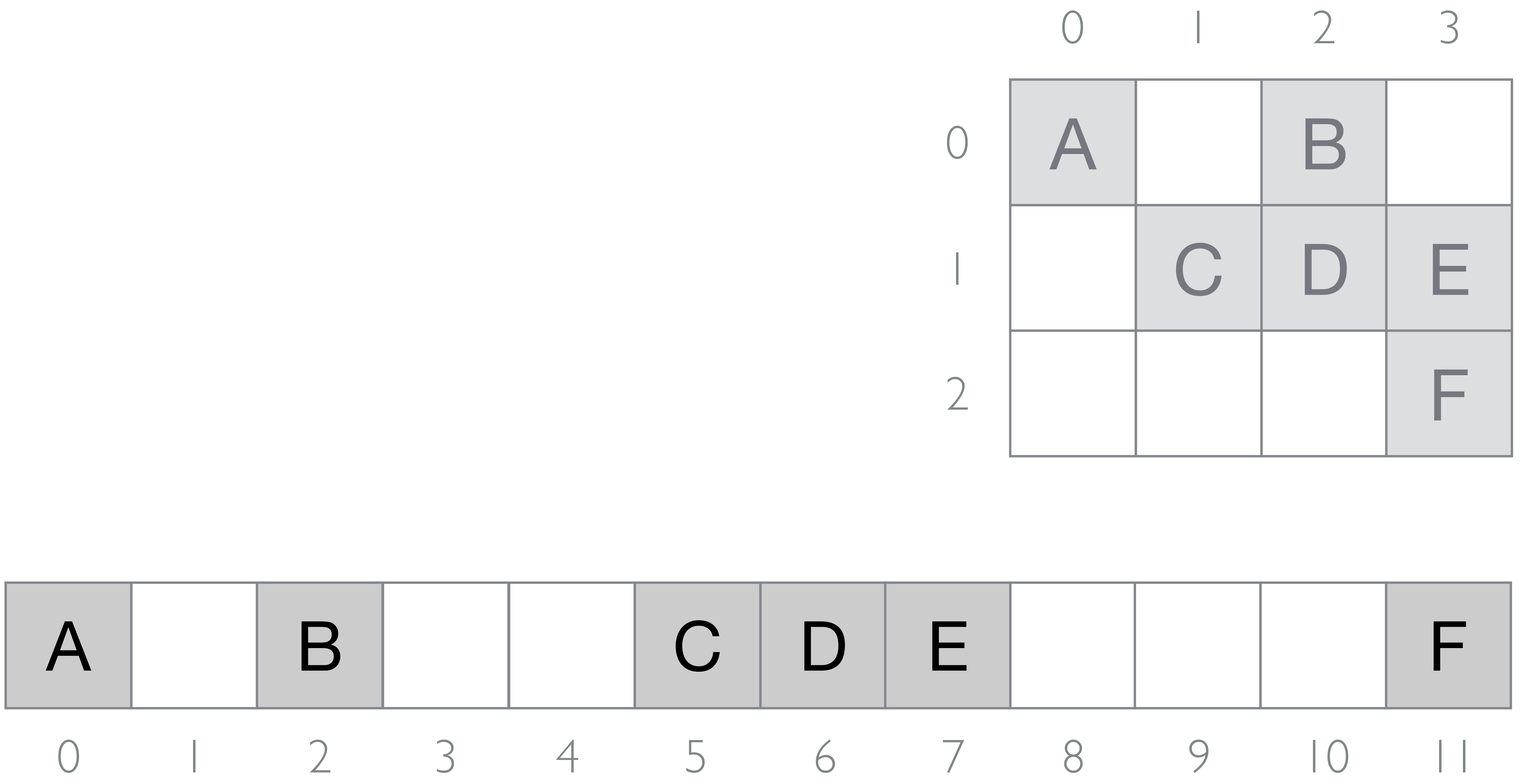
$(i_3, j_1) \to g$

$(i_3, j_2) \to h$

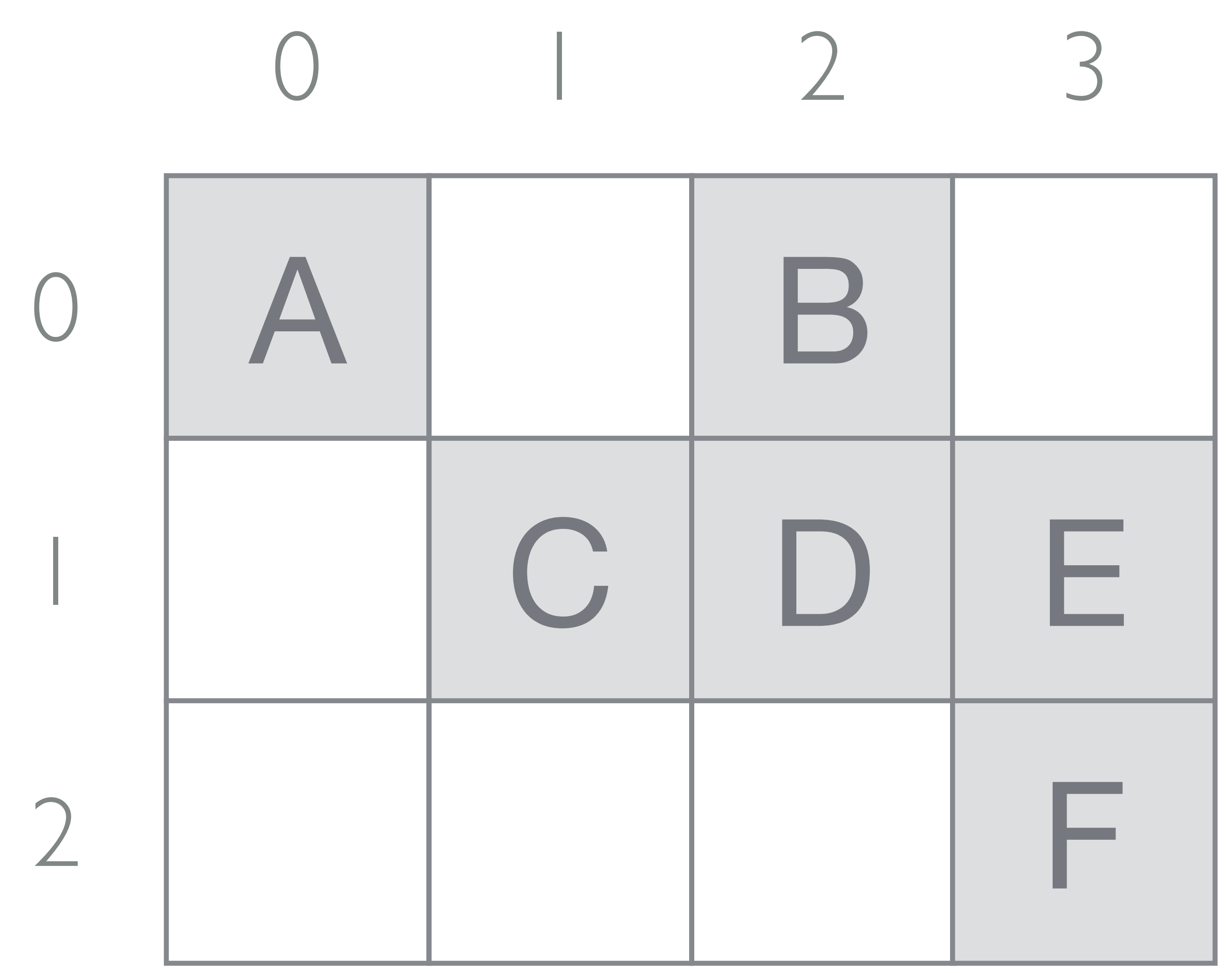
Coordinate Tree

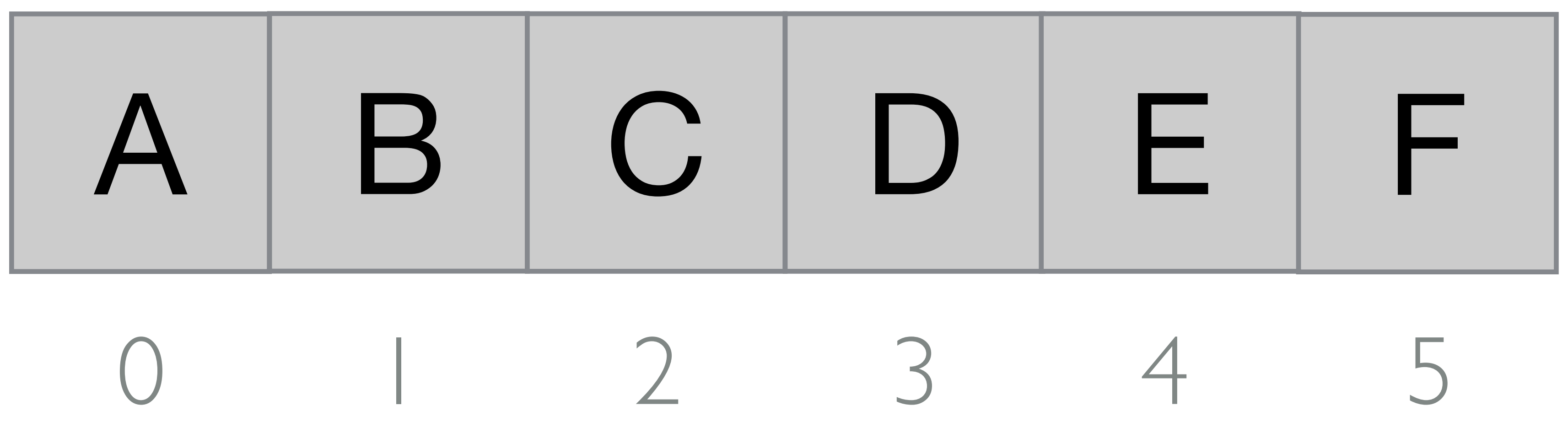


Coordinate relations → coordinate trees



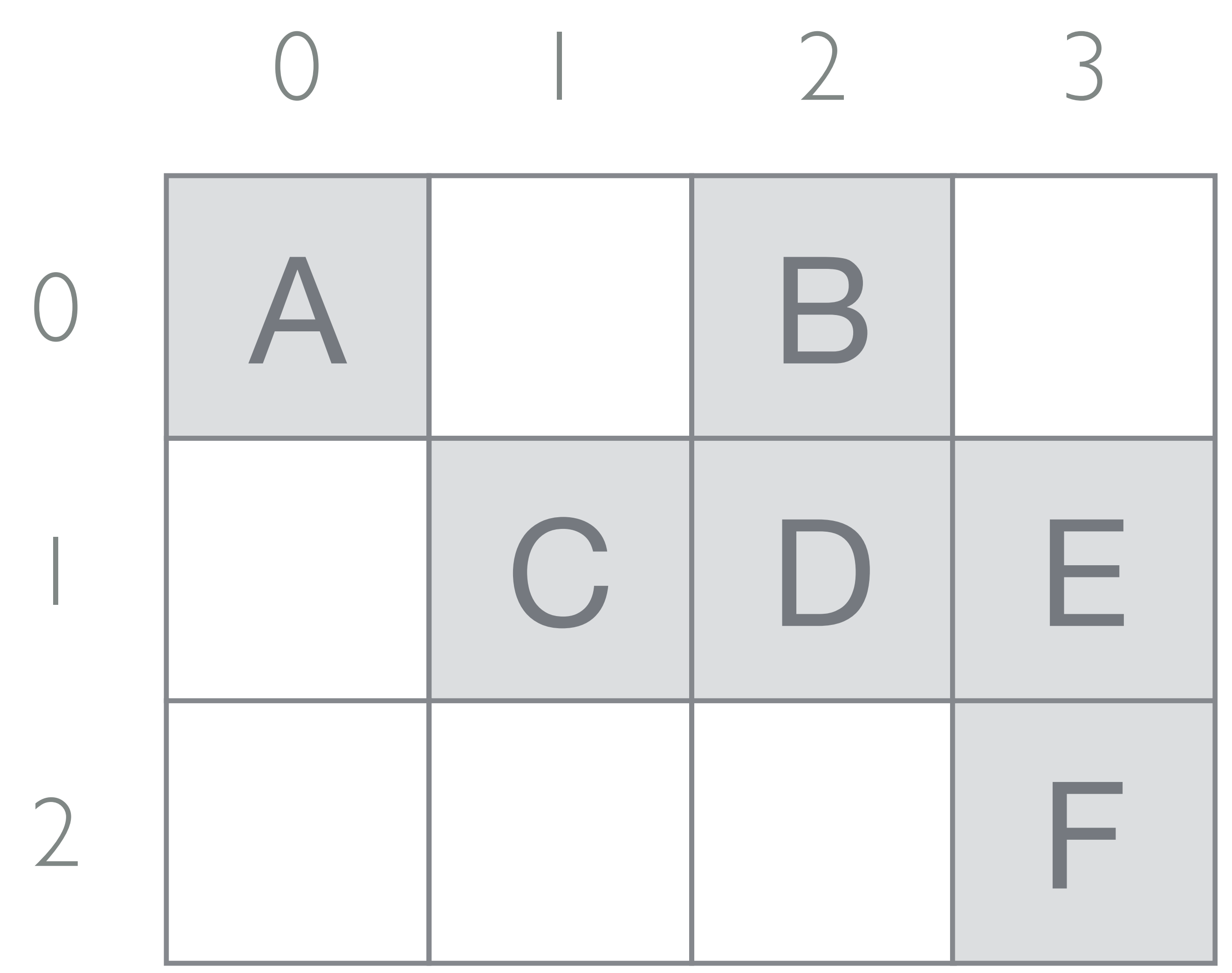
Coordinate relations → coordinate trees



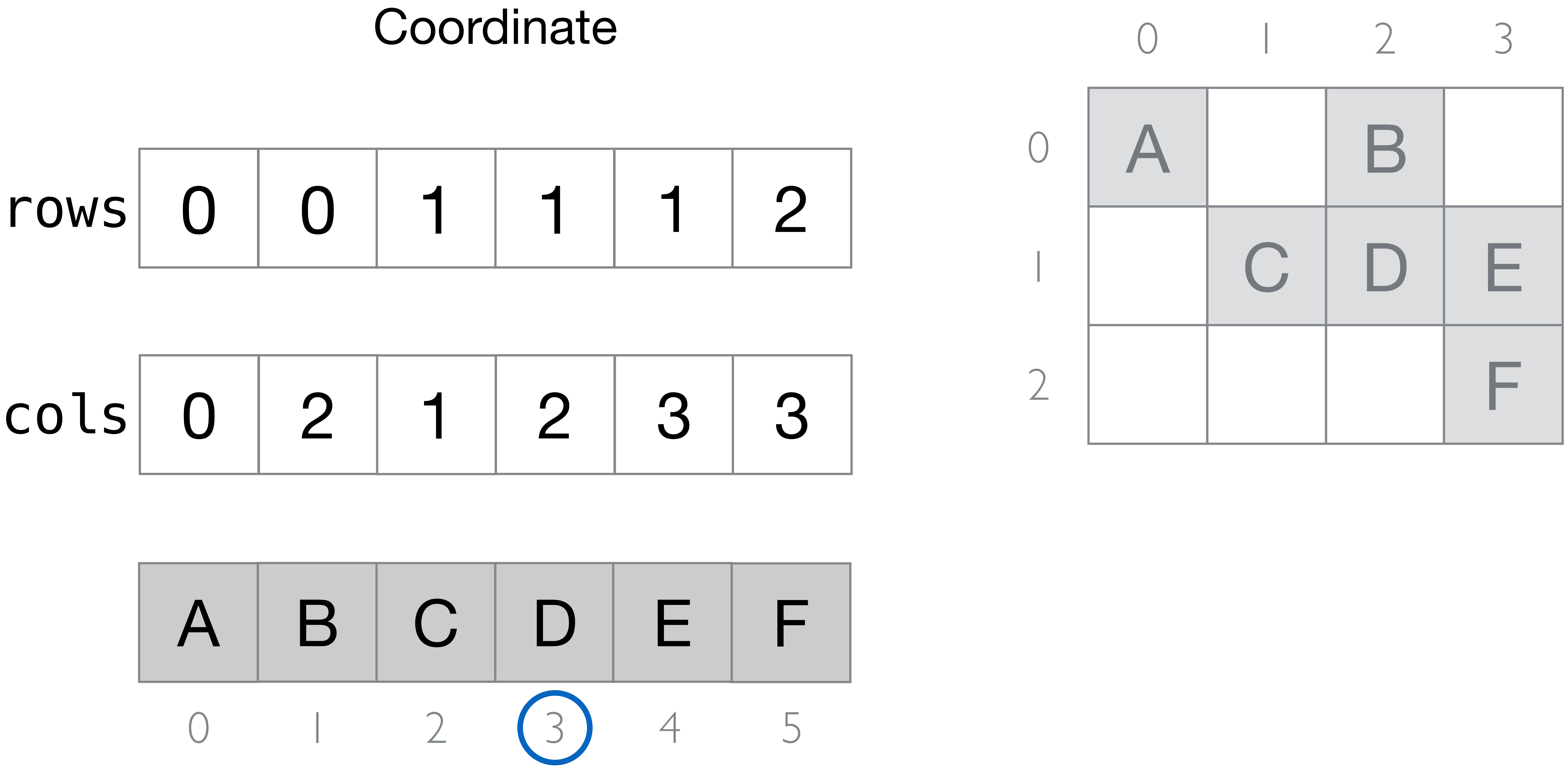


Coordinate relations → coordinate trees

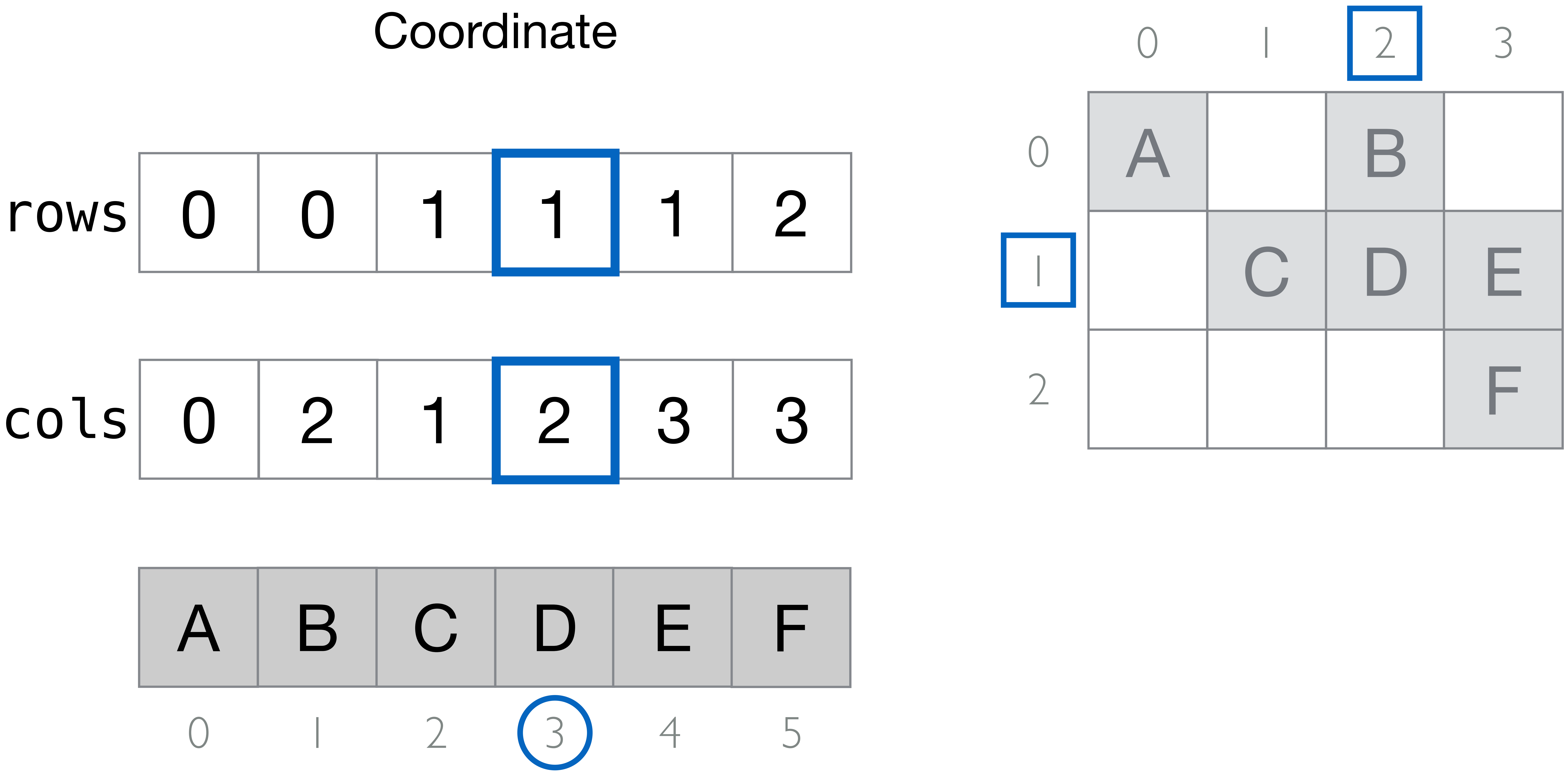
```
row(3) = ???
col(3) = ???
```



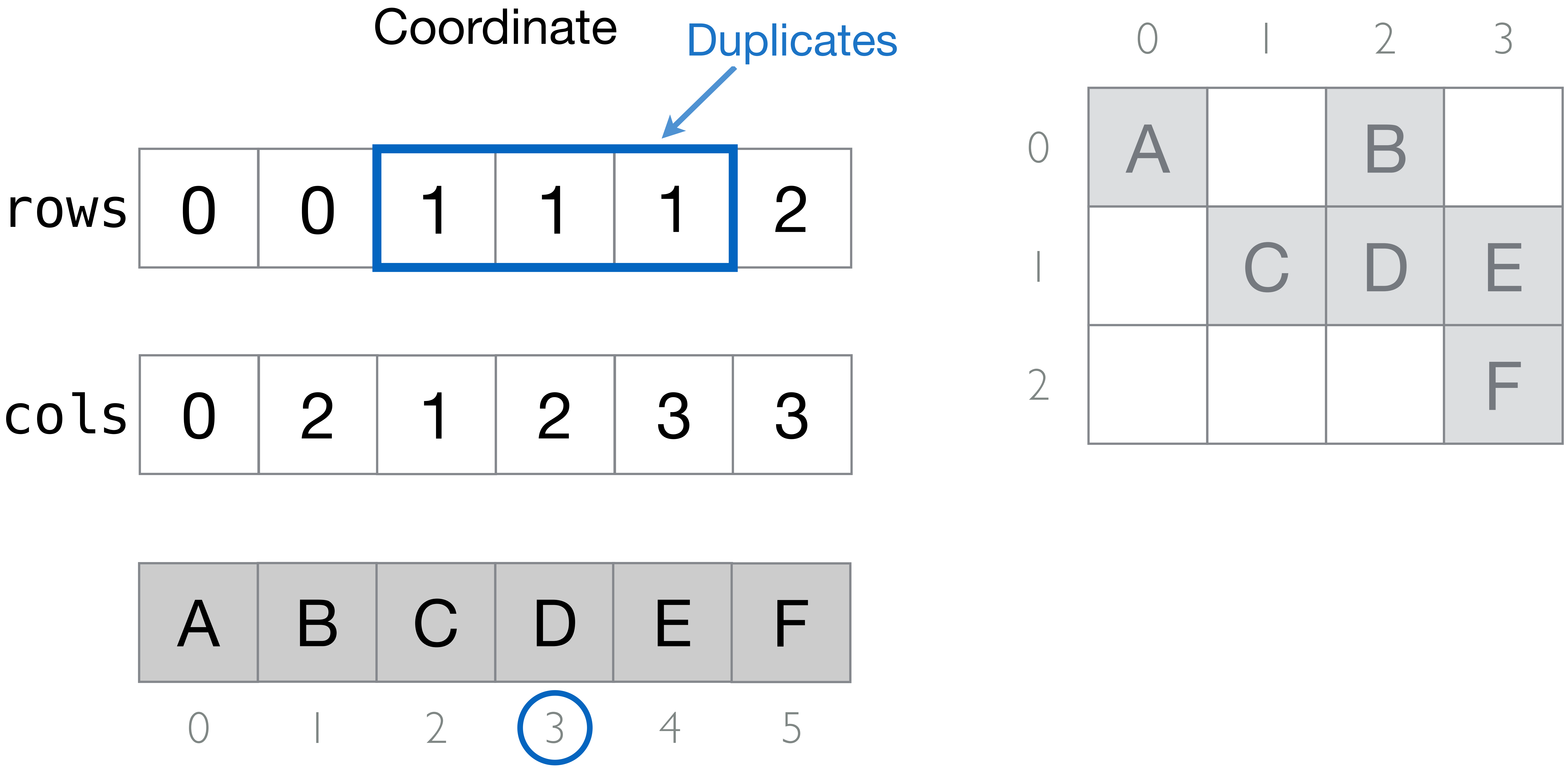
Coordinate relations → coordinate trees



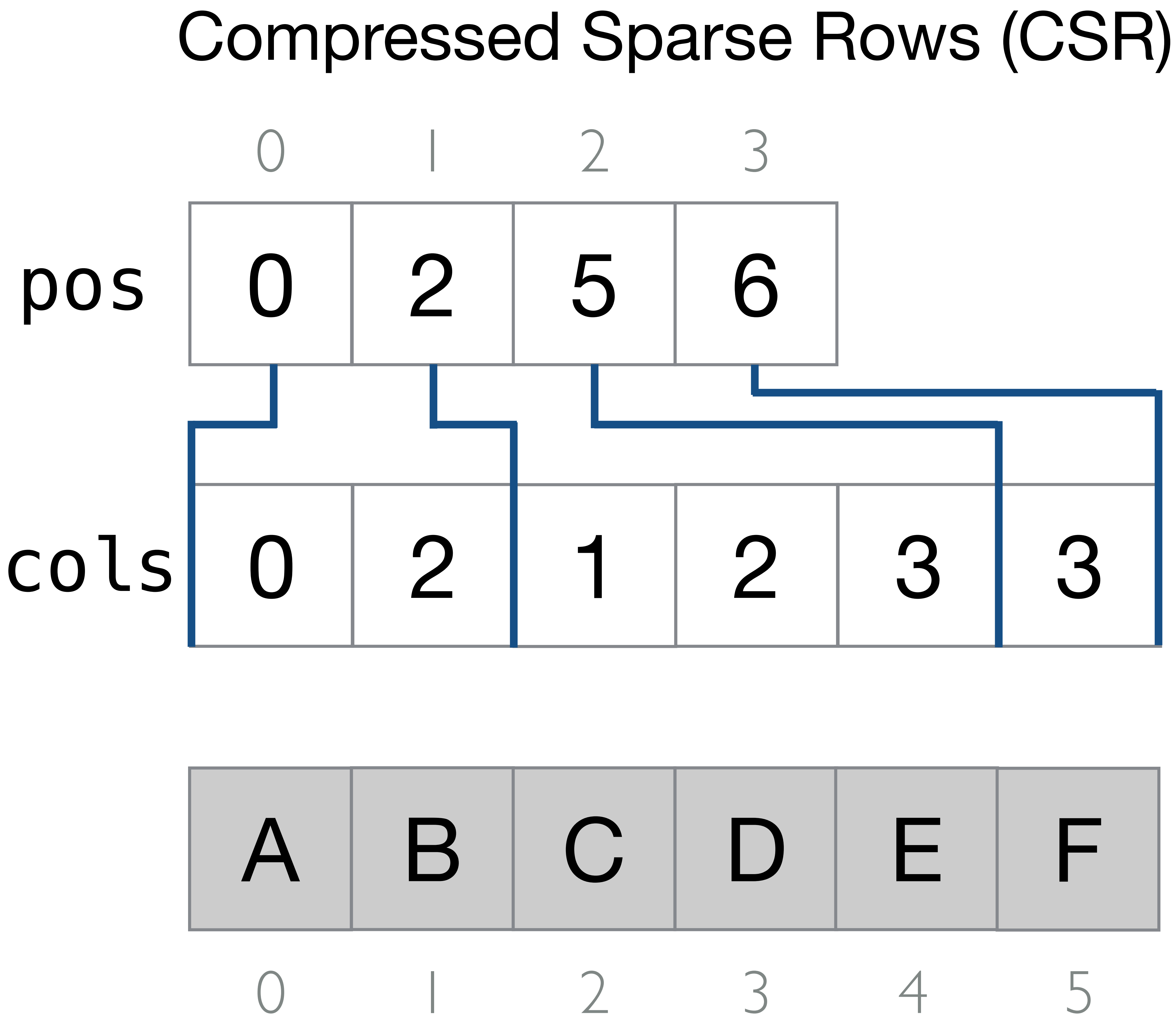
Coordinate relations → coordinate trees

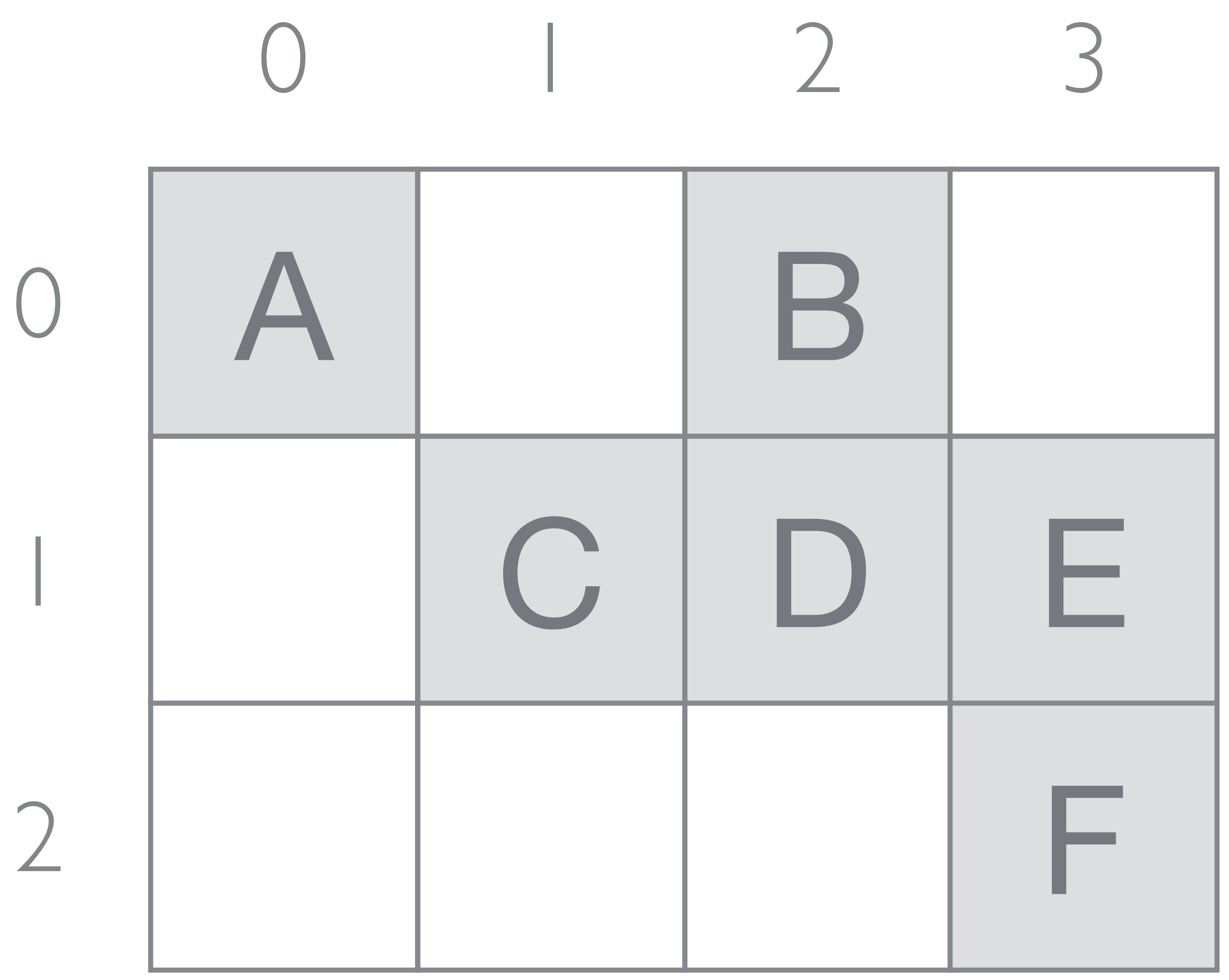


Coordinate relations → coordinate trees

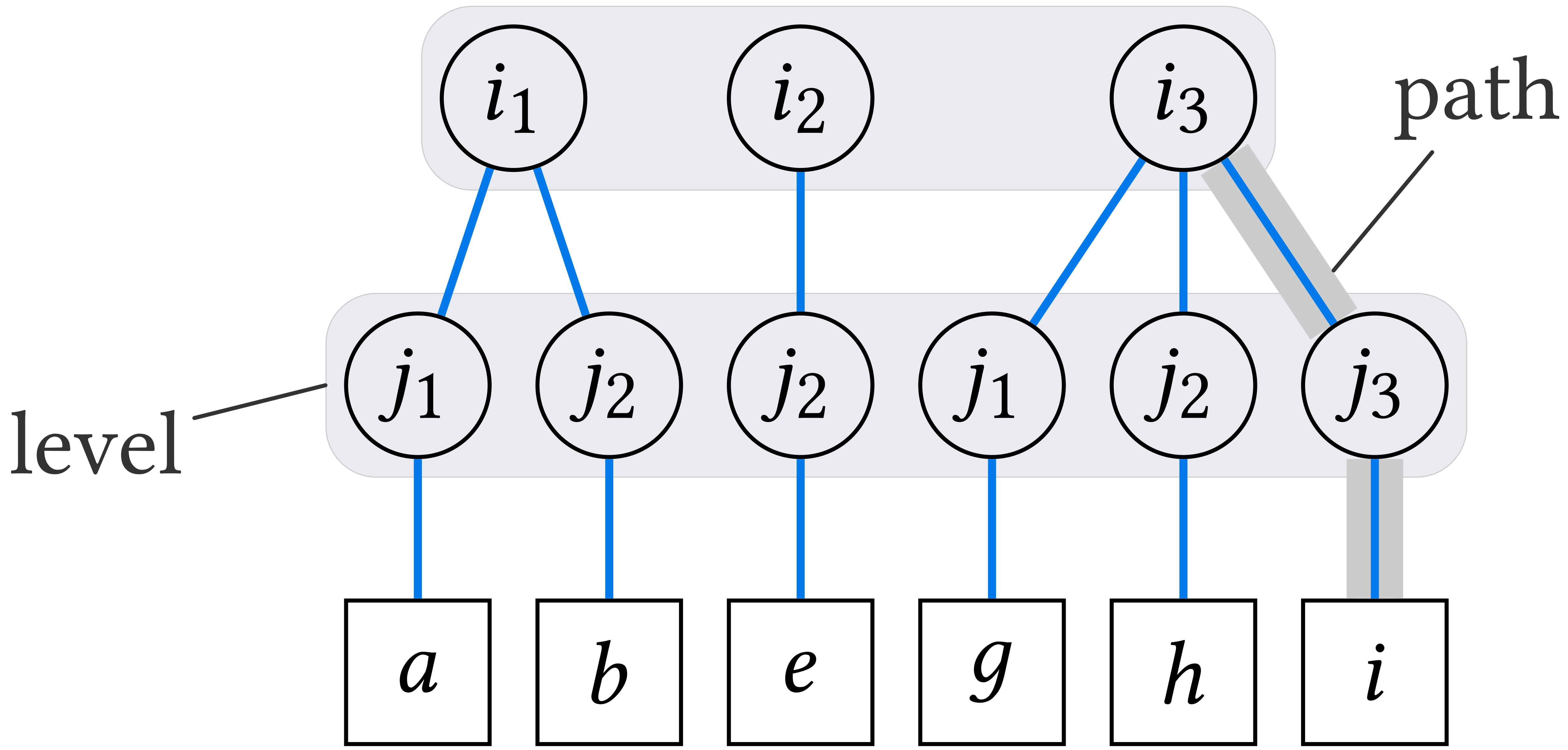


Coordinate relations → coordinate trees



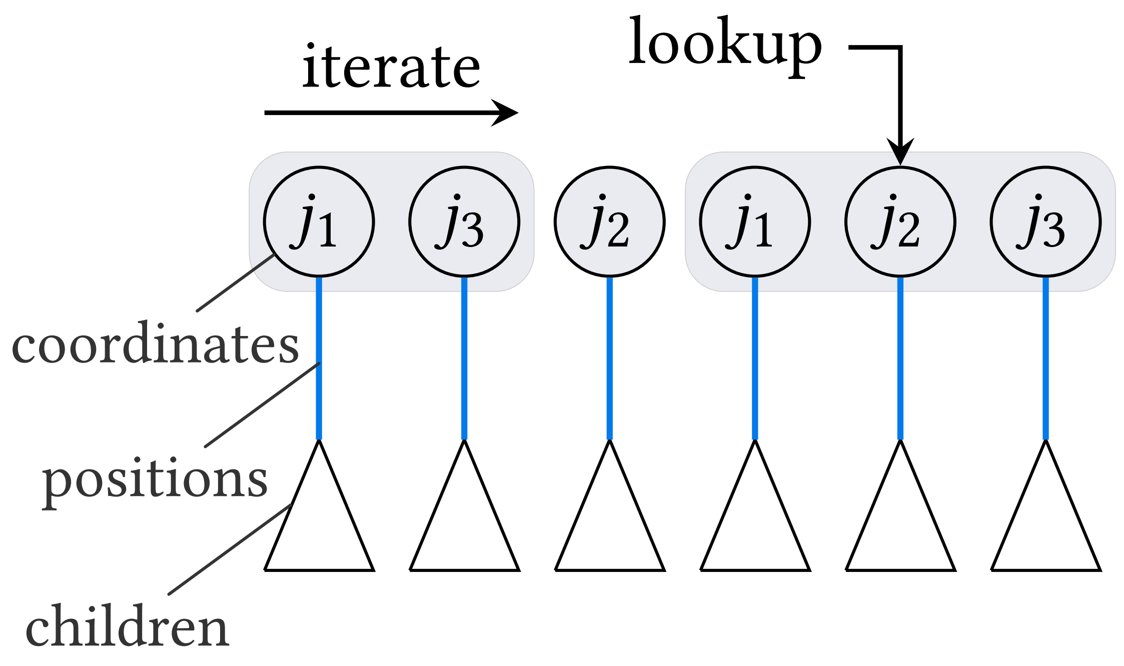


Level-based representation

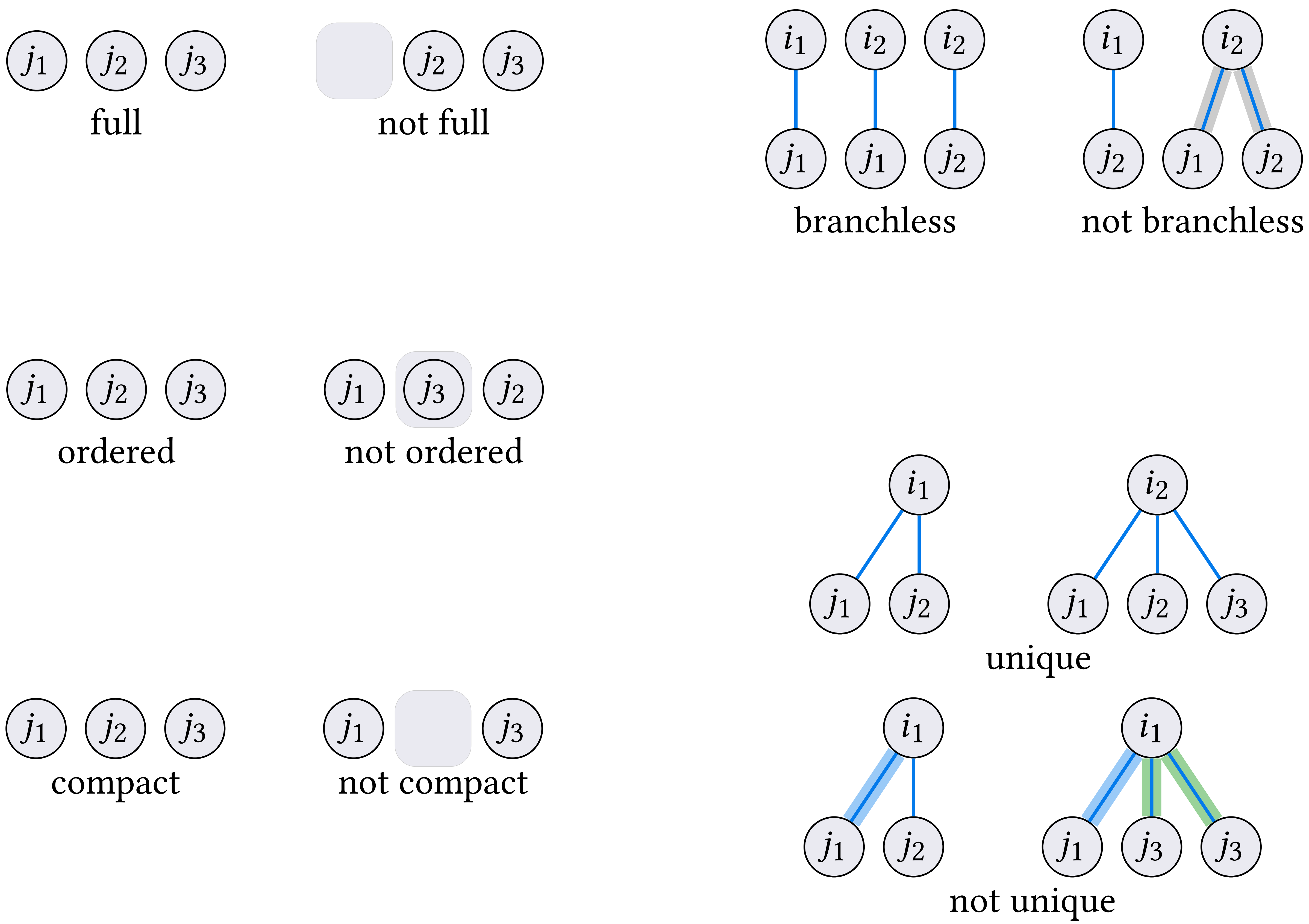


Level abstraction: capabilities and properties

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 <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>
```

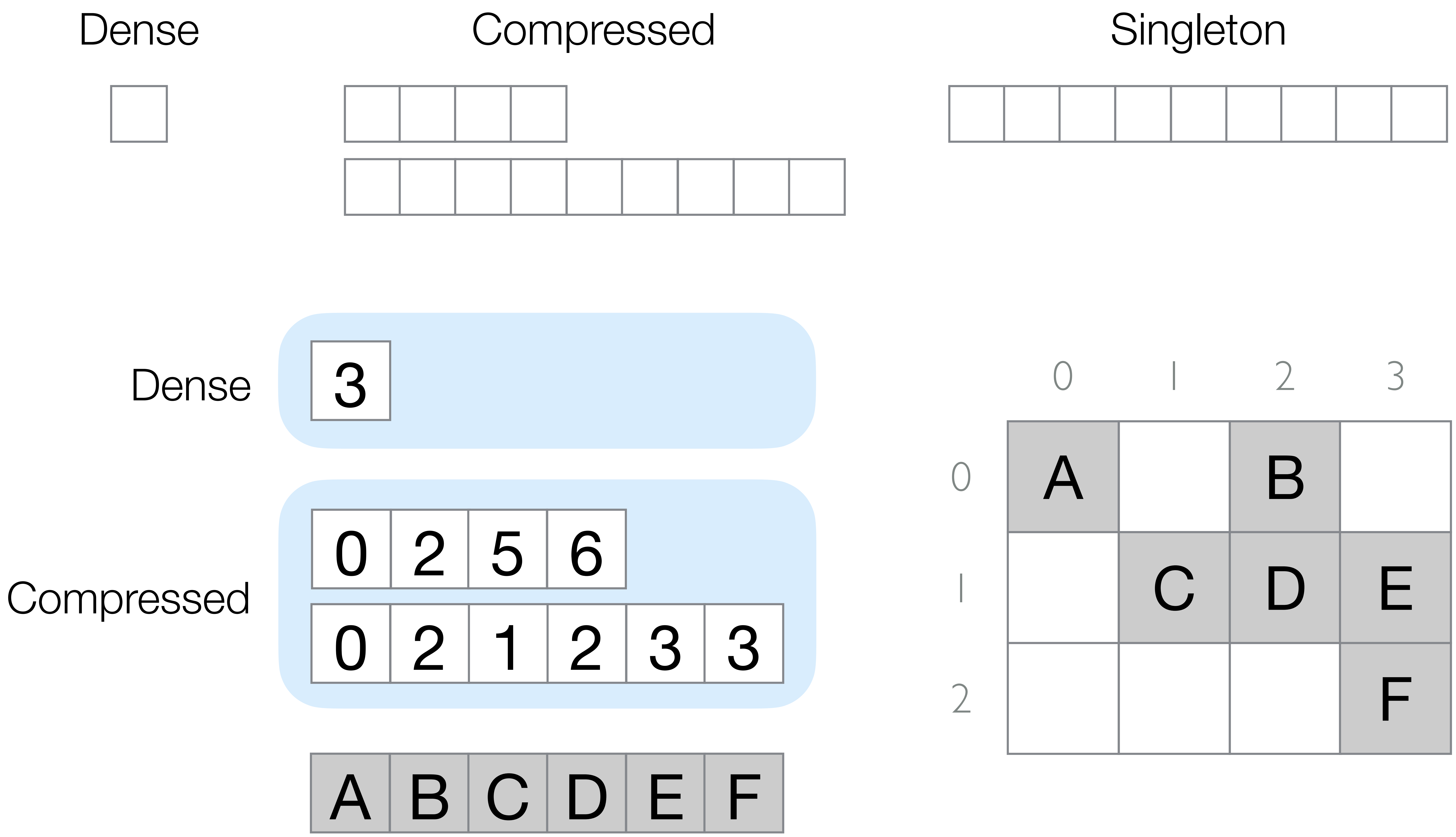
$$y = Ax$$

```
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];
  }
}
```

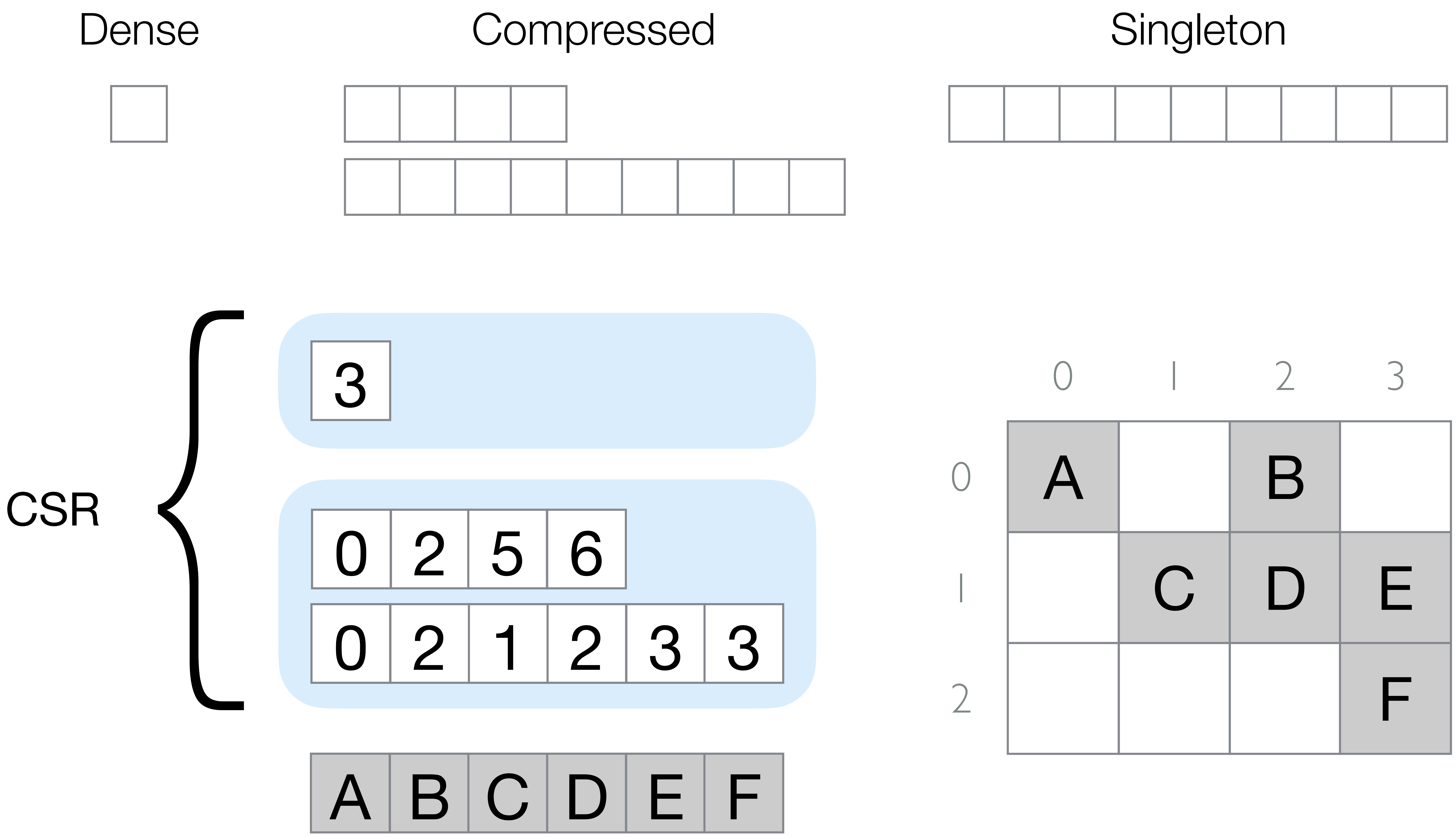
Compressed iterate

Dense locate

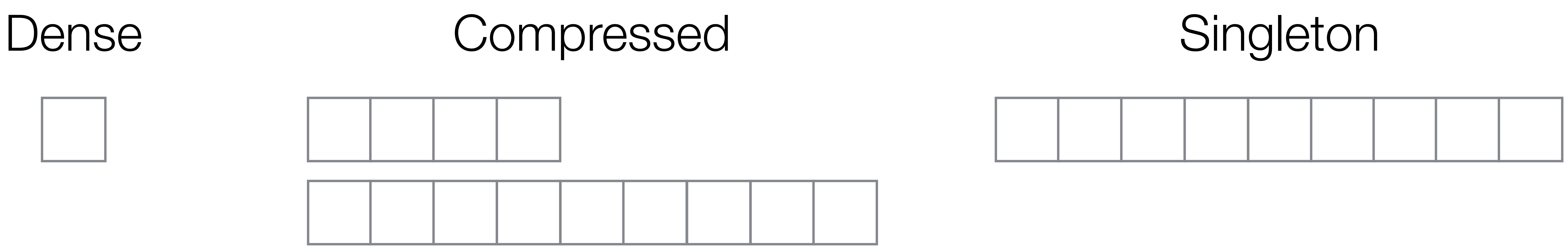
Level types can be composed in many ways

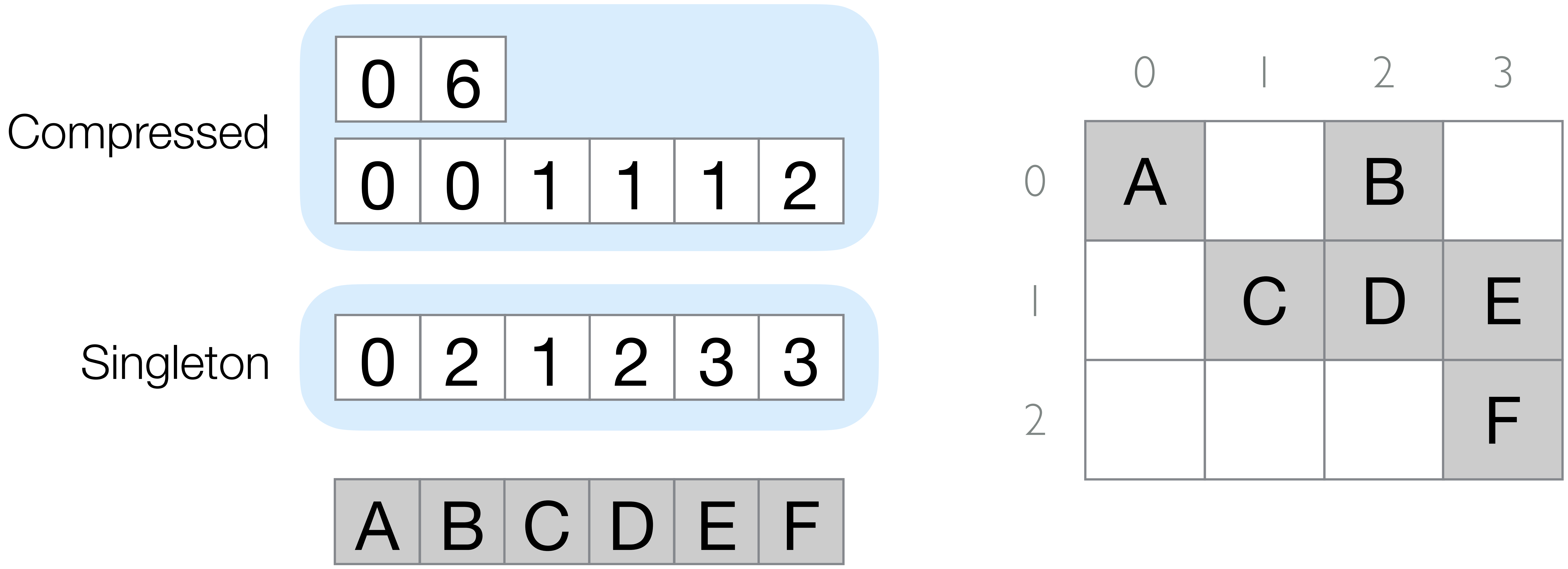


Level types can be composed in many ways

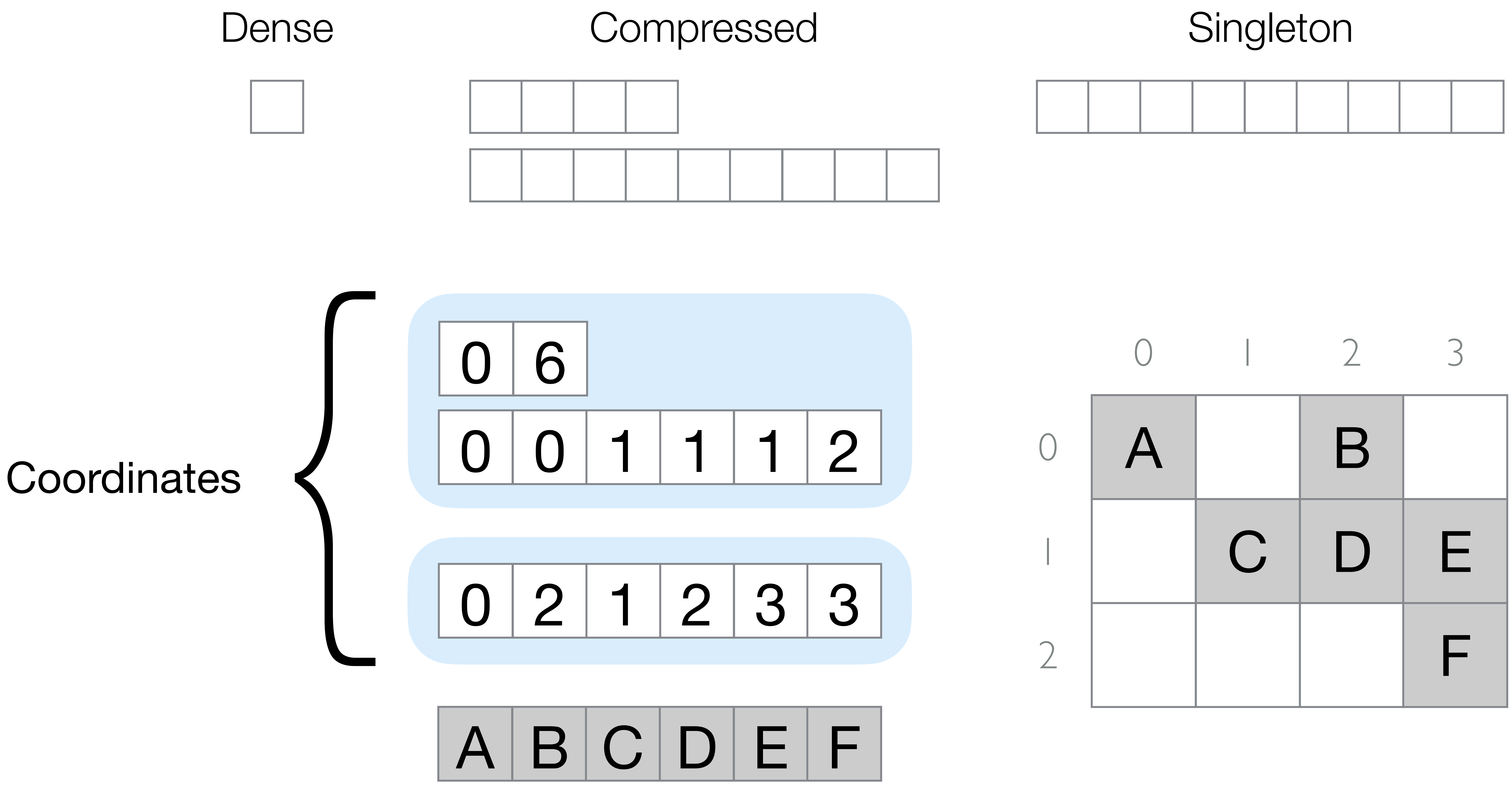


Level types can be composed in many ways





Level types can be composed in many ways



Level types can be composed in many ways

Level formats

Dense	Compressed	Singleton
Hashed	Range	Offset

Tensor formats

Coordinate matrix
Compressed
Singleton

CSR
Dense
Compressed

[Tinney and Walker, 1967]

Dense array tensor
Dense
Dense
Dense

Coordinate tensor
Compressed
Singleton
Singleton

Mode-generic tensor
Compressed
Singleton
Dense
Dense

[Baskaran et al. 2012]

BCSR
Dense
Compressed
Dense
Dense

[Im and Yelick 1998]

CSB
Dense
Dense
Compressed
Singleton

[Buluç et al. 2009]

ELLPACK
Dense
Dense
Singleton

[Kincaid et al. 1989]

Hash map vector
Hashed

[Patwary et al. 2015]

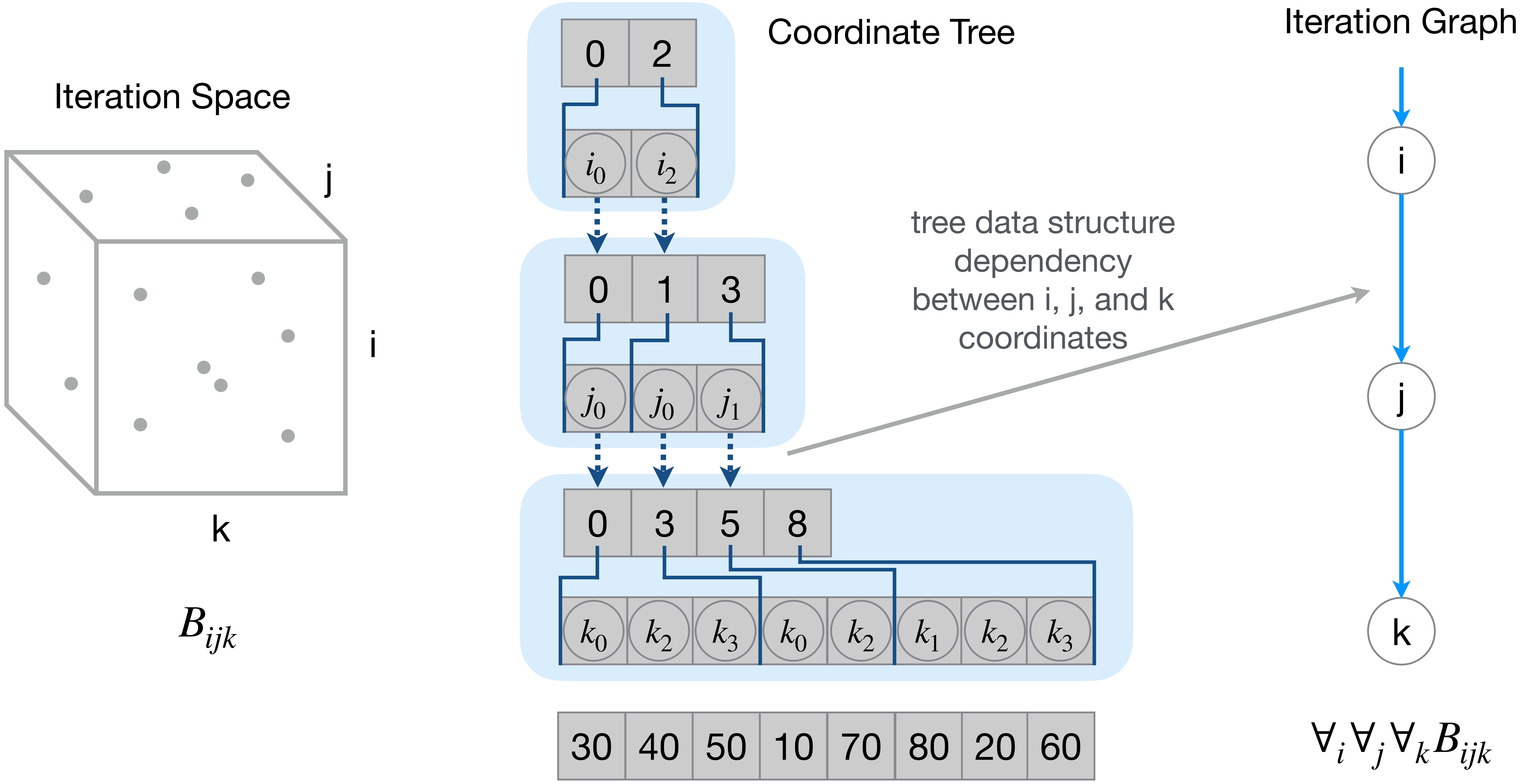
Hash map matrix
Hashed
Hashed

DIA
Dense
Range
Offset

[Saad 2003]

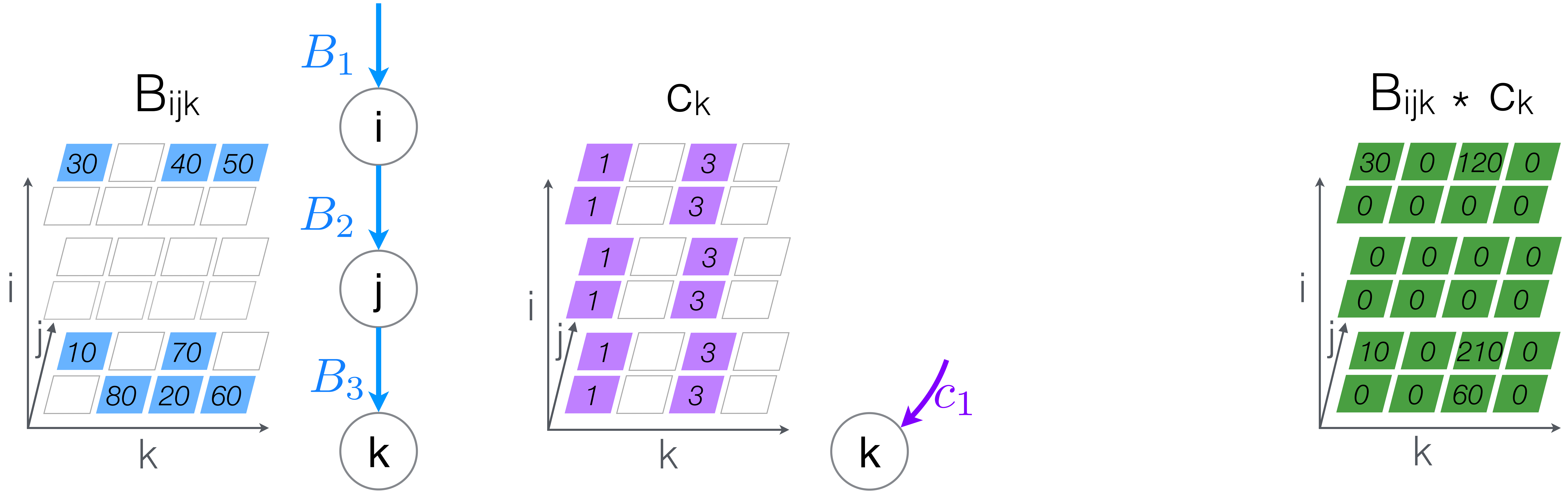
Block DIA
Dense
Range
Offset
Dense
Dense

Iteration graphs express iteration spaces and data structure ordering



Sparse iteration spaces and Iteration Graphs

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

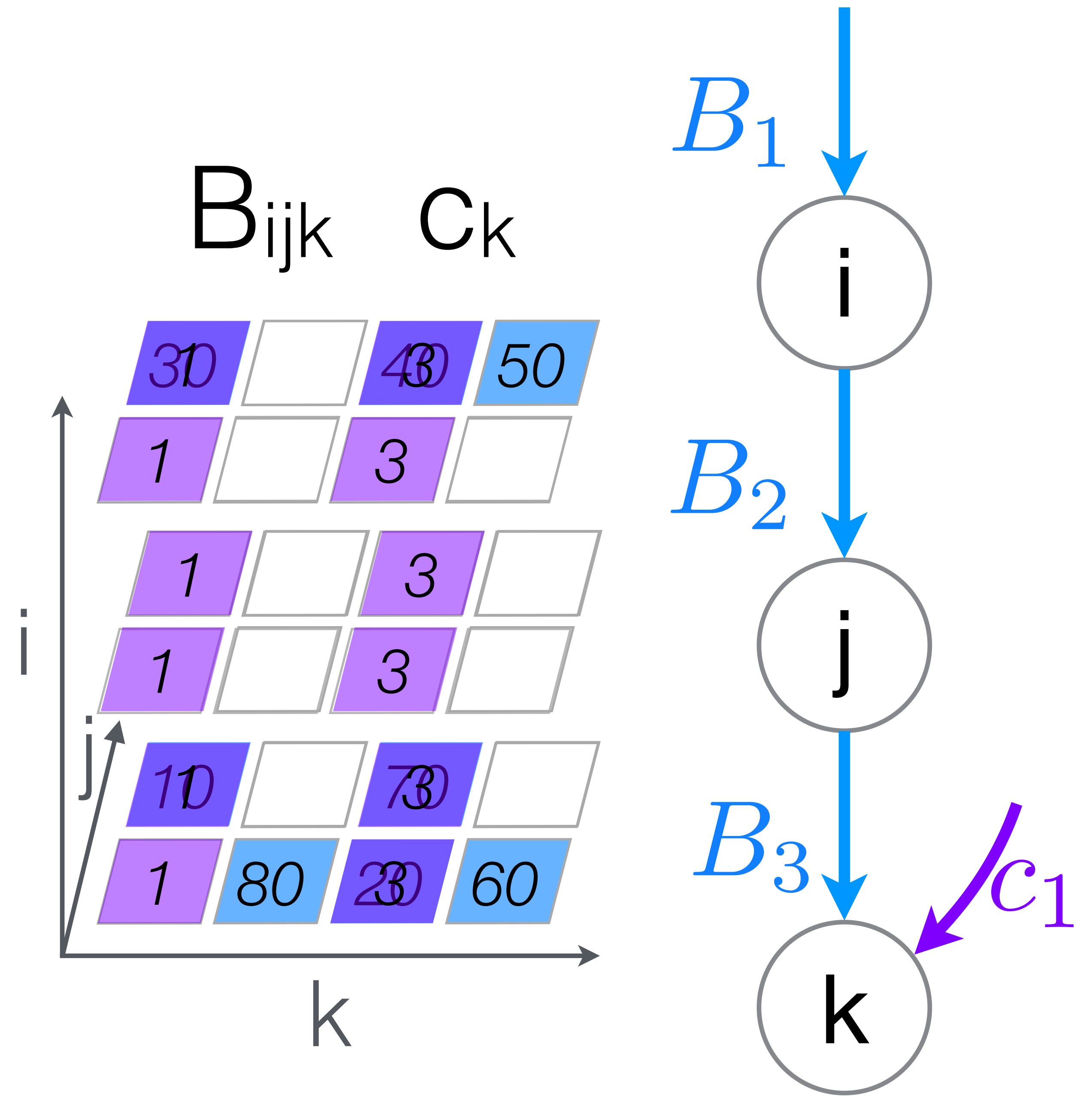


Sparse

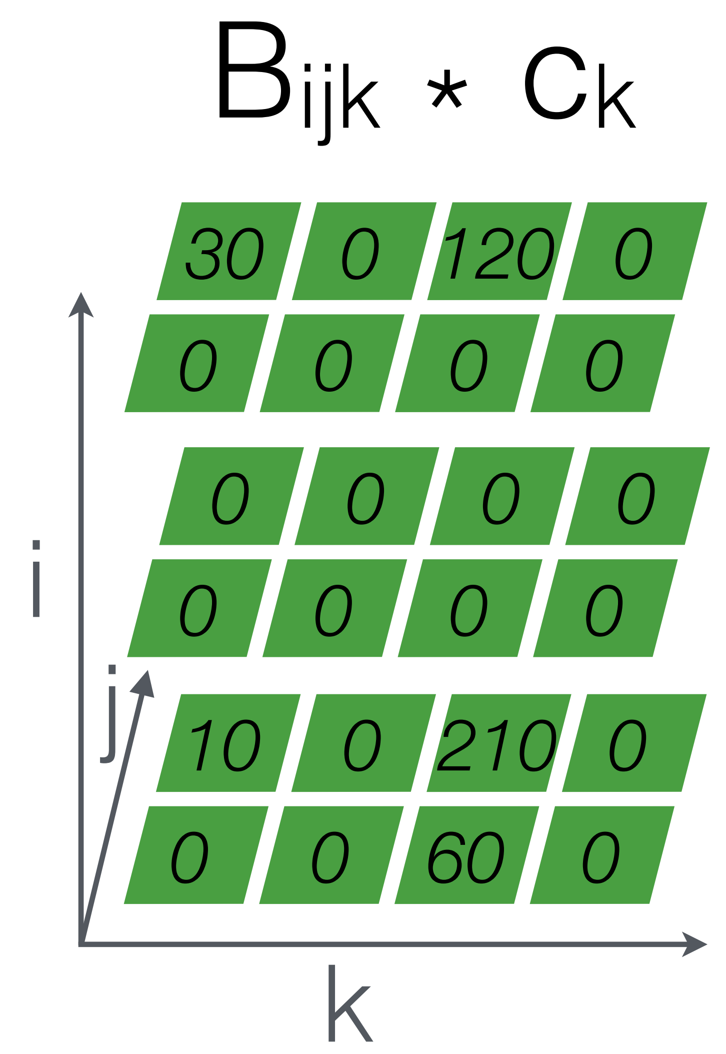
Dense

Sparse iteration spaces and Iteration Graphs

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



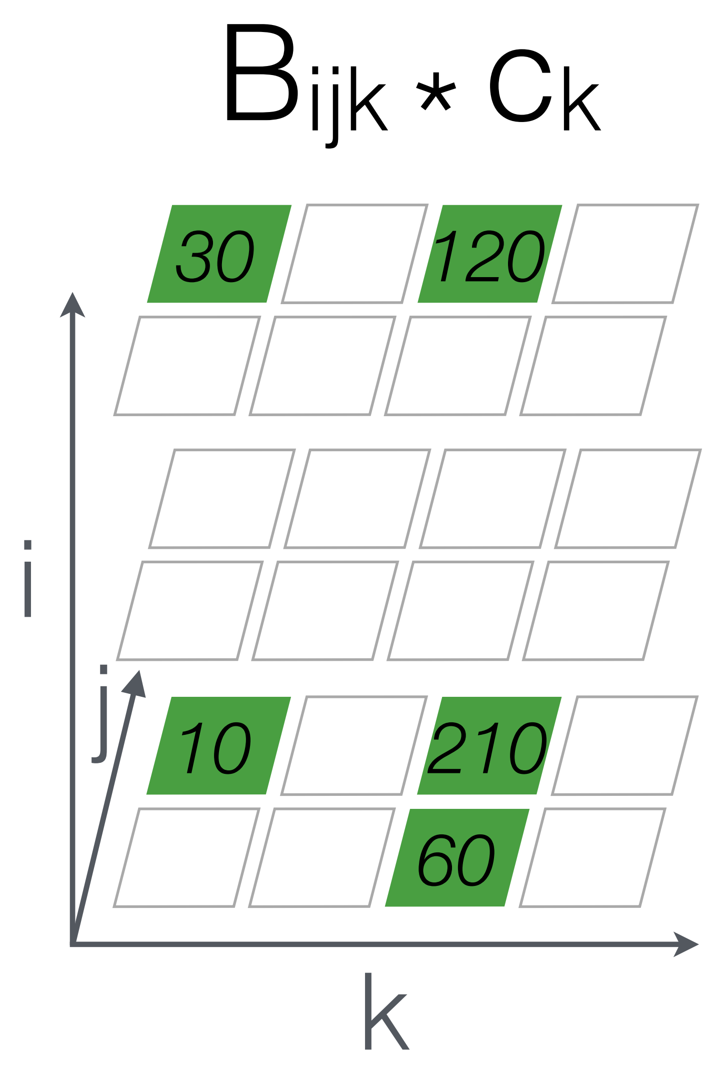
Sparse

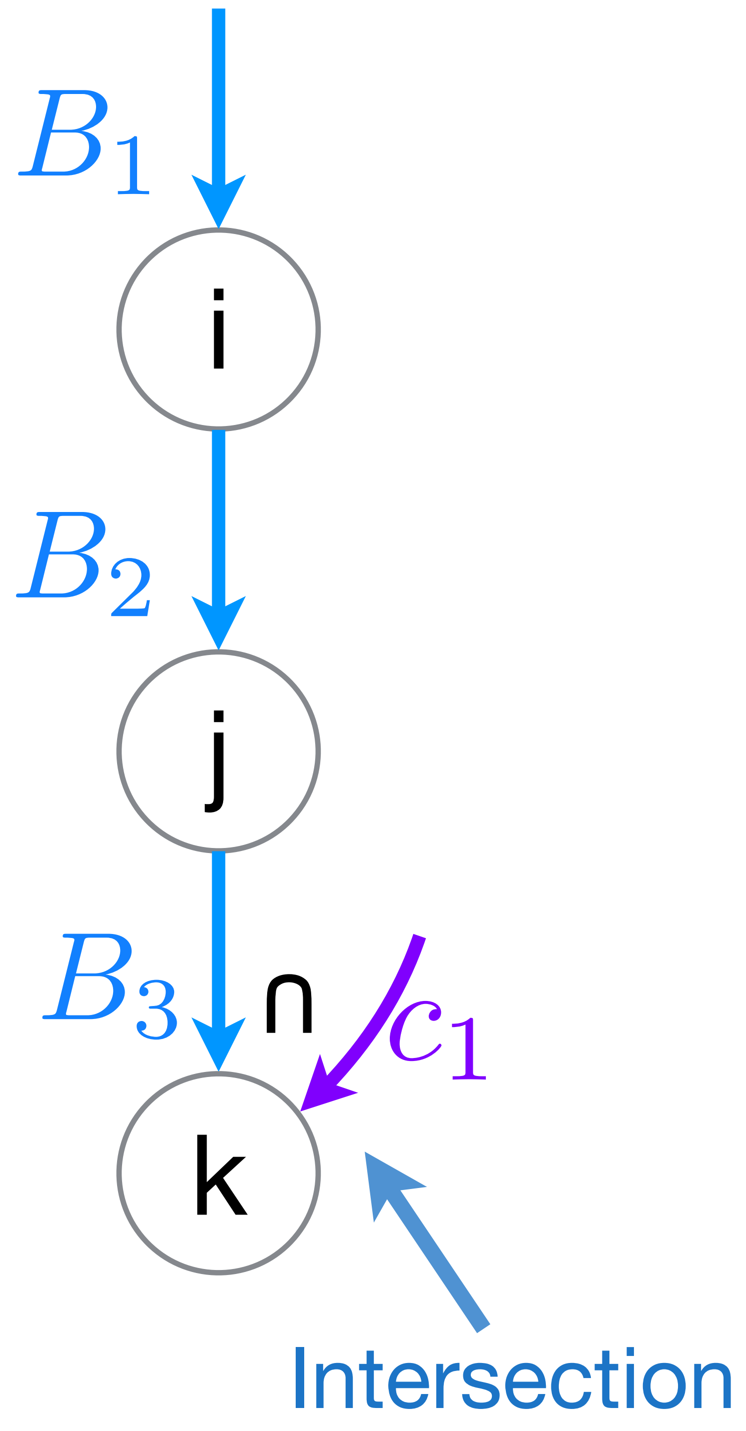


Dense

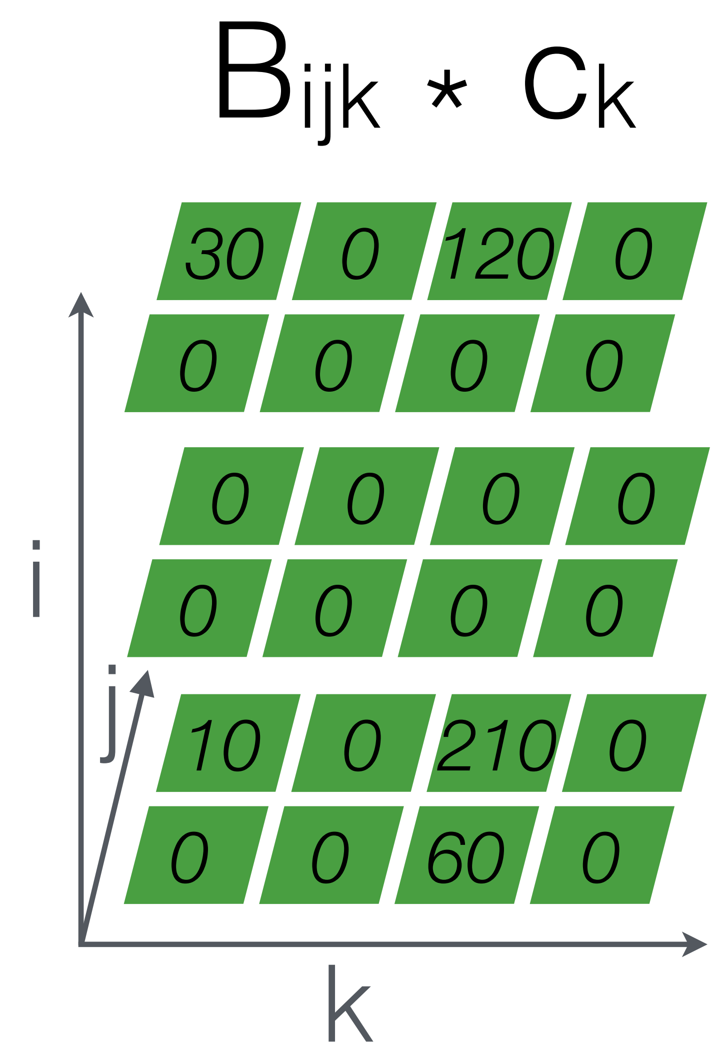
Sparse iteration spaces and Iteration Graphs

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





Sparse



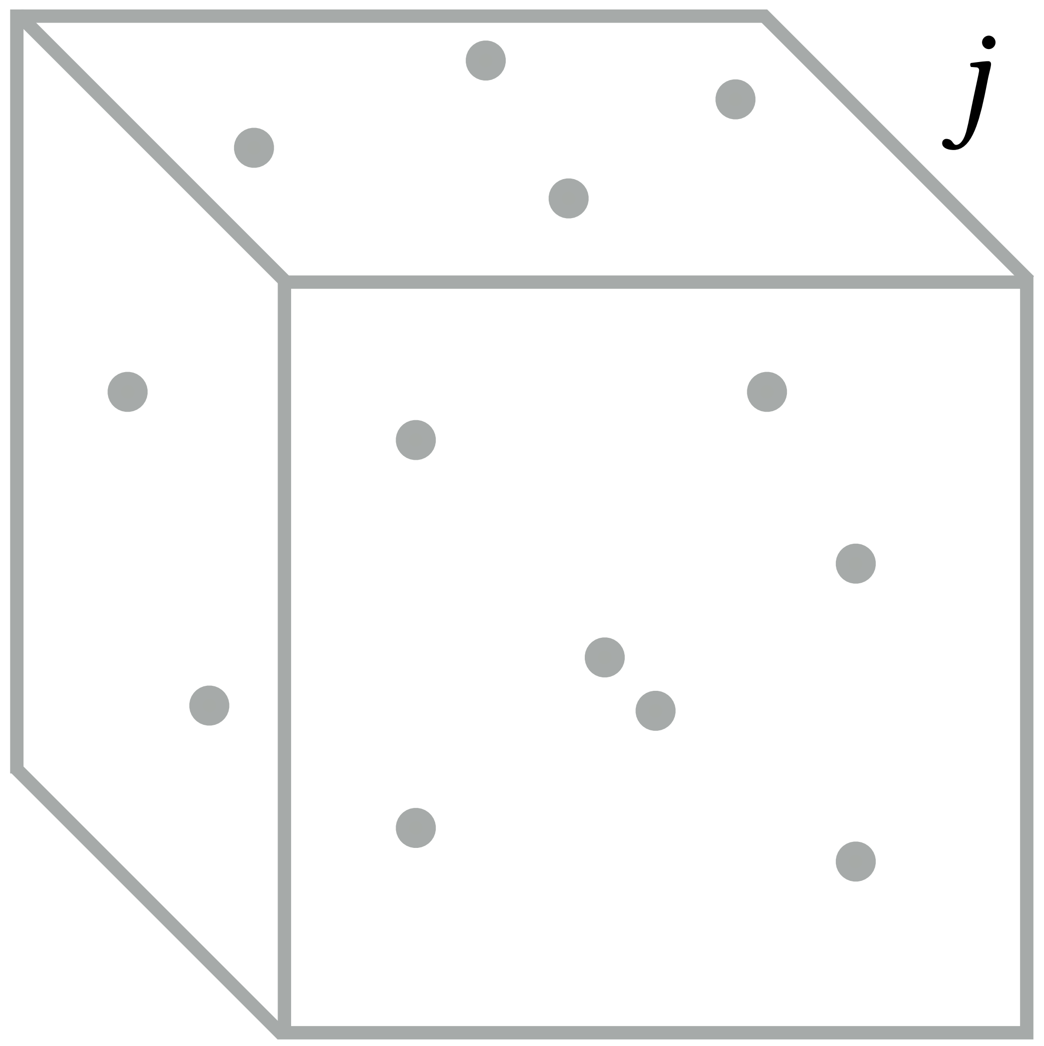
Dense

Per-dimension iteration expressions

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

Sparse Iteration Space

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



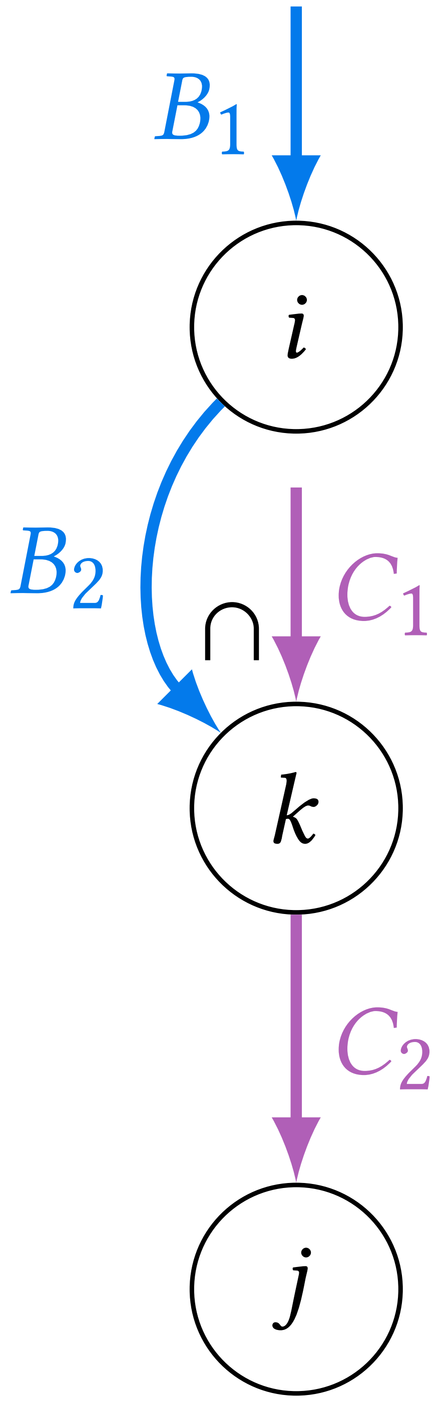
$j \in \mathbb{U}_j \cap C_2$

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

$k \in B_2 \cap C_1$

Sparse Iteration Graph

$\forall_i \forall_k \forall_j \; B_{ik} \cap C_{kj}$

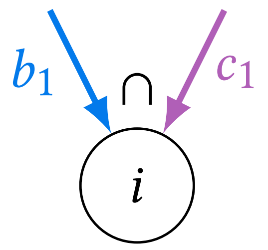


$i \in B_1 \cap \mathbb{U}_i$

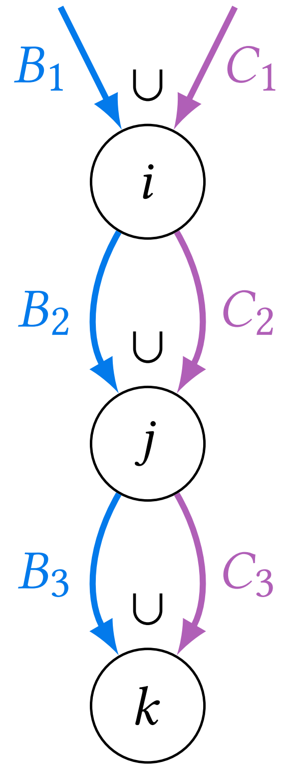
$k \in B_2 \cap C_1$

$j \in \mathbb{U}_j \cap C_2$

Iteration graph examples



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

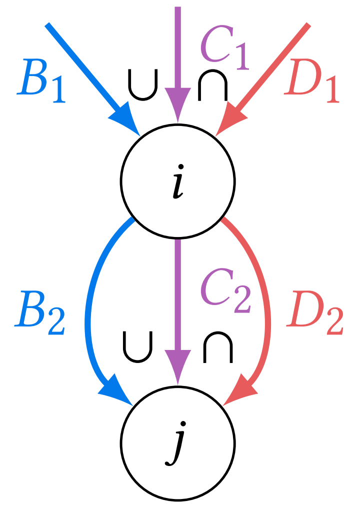


$$\forall_i \forall_j \forall_k\ B_{ijk} \cup C_{ijk}$$

$i \in B_1 \cup C_1$

$j \in B_2 \cup C_2$

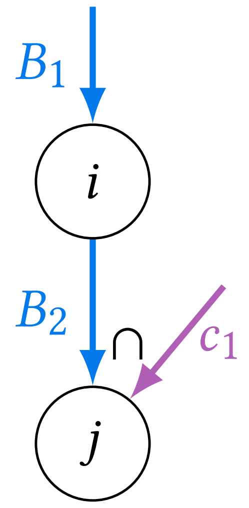
$k \in B_3 \cup C_3$



$$\forall_i \forall_j\ (B_{ij} \cup C_{ij}) \cap D_{ij}$$

$i \in (B_1 \cup C_1) \cap D_1$

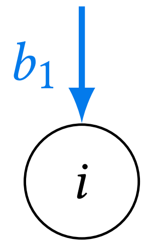
$j \in (B_2 \cup C_2) \cap D_2$



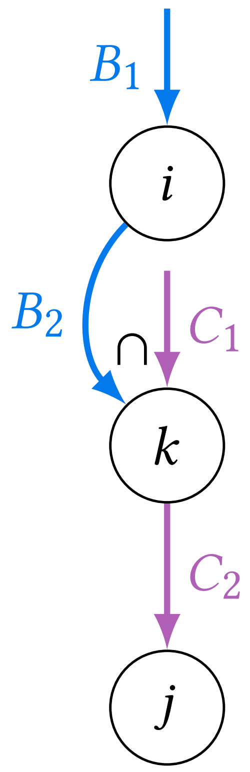
$$\forall_i \forall_j\ B_{ij} \cap c_j$$

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

$j \in B_2 \cap c_1$



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

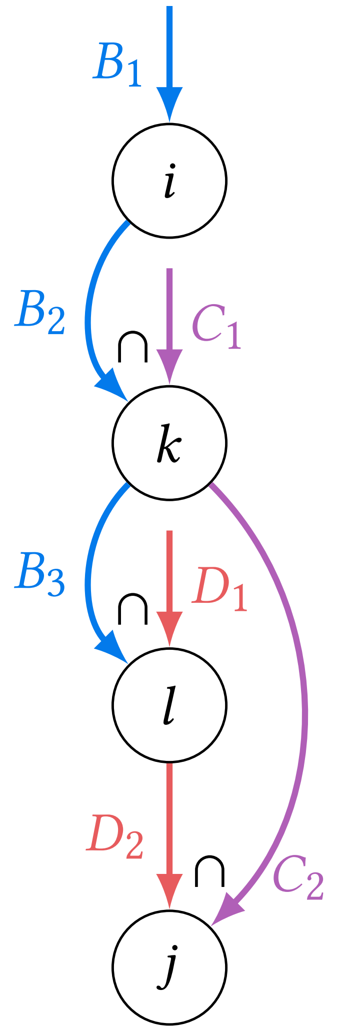


$$\forall_i \forall_k \forall_j\ B_{ik} \cap C_{kj}$$

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

$k \in B_2 \cap C_1$

$j \in \mathbb{U}_j \cap C_2$



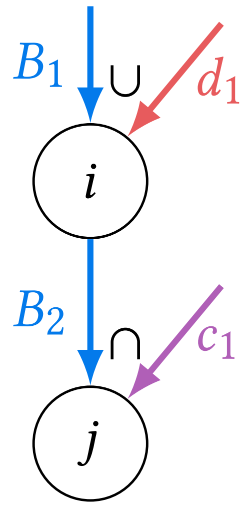
$$\forall_i \forall_k \forall_l \forall_j\ B_{ikl} \cap C_{kj} \cap D_{lj}$$

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

$k \in B_2 \cap C_1 \cap \mathbb{U}_k$

$l \in B_3 \cap \mathbb{U}_l \cap D_1$

$j \in \mathbb{U}_j \cap C_2 \cap D_2$



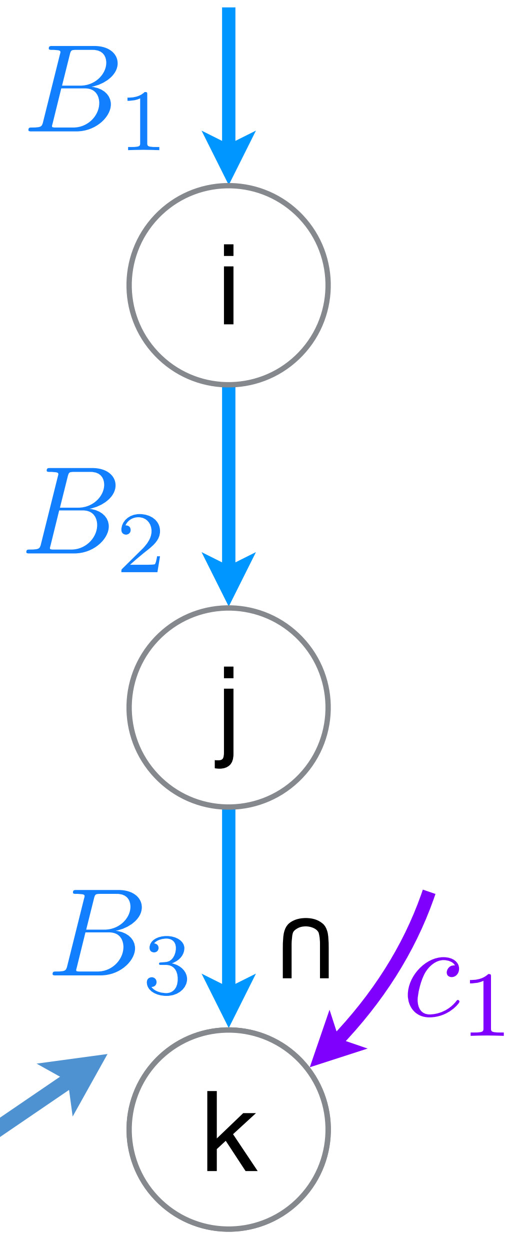
$$\forall_i (\forall_j\ B_{ij} \cap c_j) \cup d_i$$

$i \in (B_1 \cap \mathbb{U}_i) \cup d_1$

$j \in B_2 \cap c_1$

Iteration graphs are lowered to sparse code

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

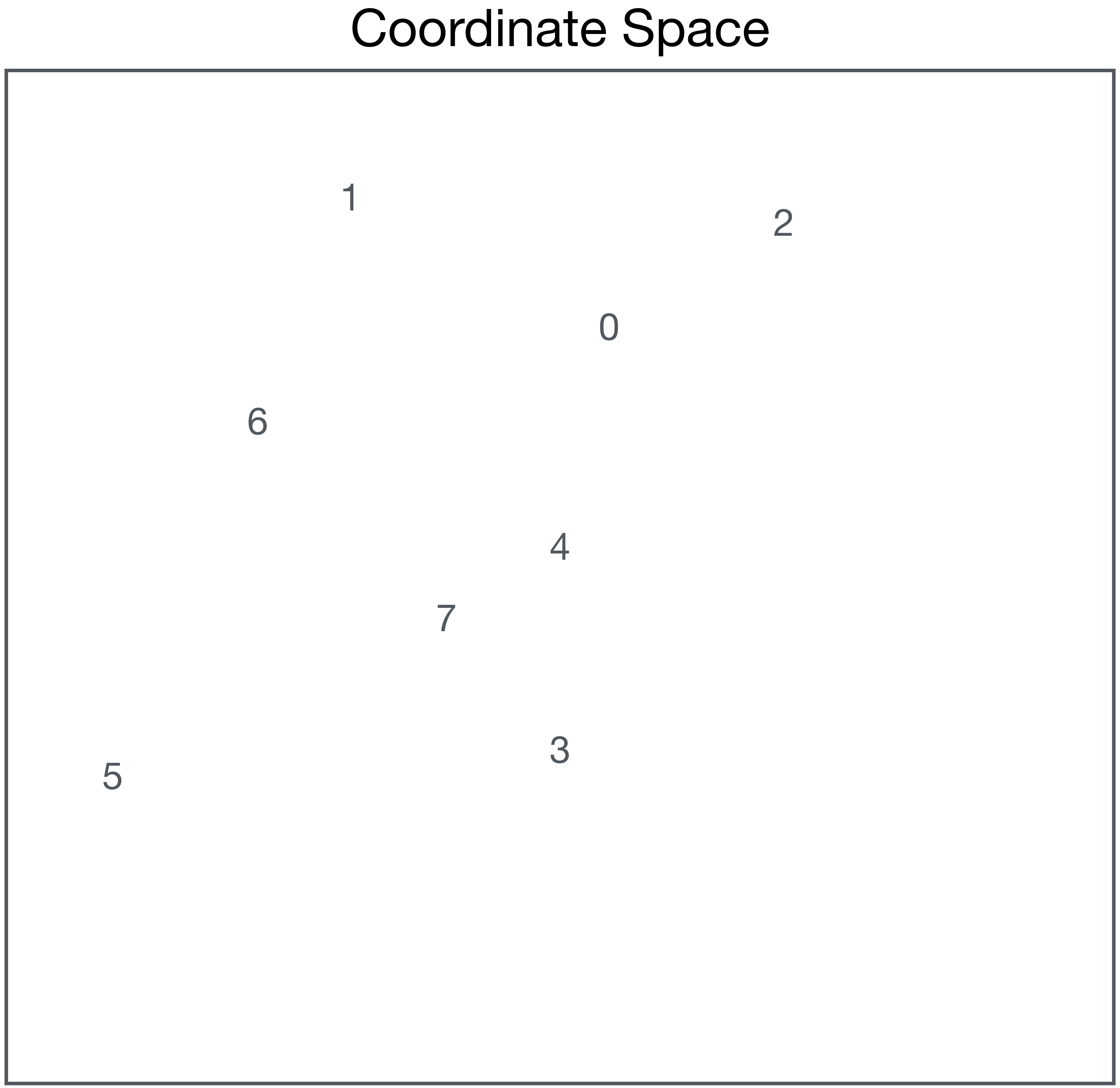


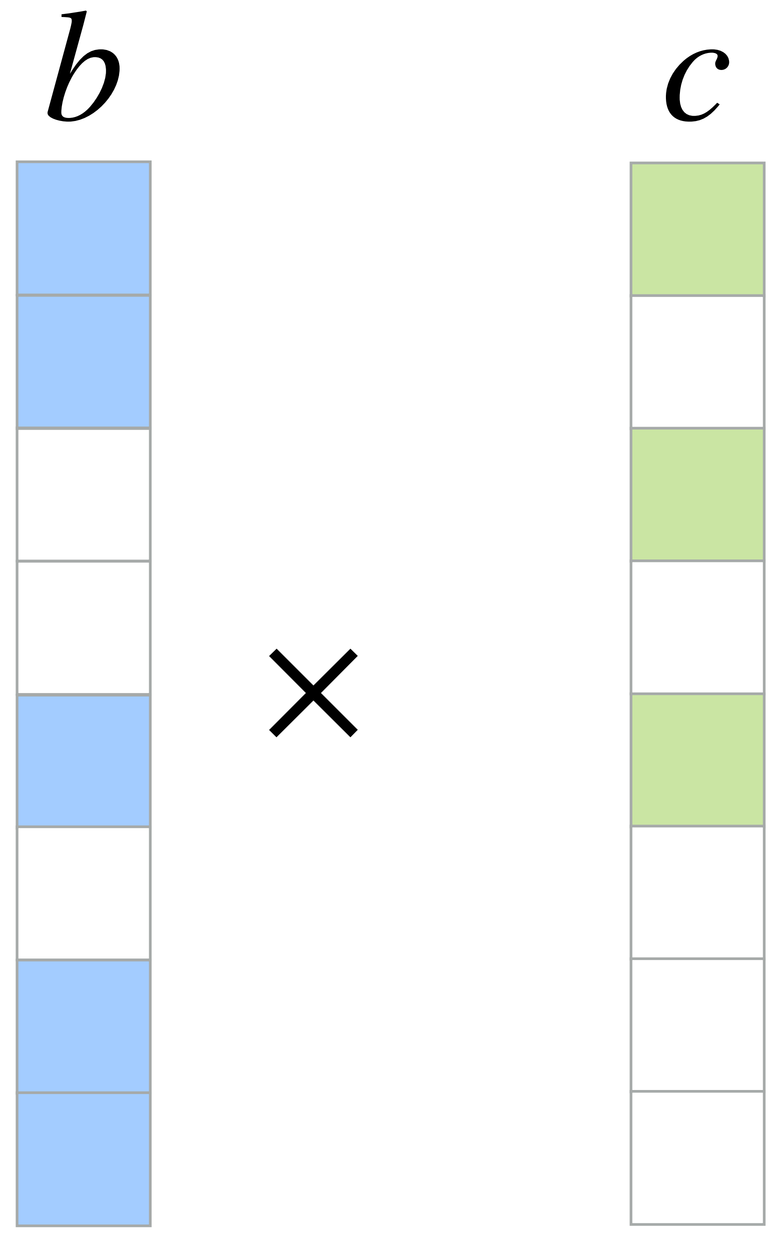

```
for (int i = 0; i < m; i++) {

  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

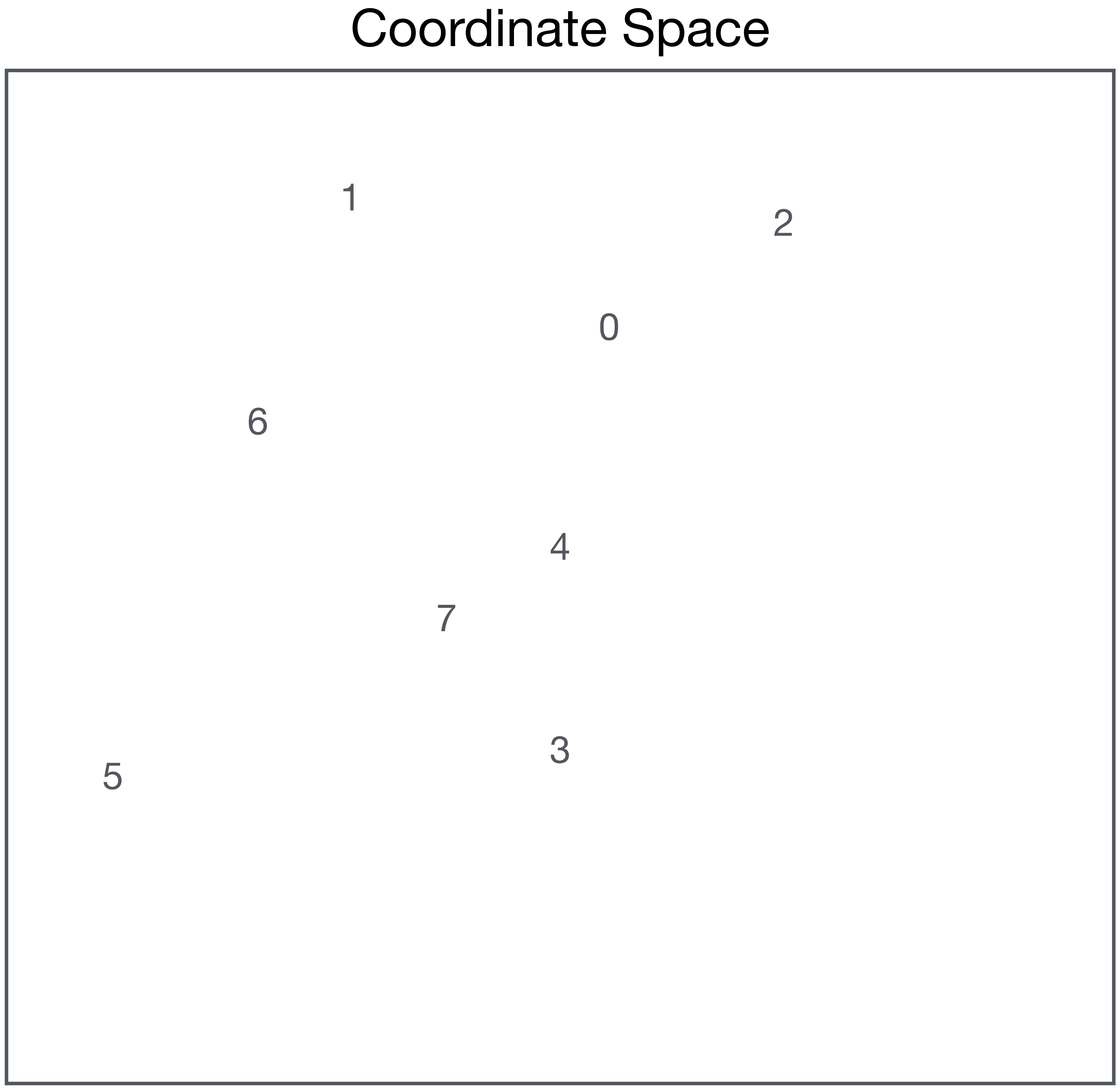
Intersection coiteration

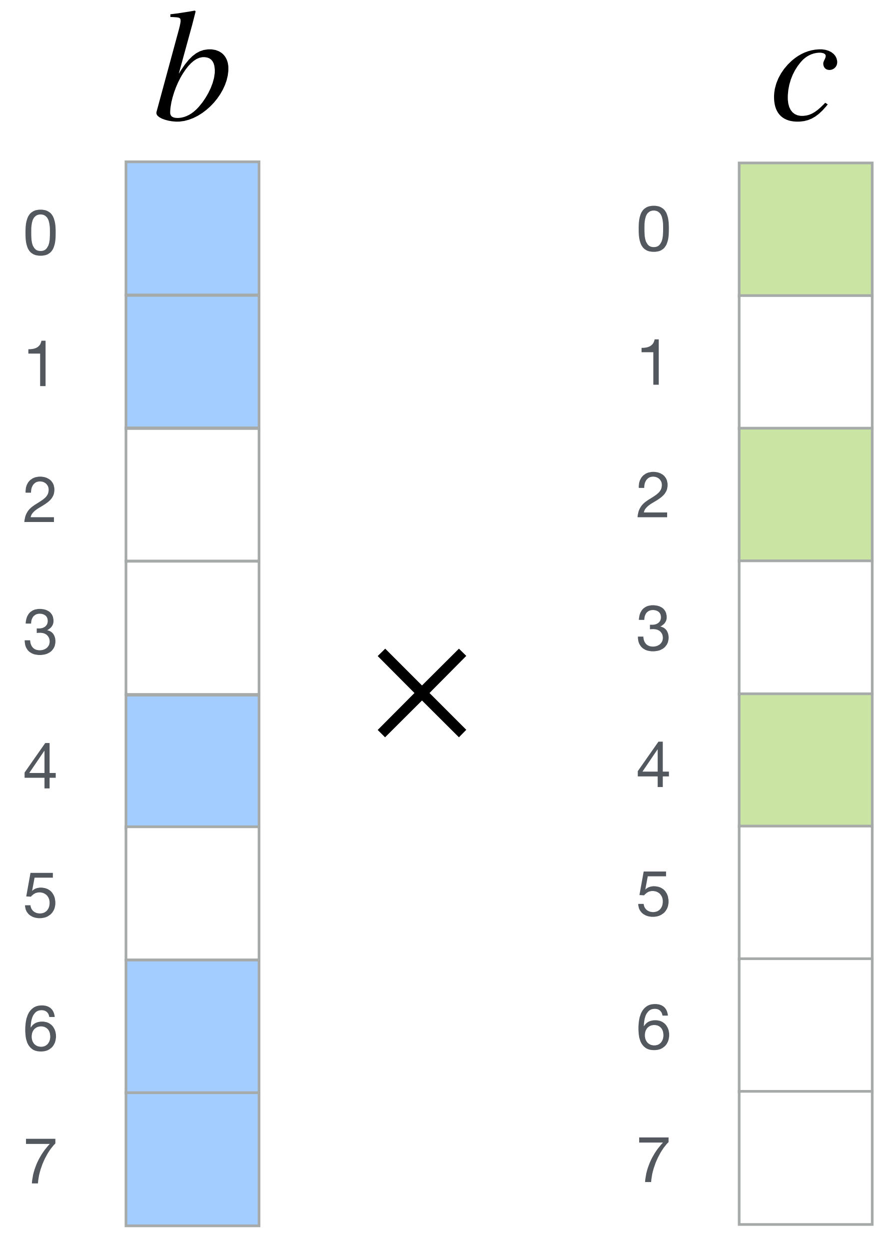
Data structure coiteration



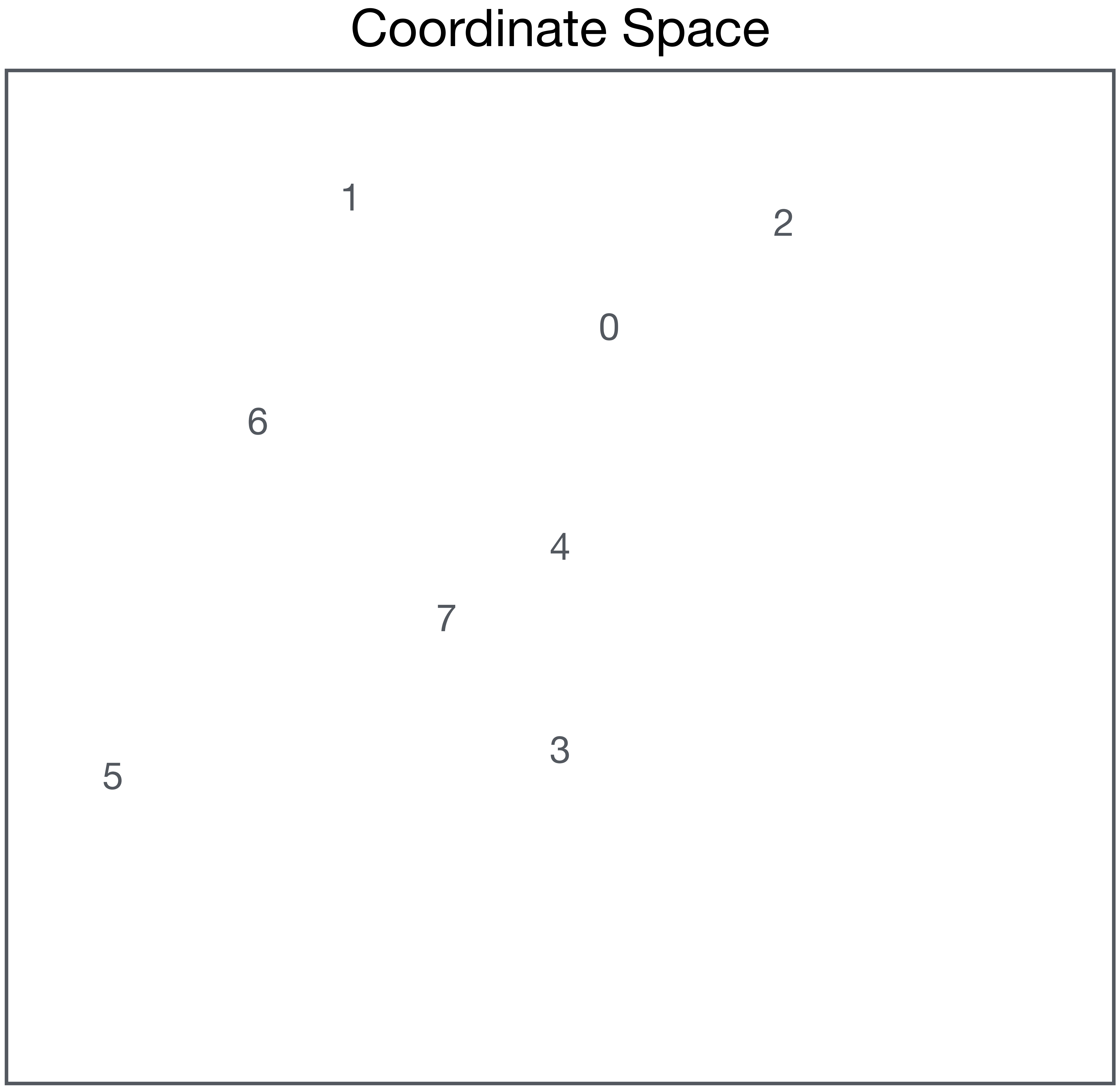


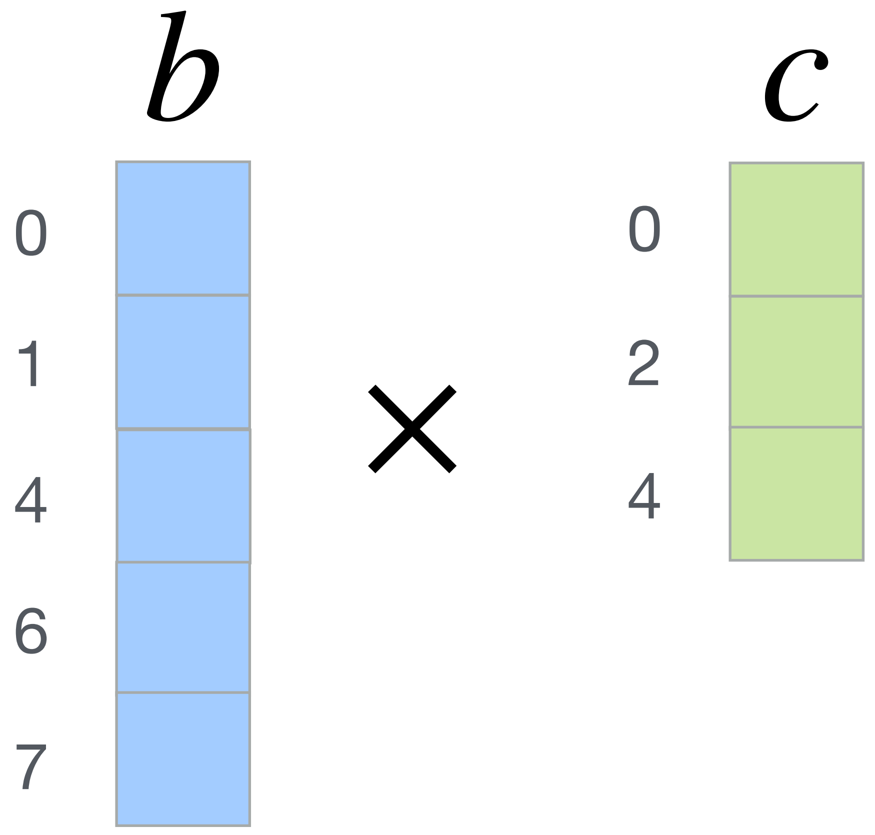
Data structure coiteration



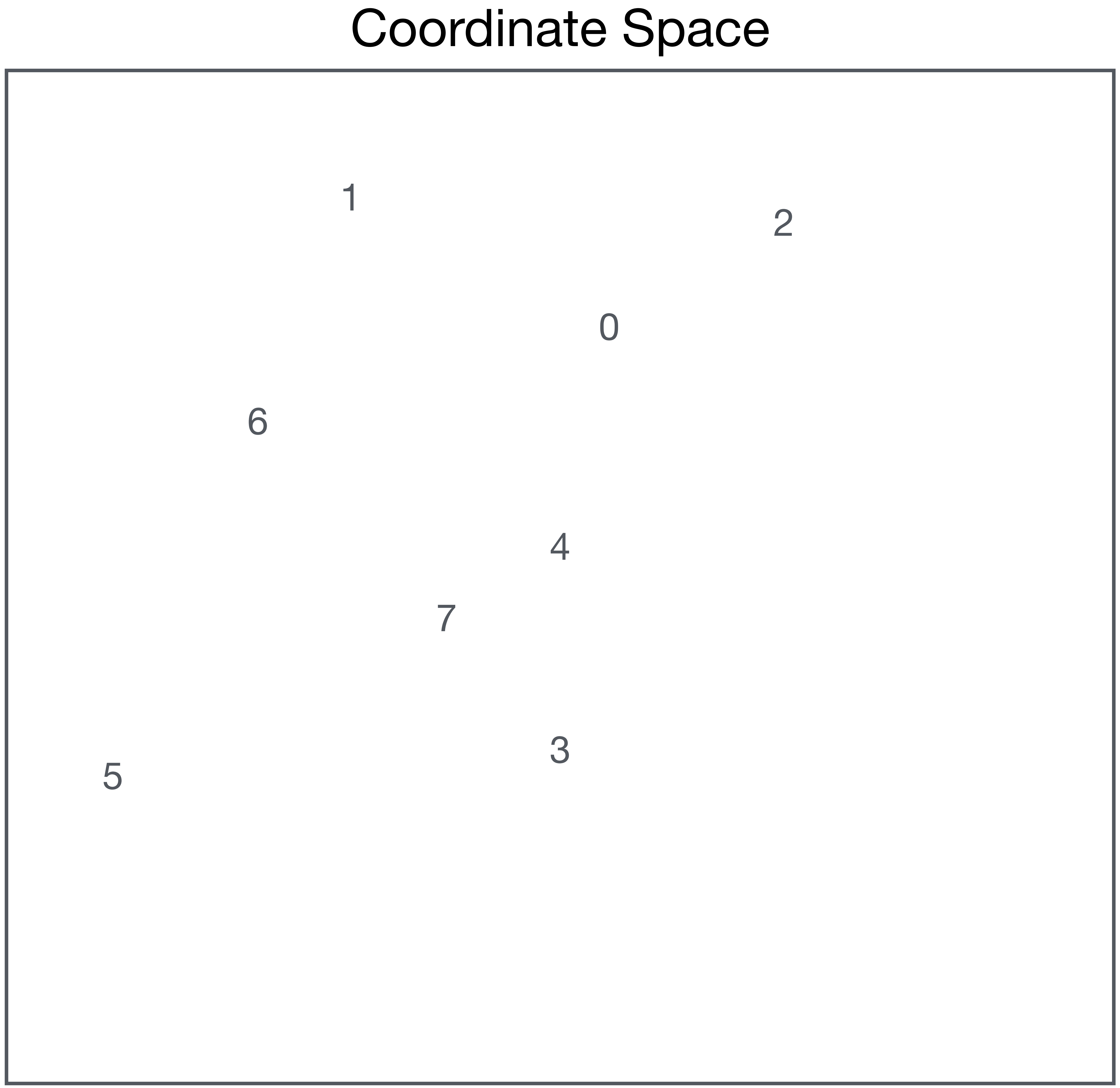


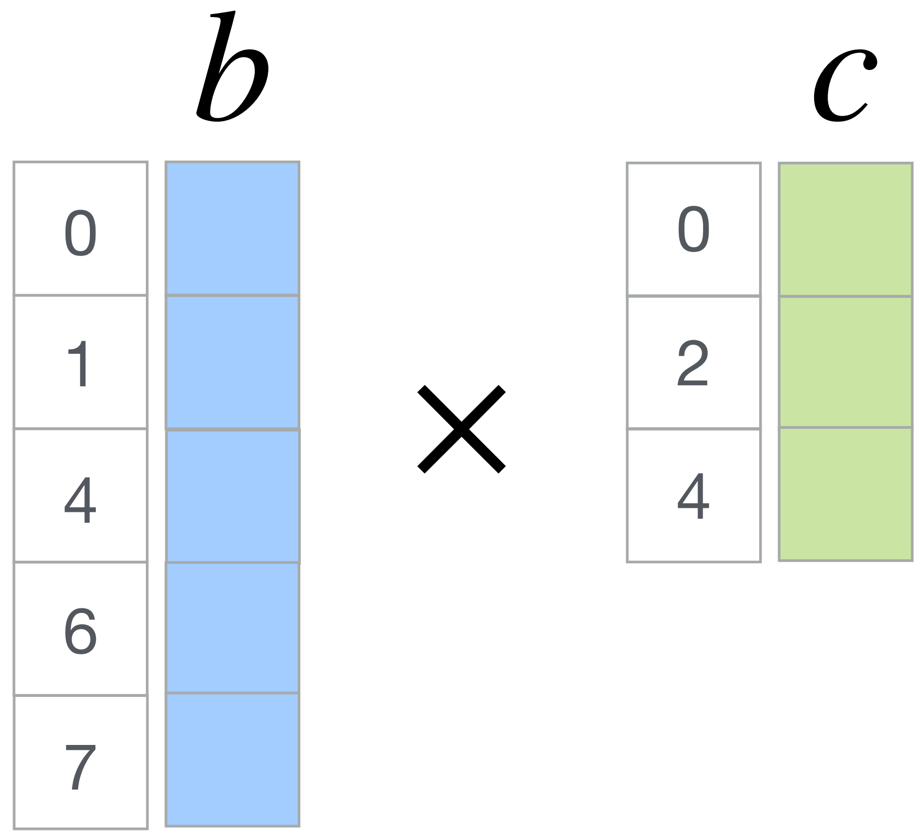
Data structure coiteration



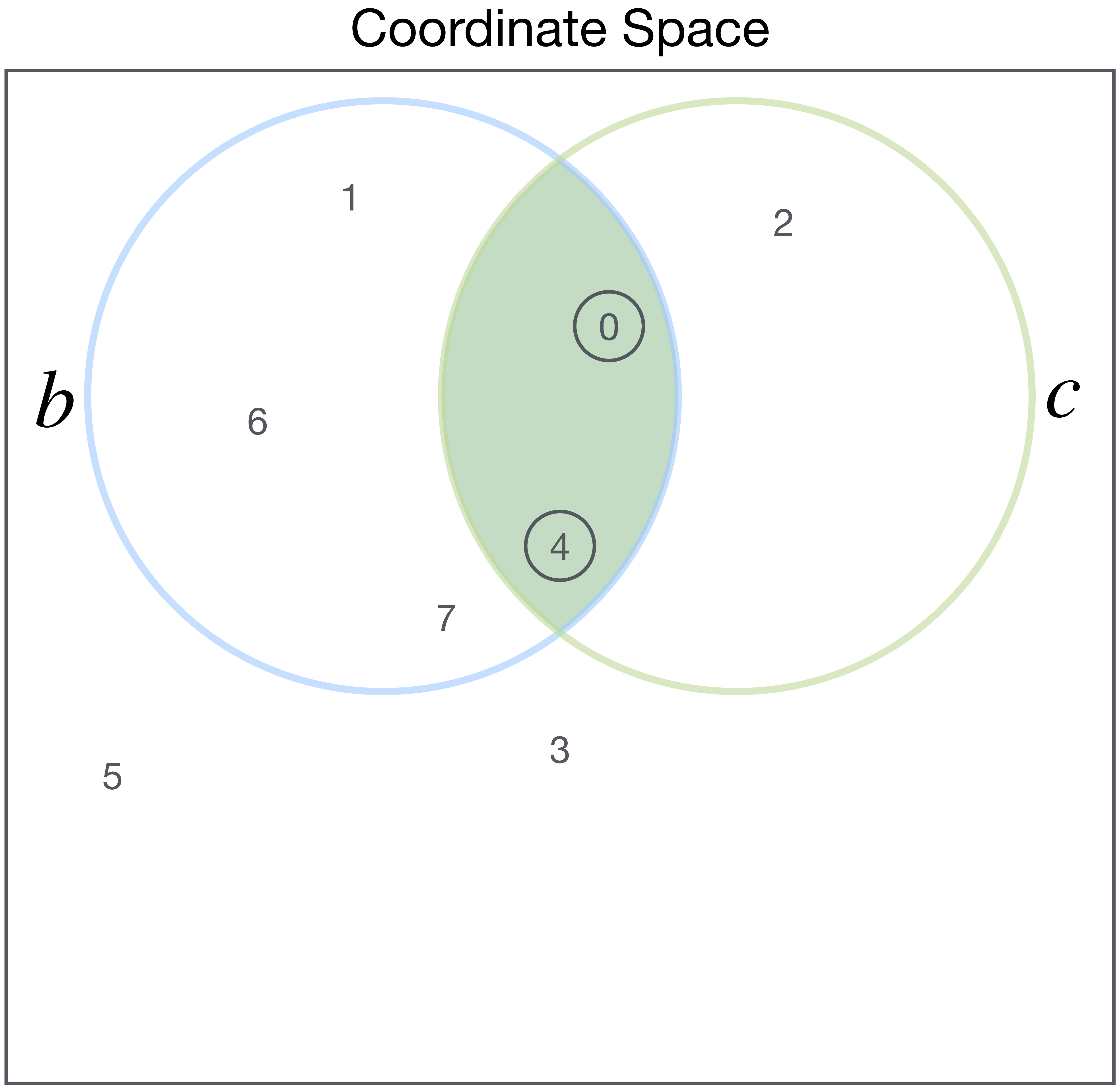


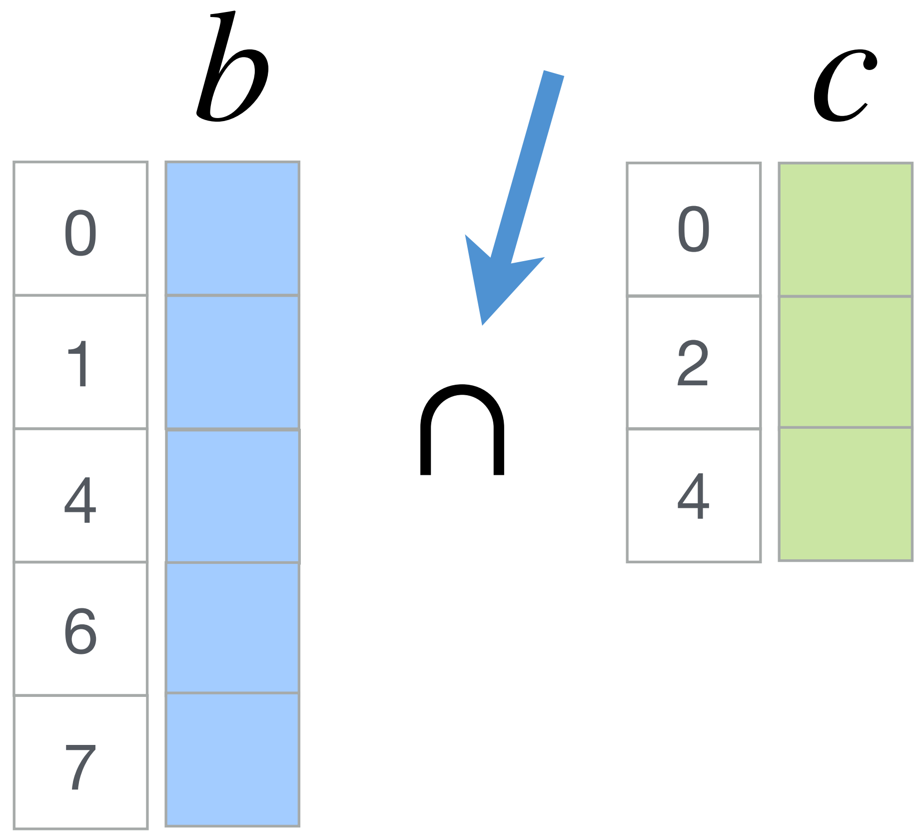
Data structure coiteration



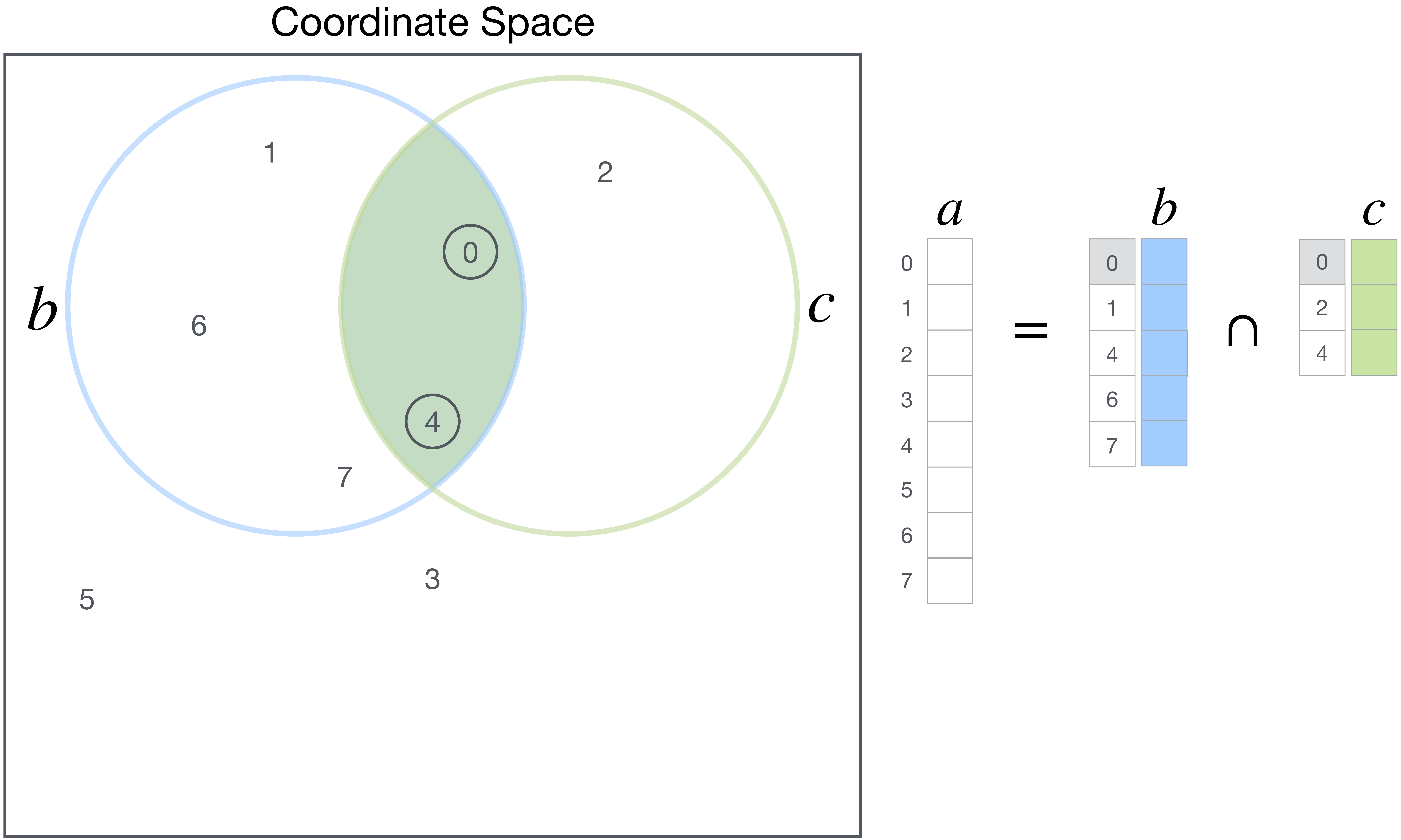


Data structure coiteration



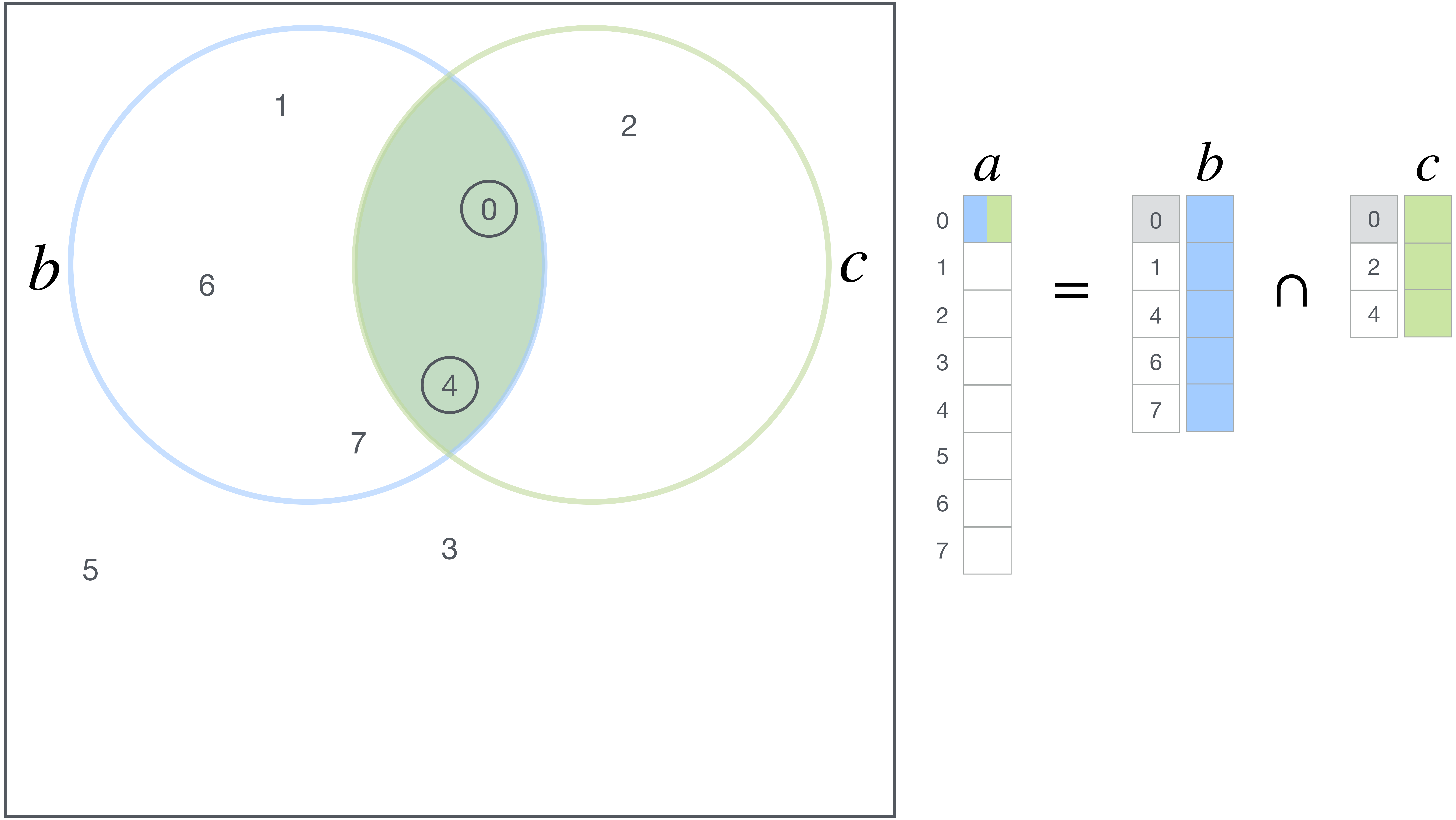


Data structure coiteration

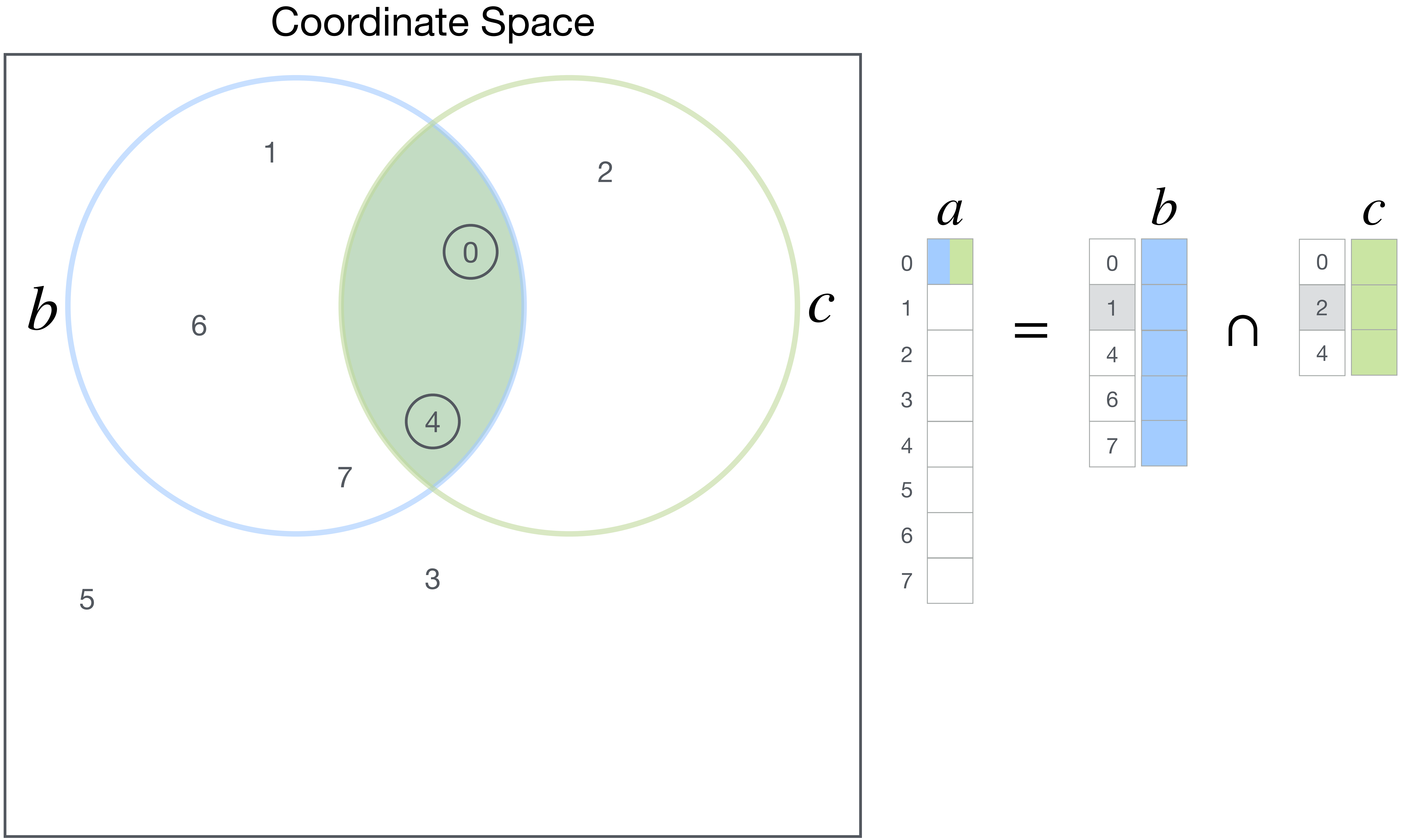


Data structure coiteration

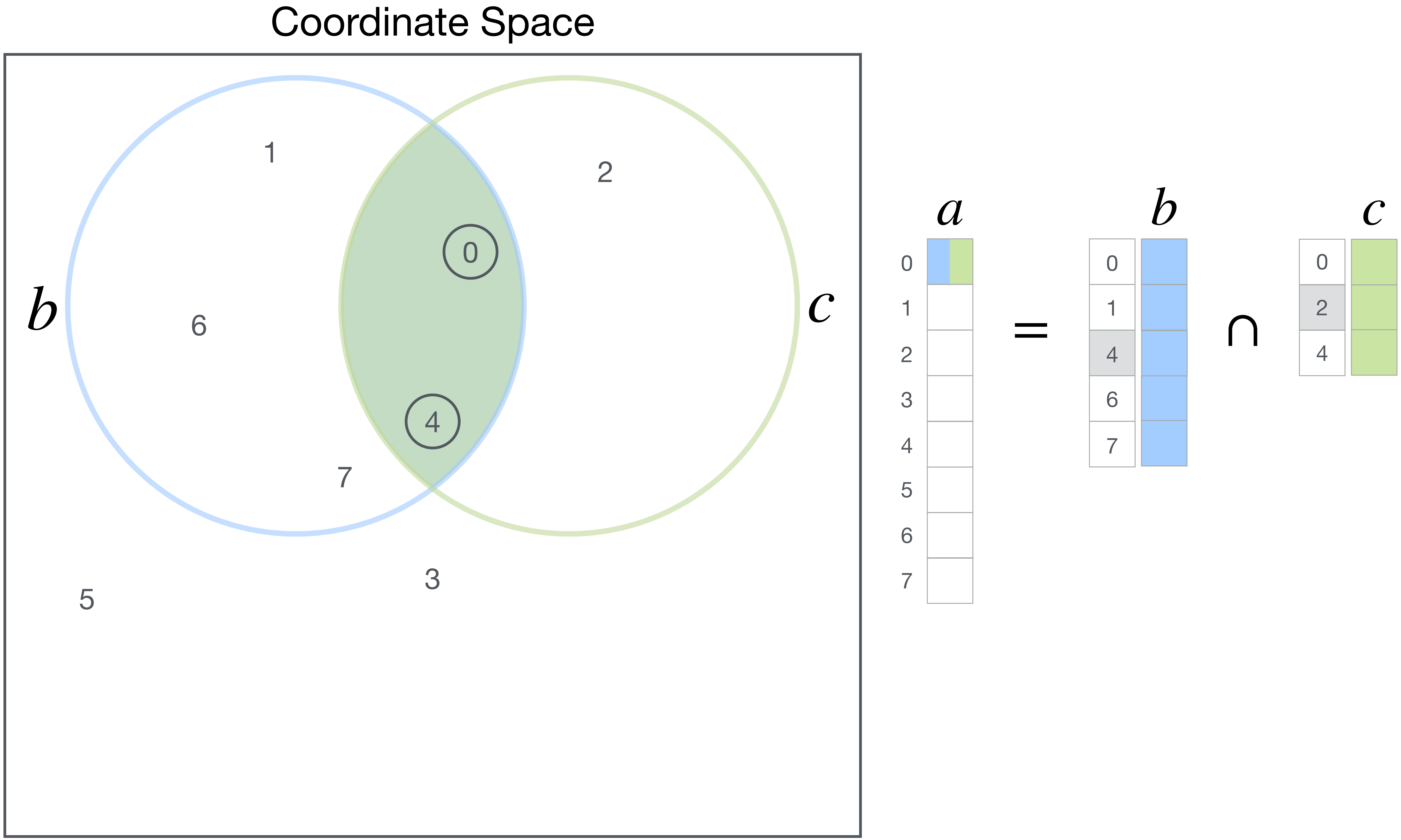
Coordinate Space



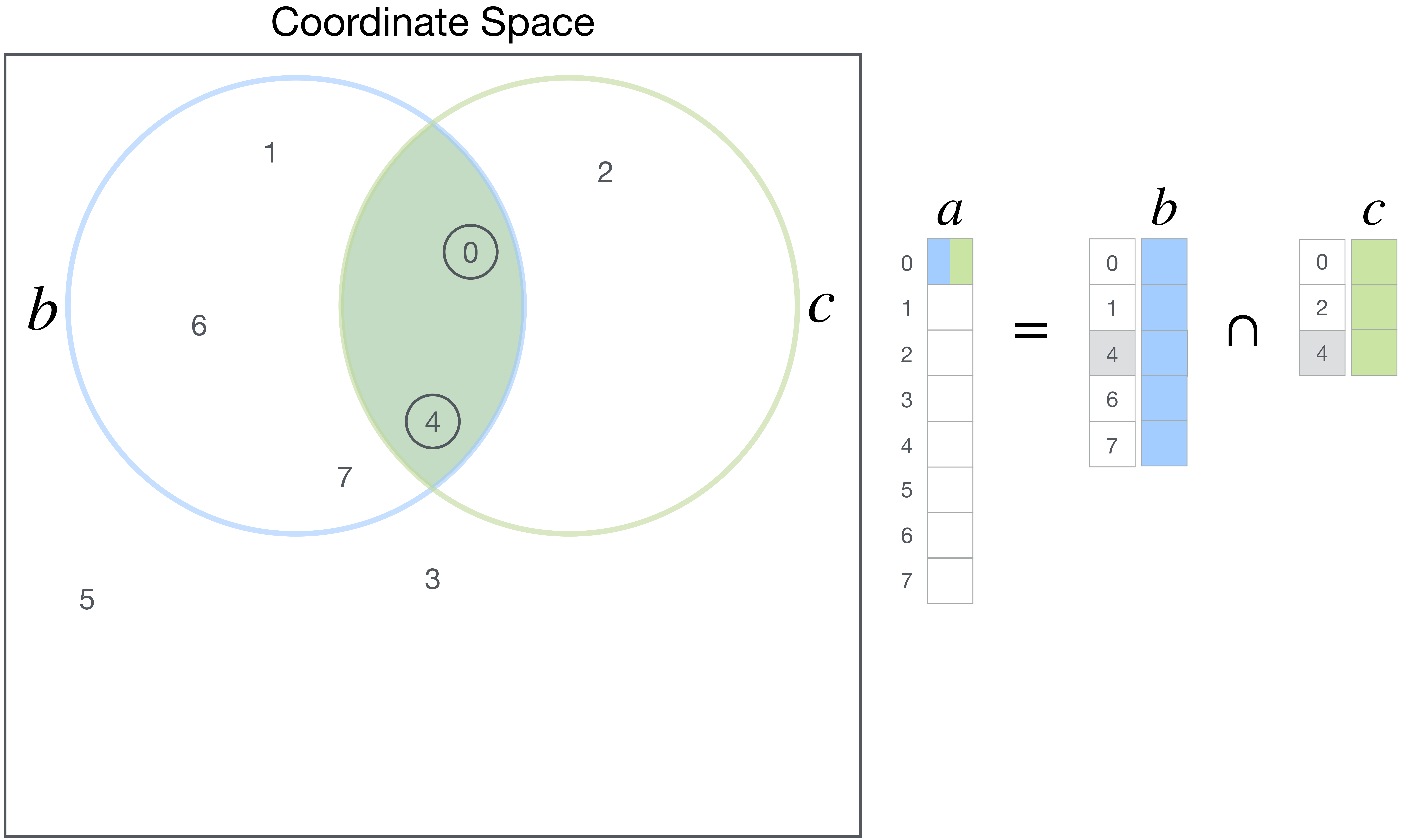
Data structure coiteration



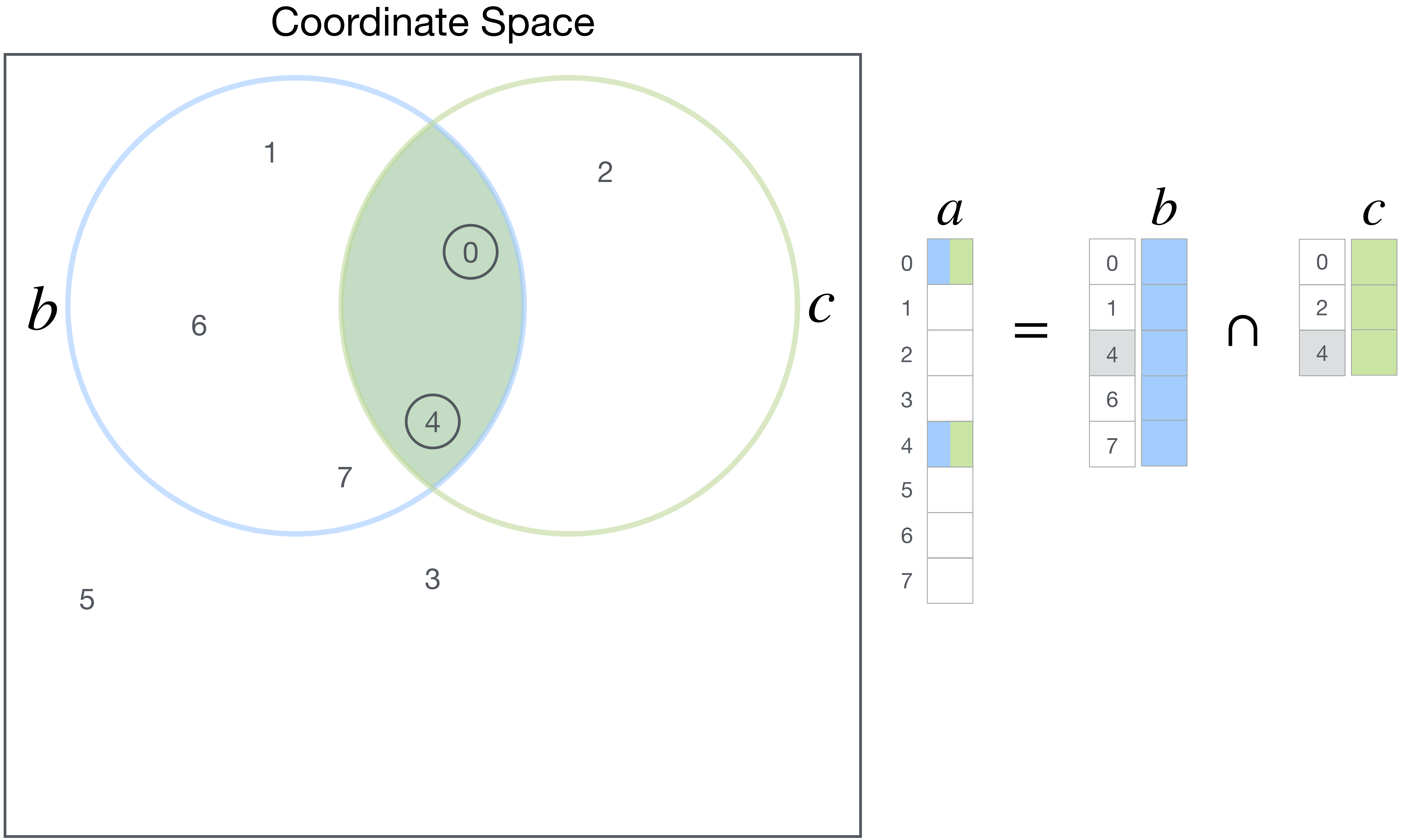
Data structure coiteration



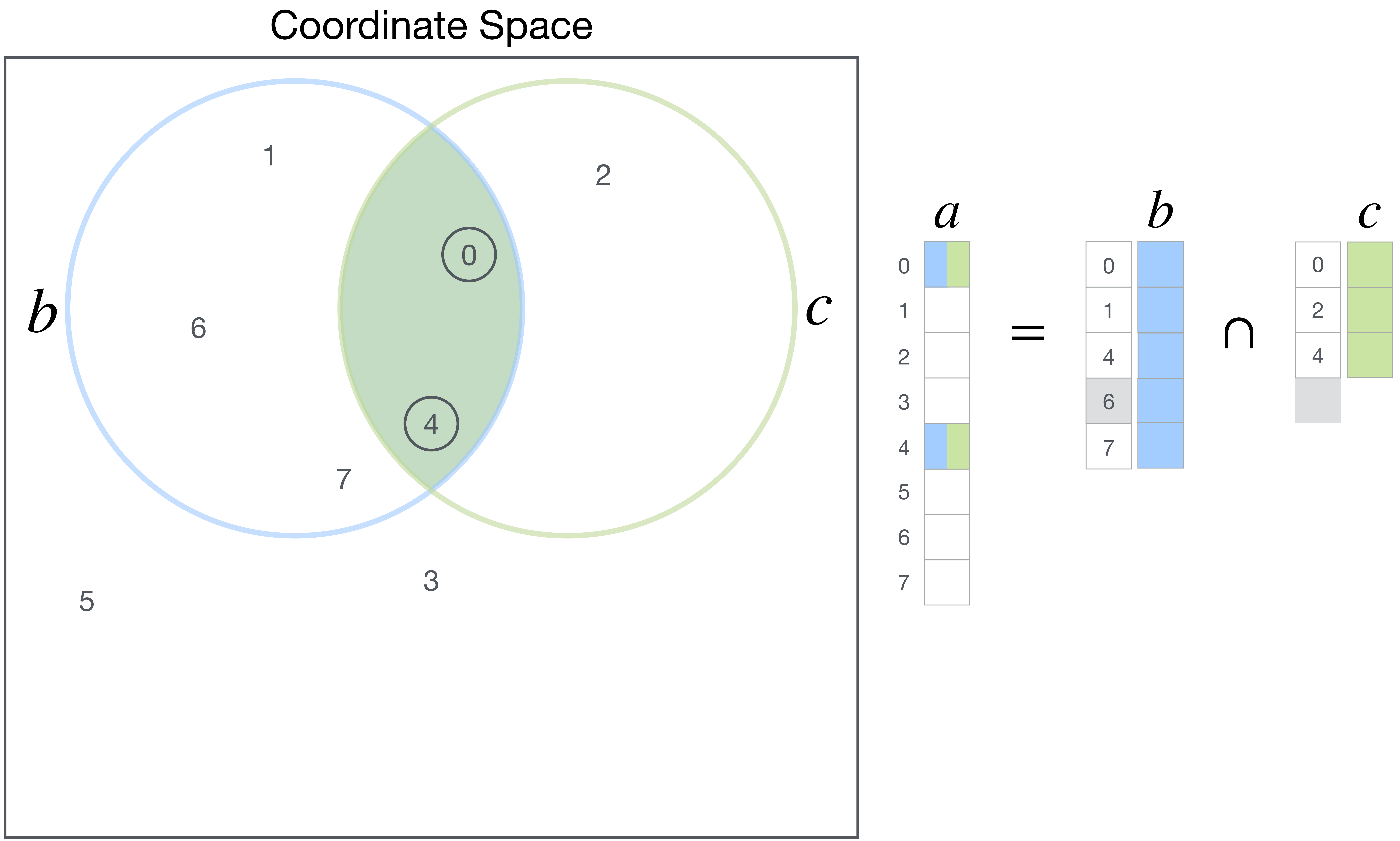
Data structure coiteration



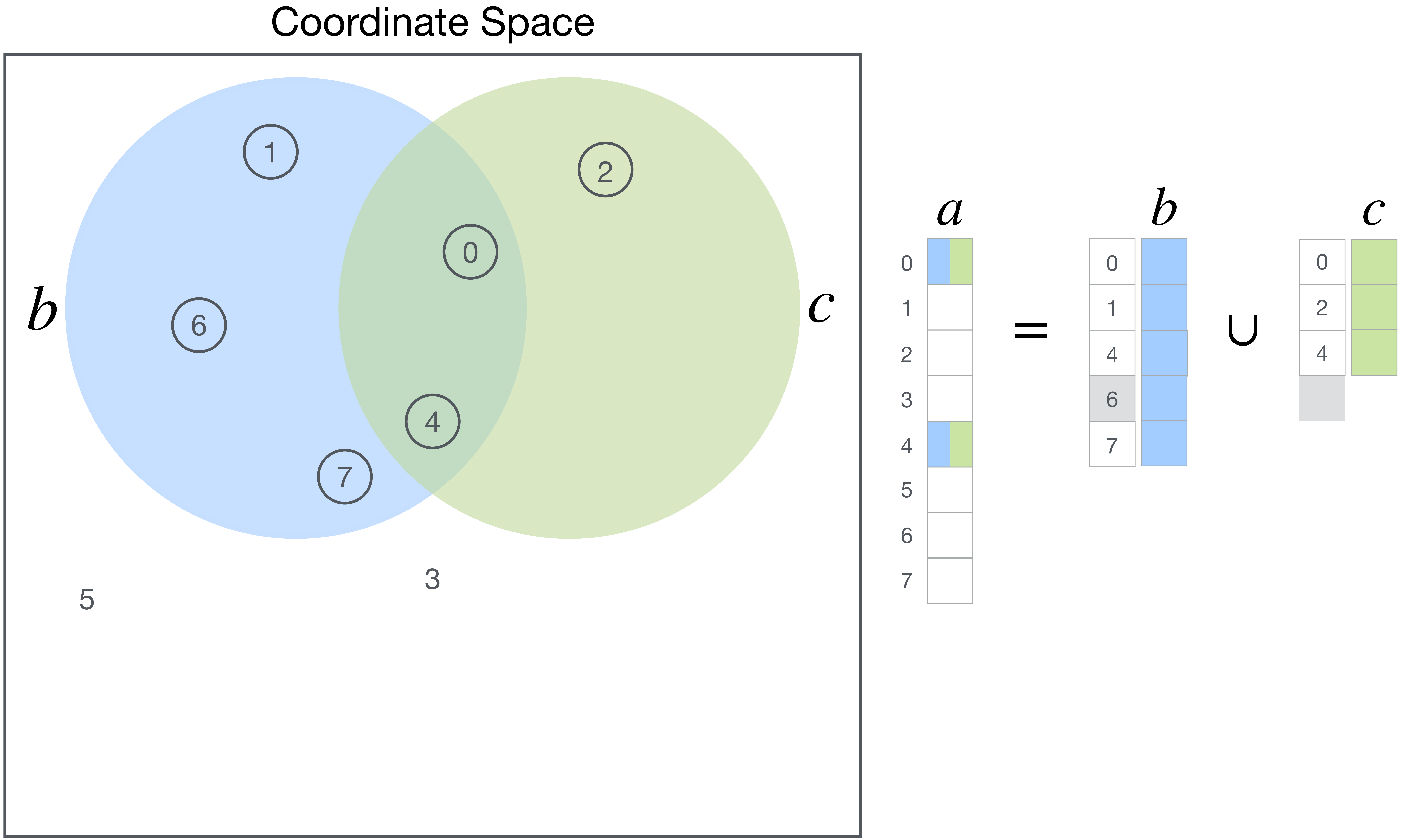
Data structure coiteration



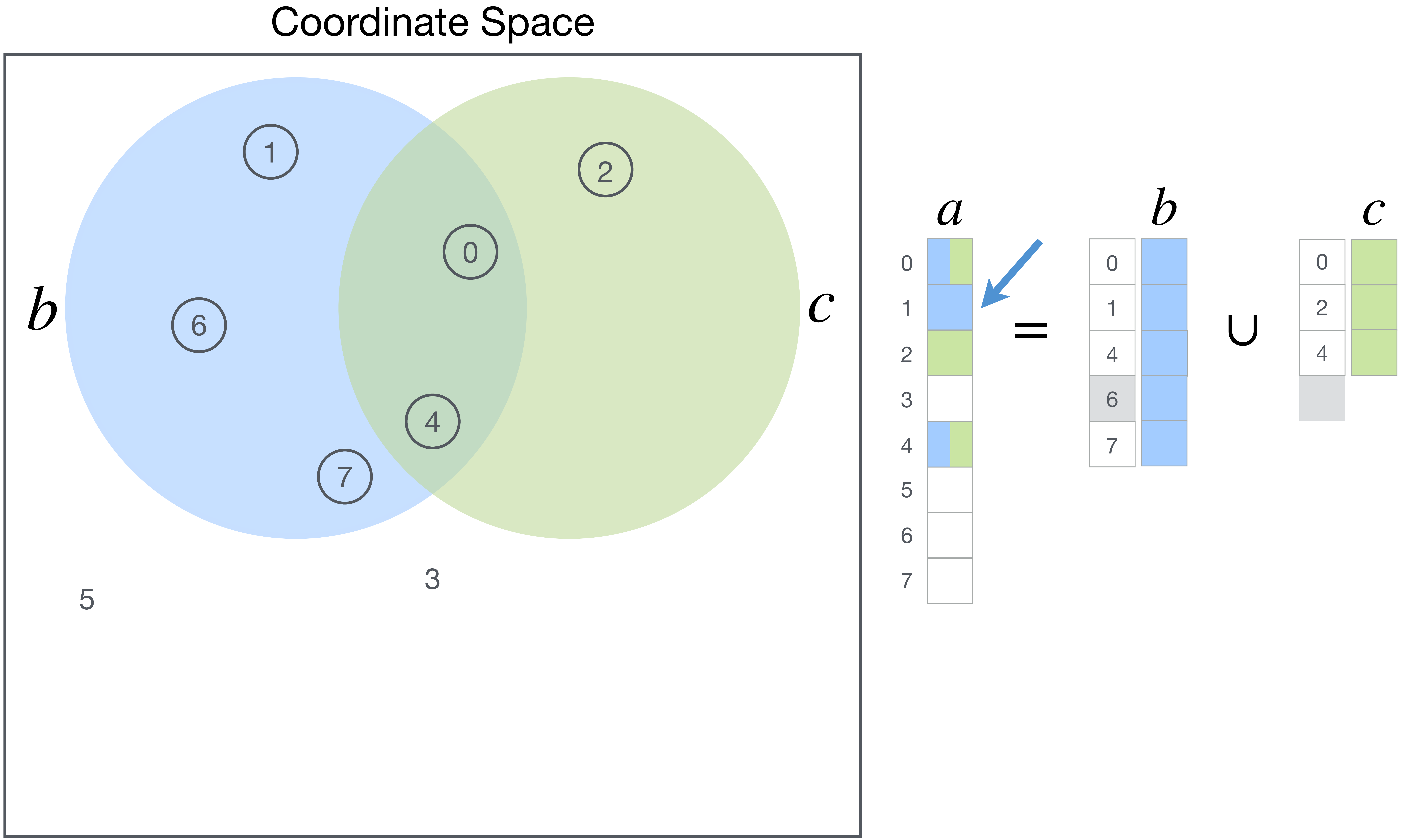
Data structure coiteration



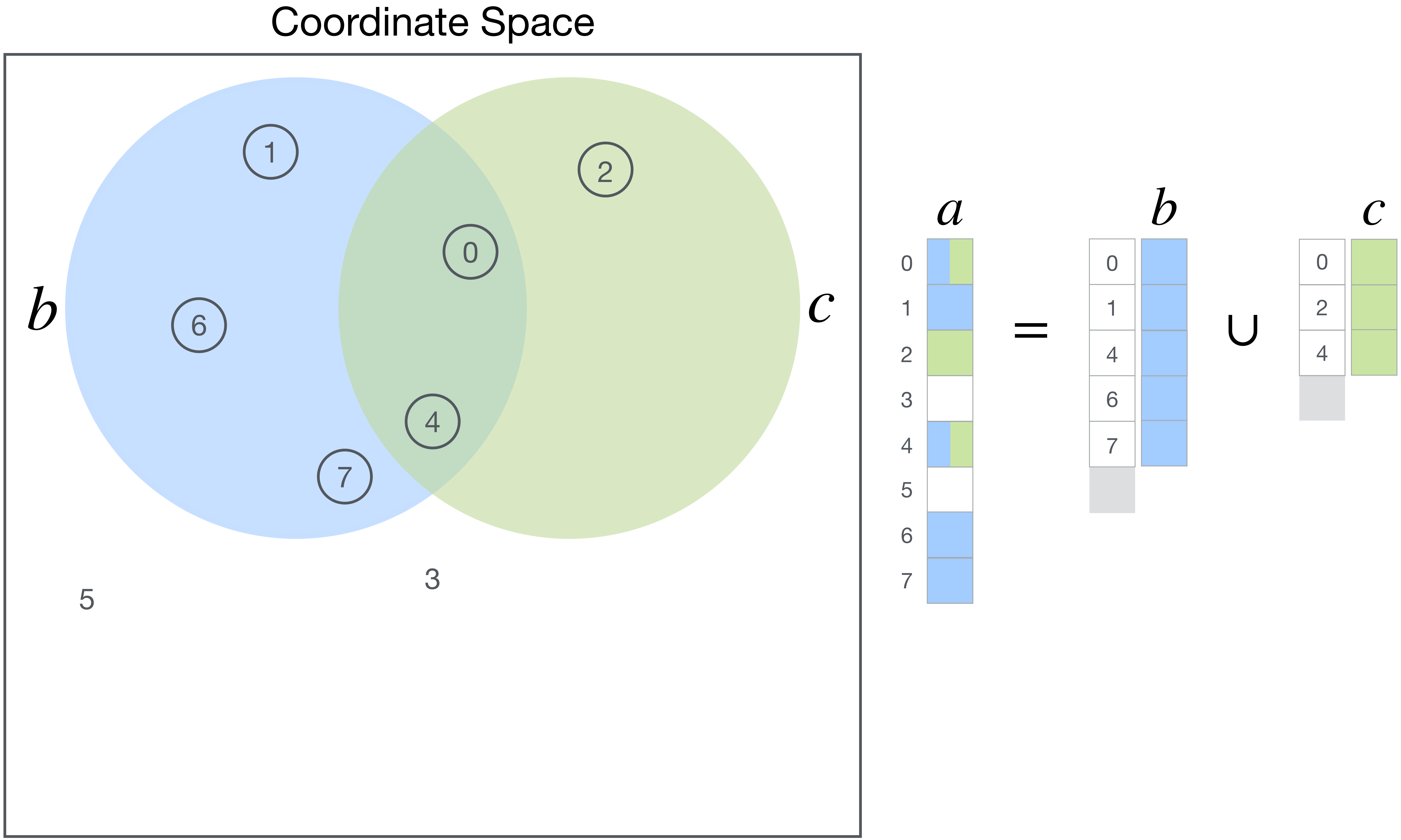
Data structure coiteration



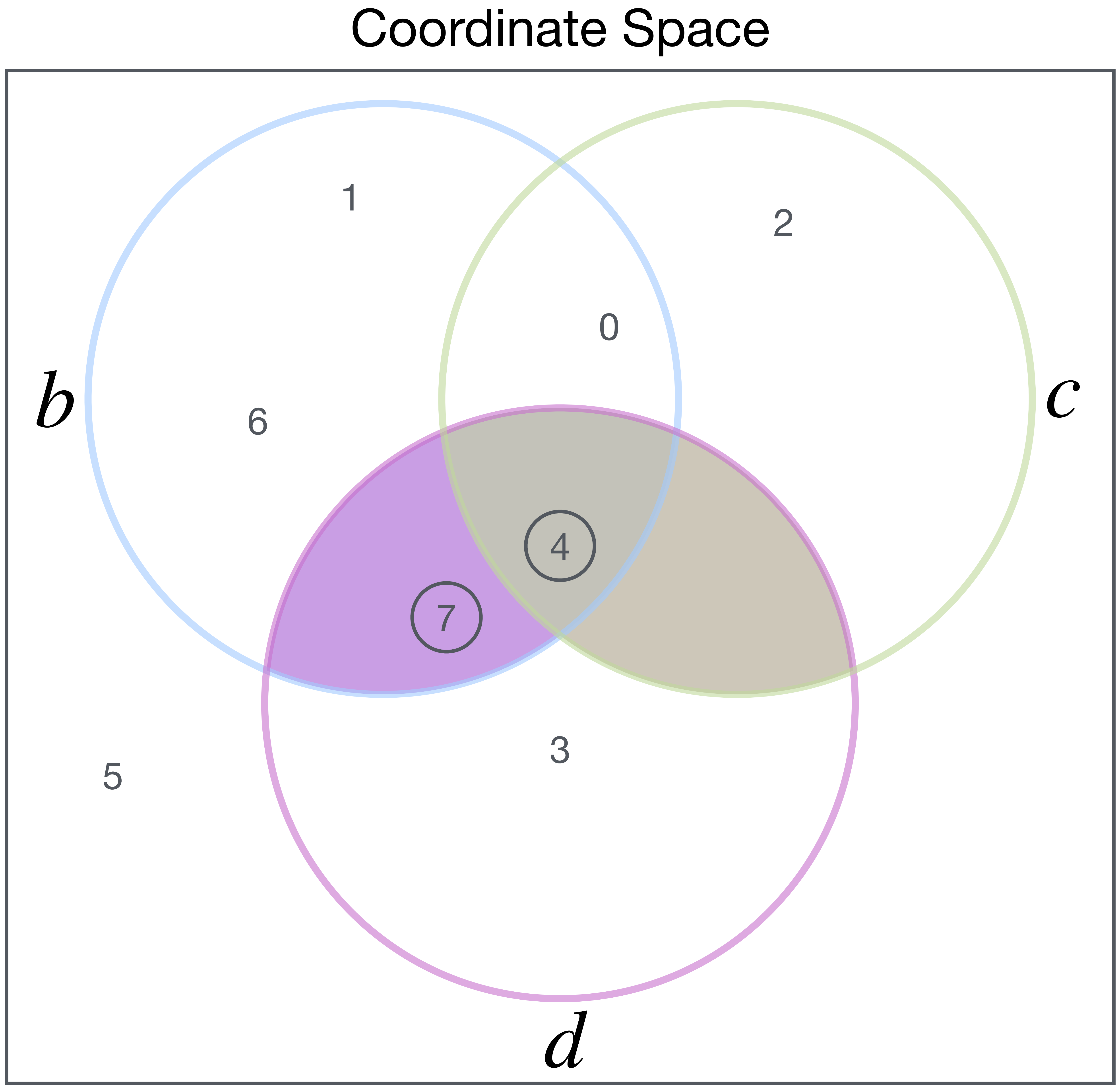
Data structure coiteration

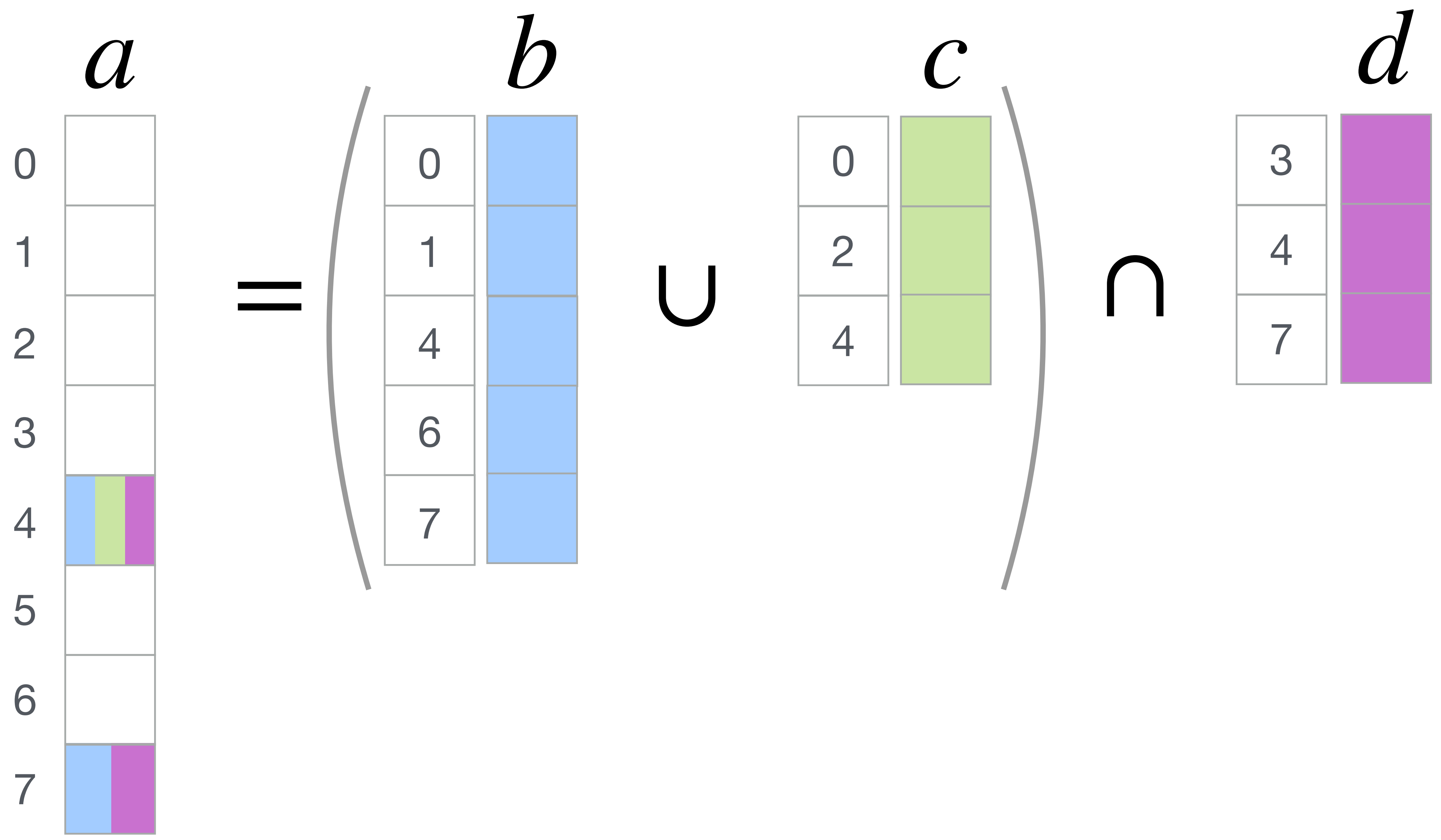


Data structure coiteration

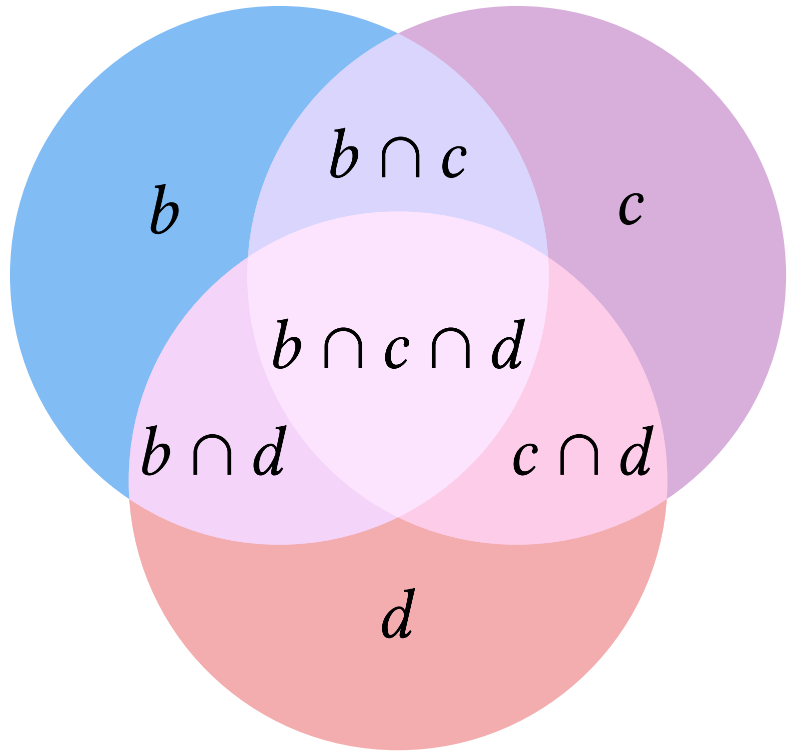


Data structure coiteration

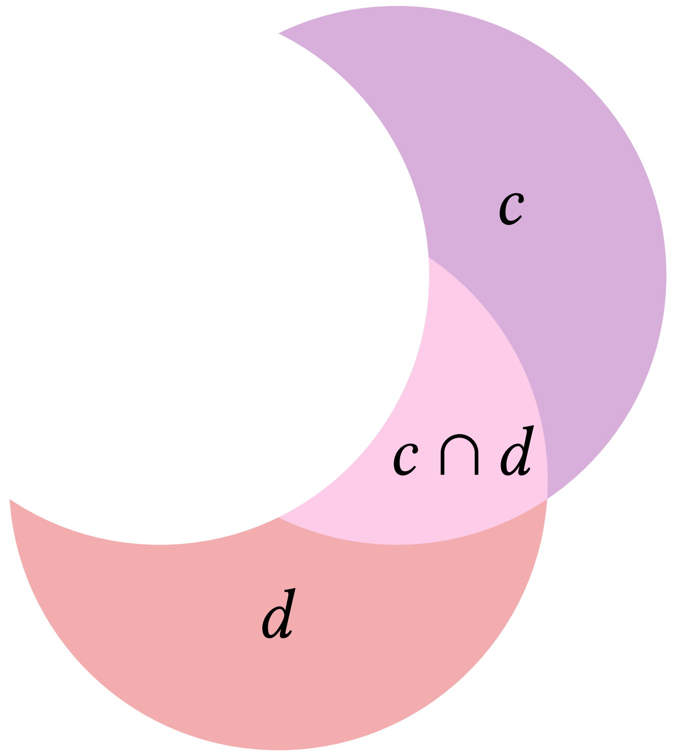




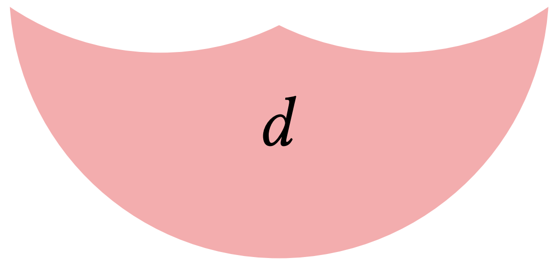
Coiteration successively eliminates data structures



Coiterate over regions with b, c, and d



b runs out of values

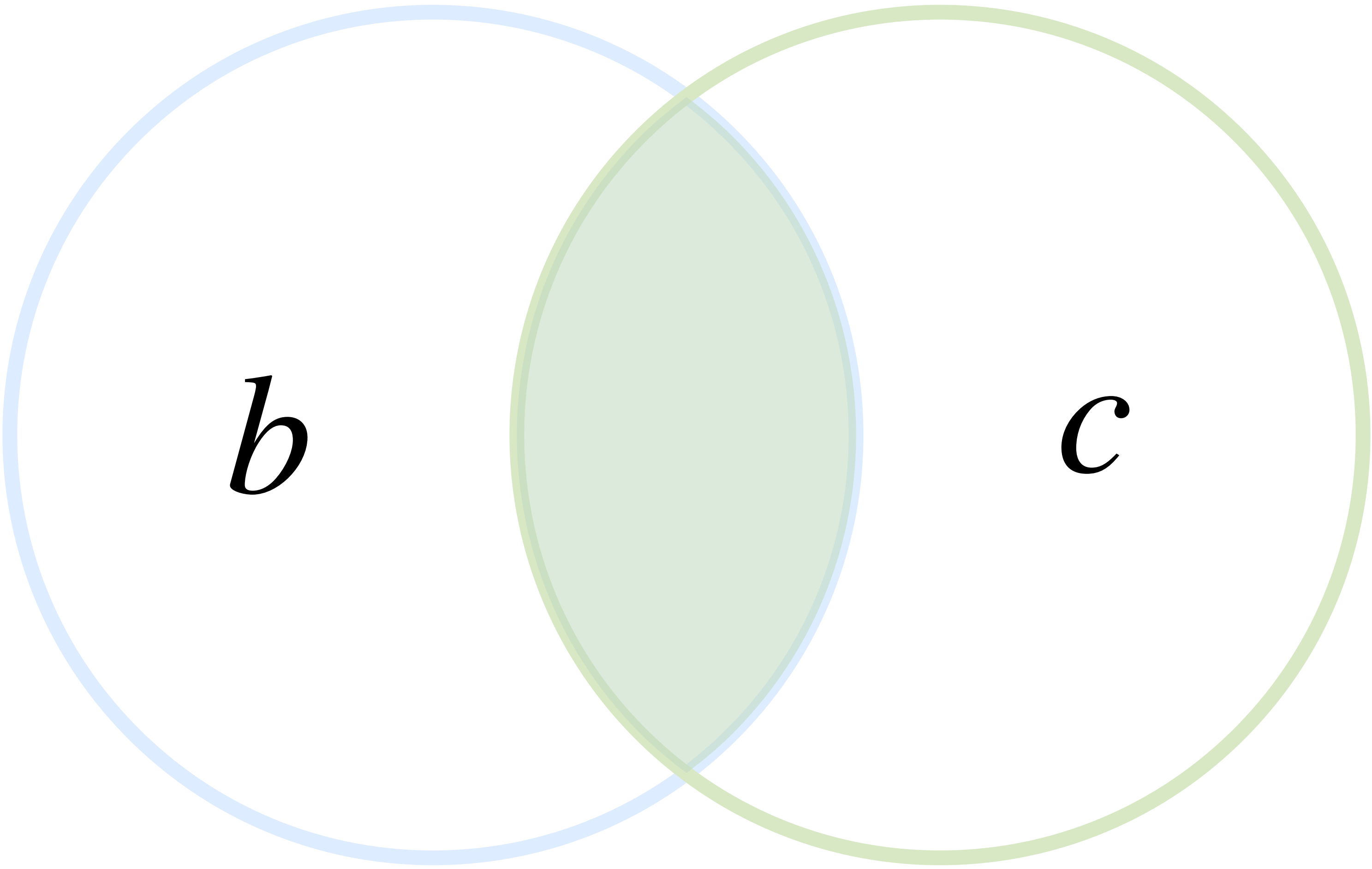


c runs out of values

Iteration lattice for multiplications

$$a_i = b_i c_i$$

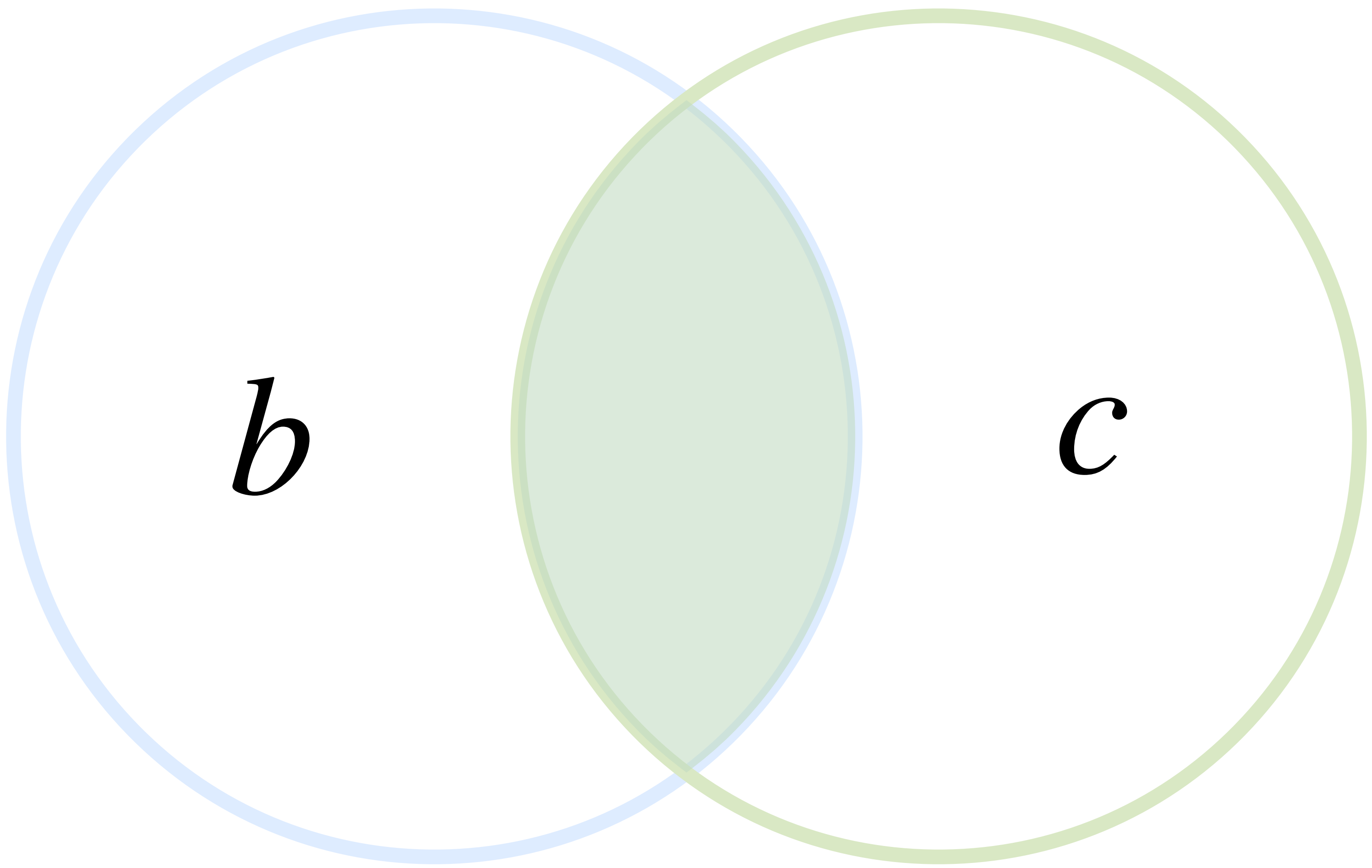
Multiplication requires intersection

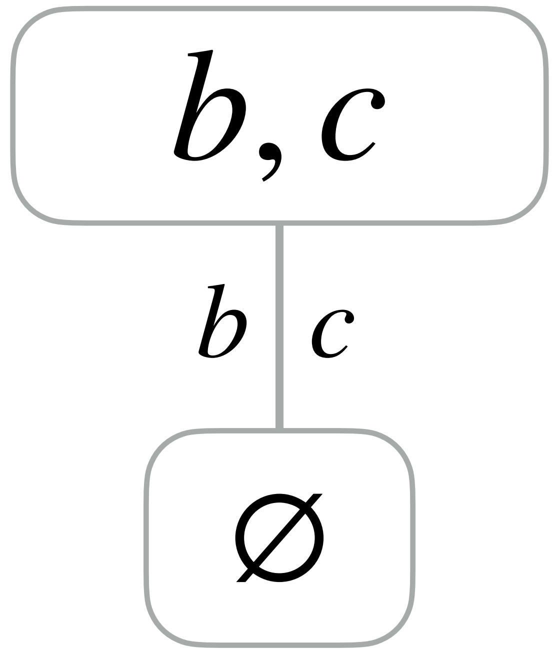


$$b \cap c$$

Iteration lattice for multiplications

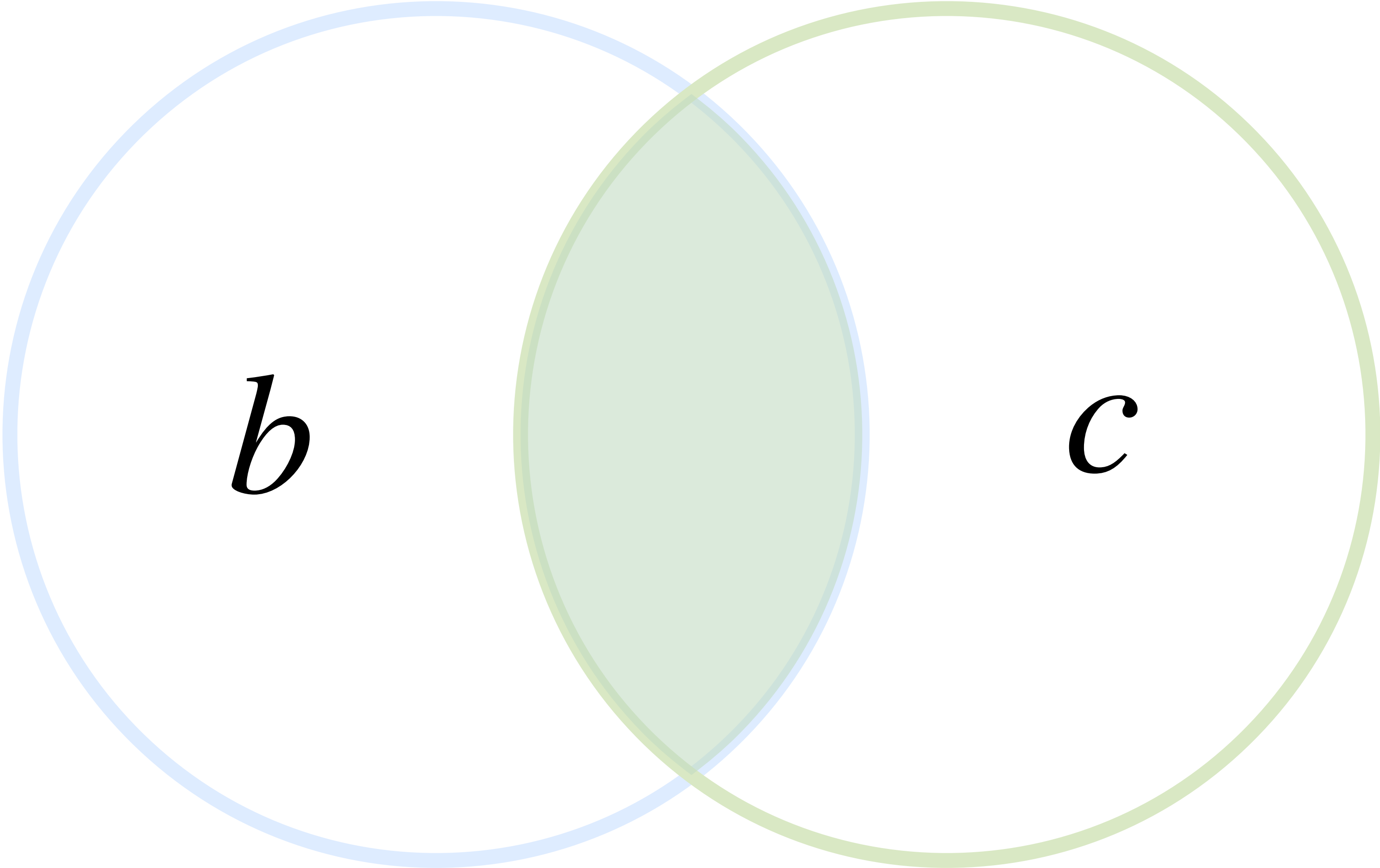
$$a_i = b_i c_i$$

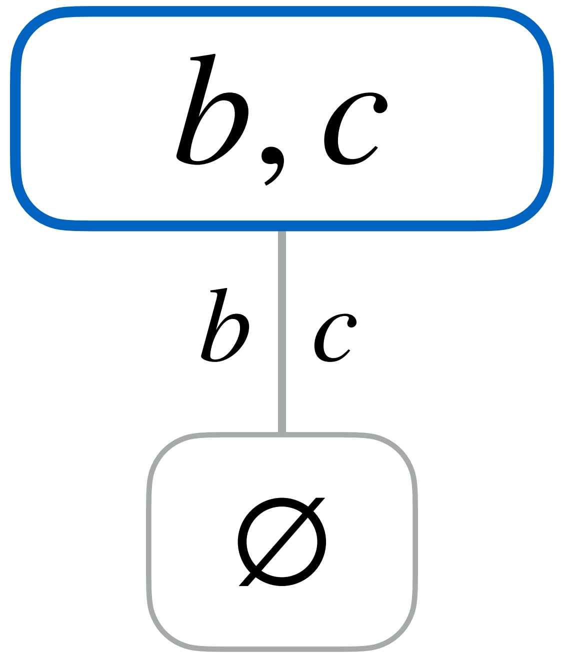




Iteration lattice for multiplications

$$a_i = b_i c_i$$

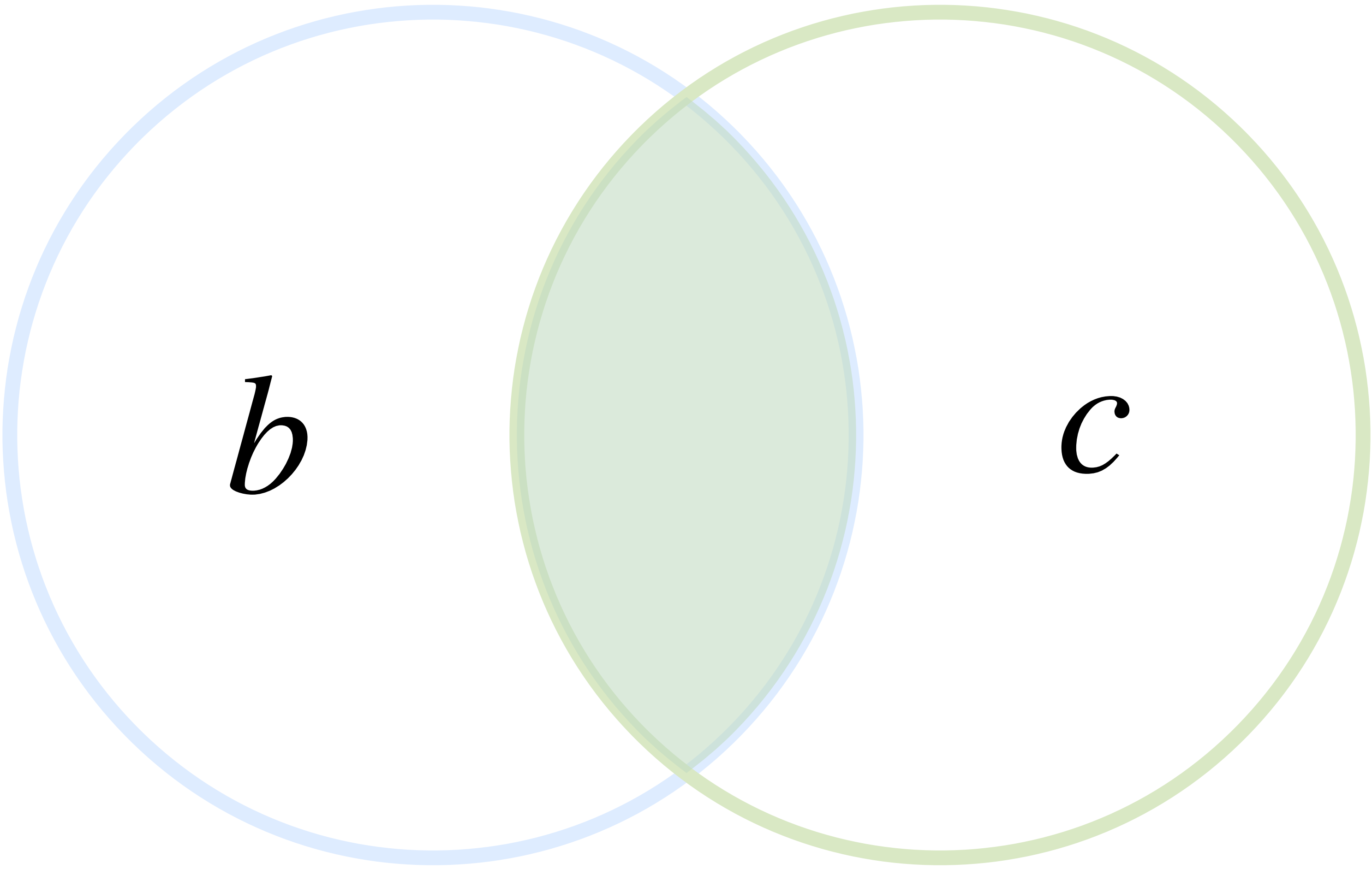


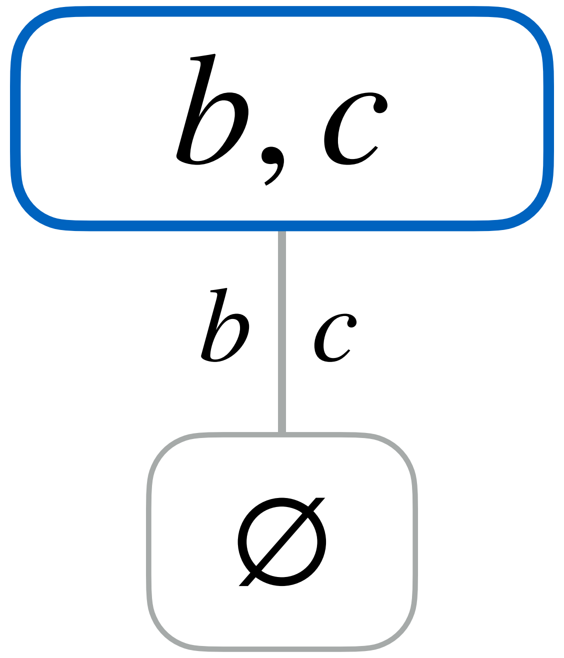



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);


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

Iteration lattice for multiplications

$$a_i = b_i c_i$$

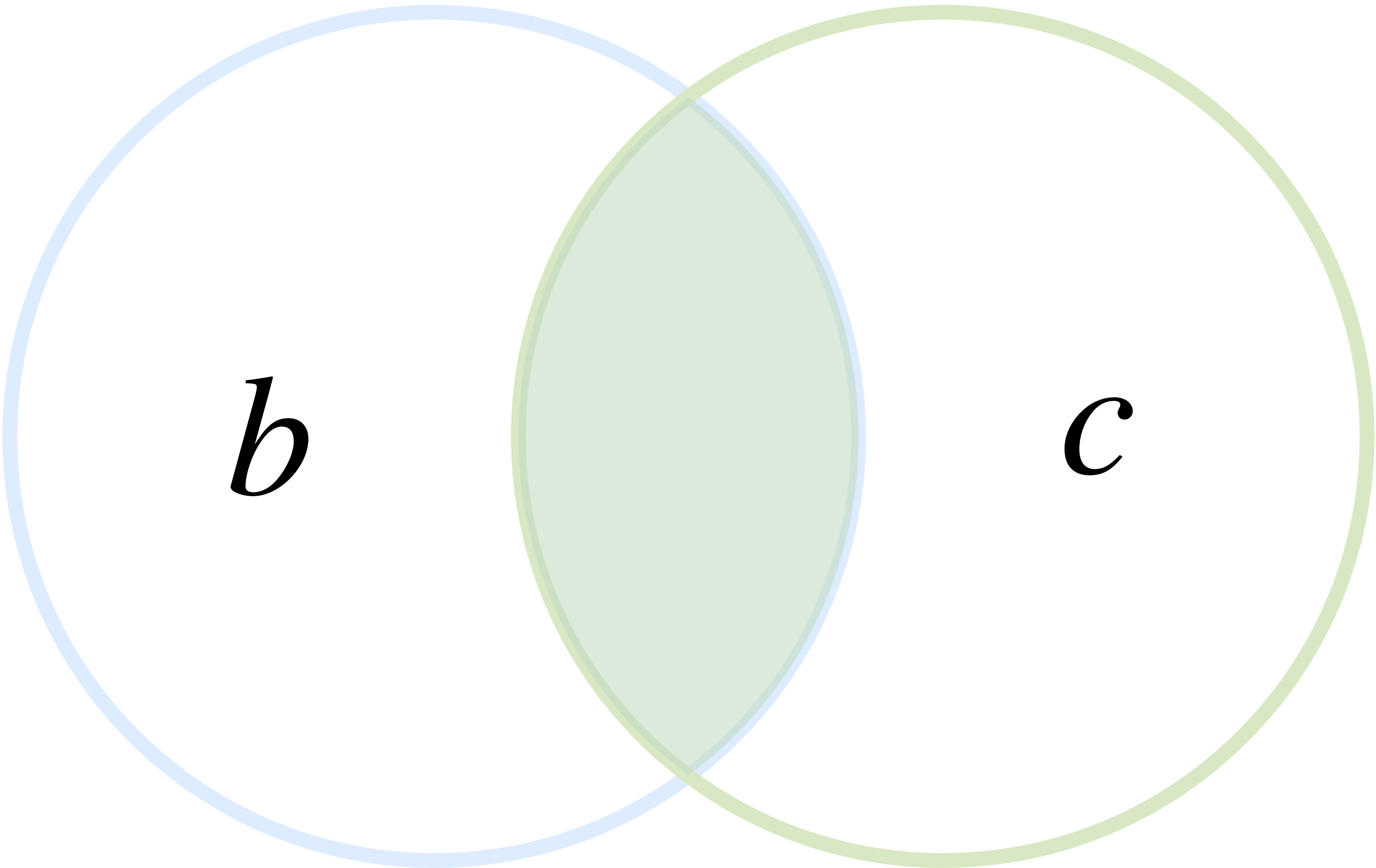


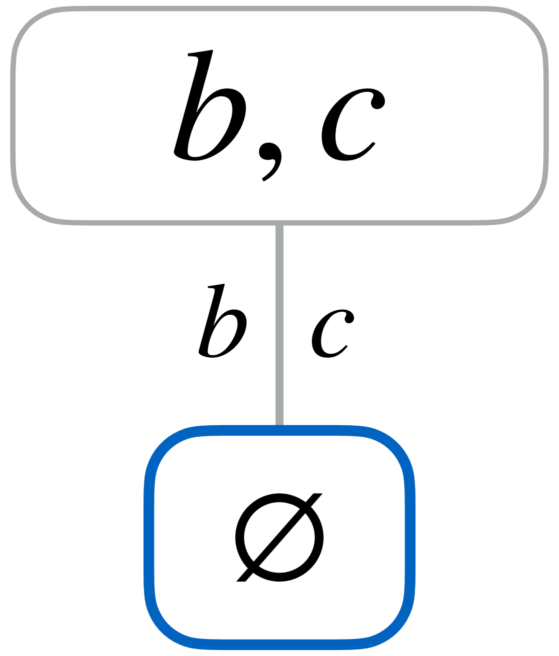



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] * c[pc1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}
```

Iteration lattice for multiplications

$$a_i = b_i c_i$$





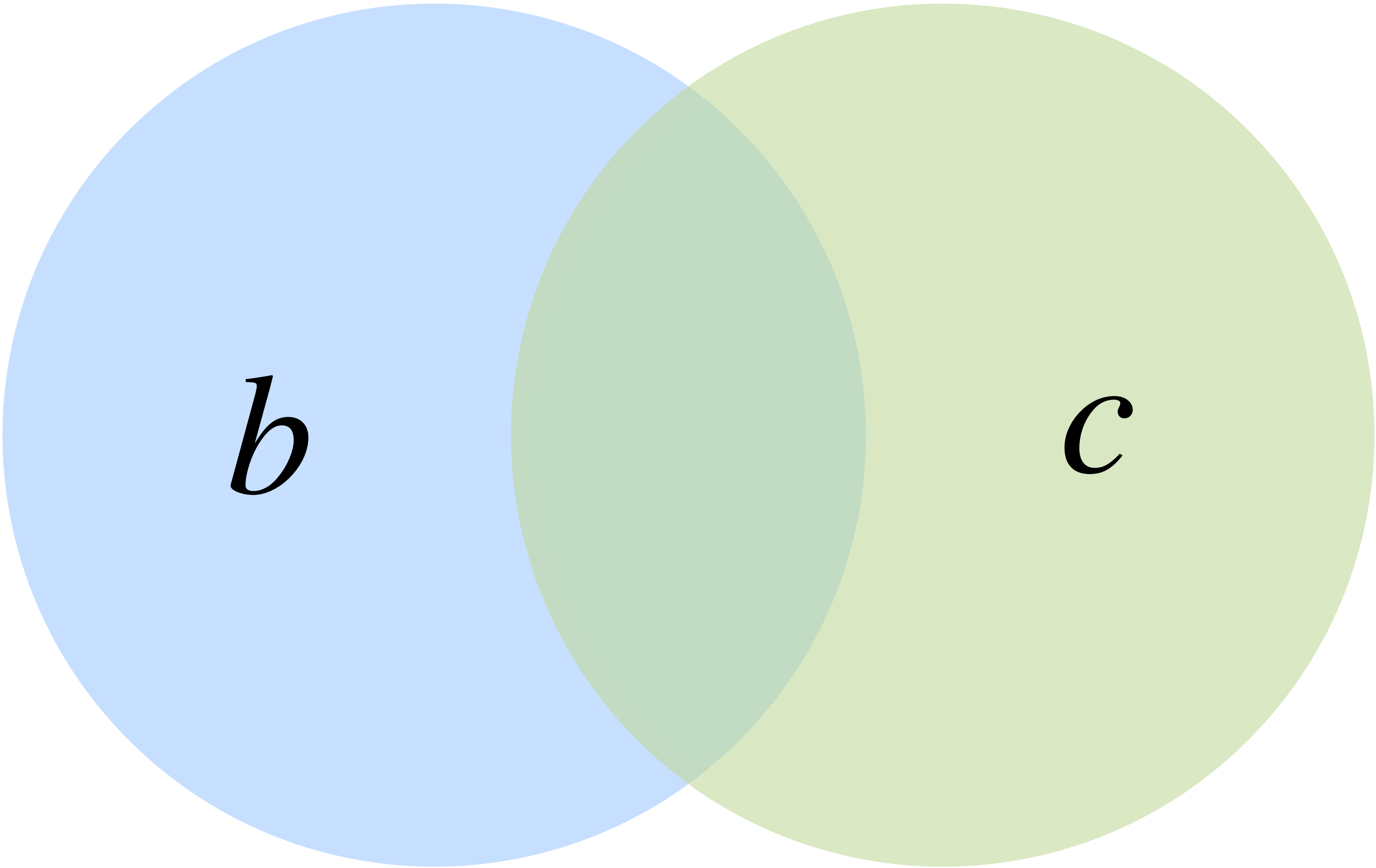

```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] * c[pc1];
  }
  if (ib == i) pb1++;
  if (ic == i) pc1++;
}
```

Iteration lattice for additions

$$a_i = b_i + c_i$$

Iteration lattice for additions

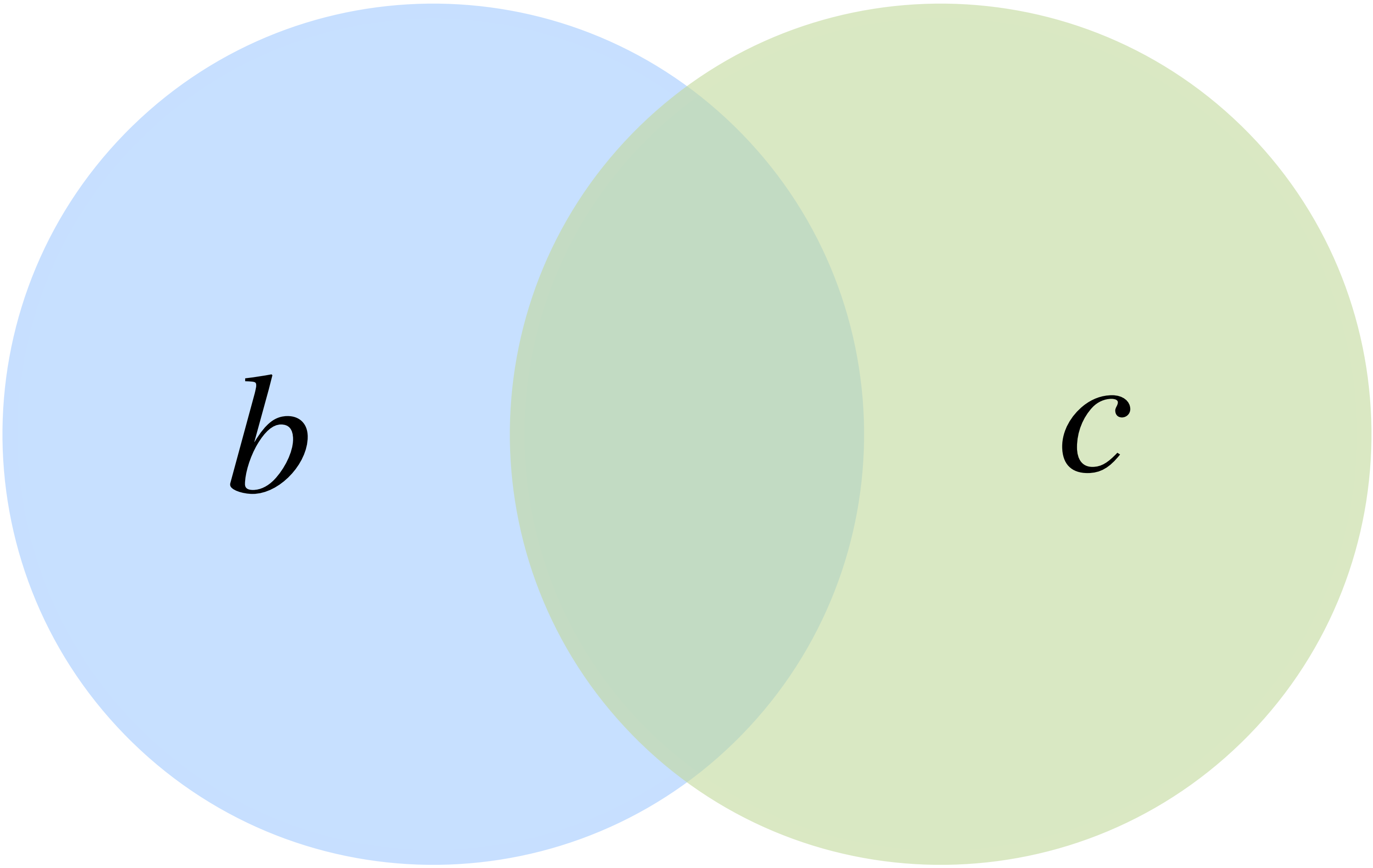
$$a_i = b_i + c_i$$



$$b \cup c$$

Iteration lattice for additions

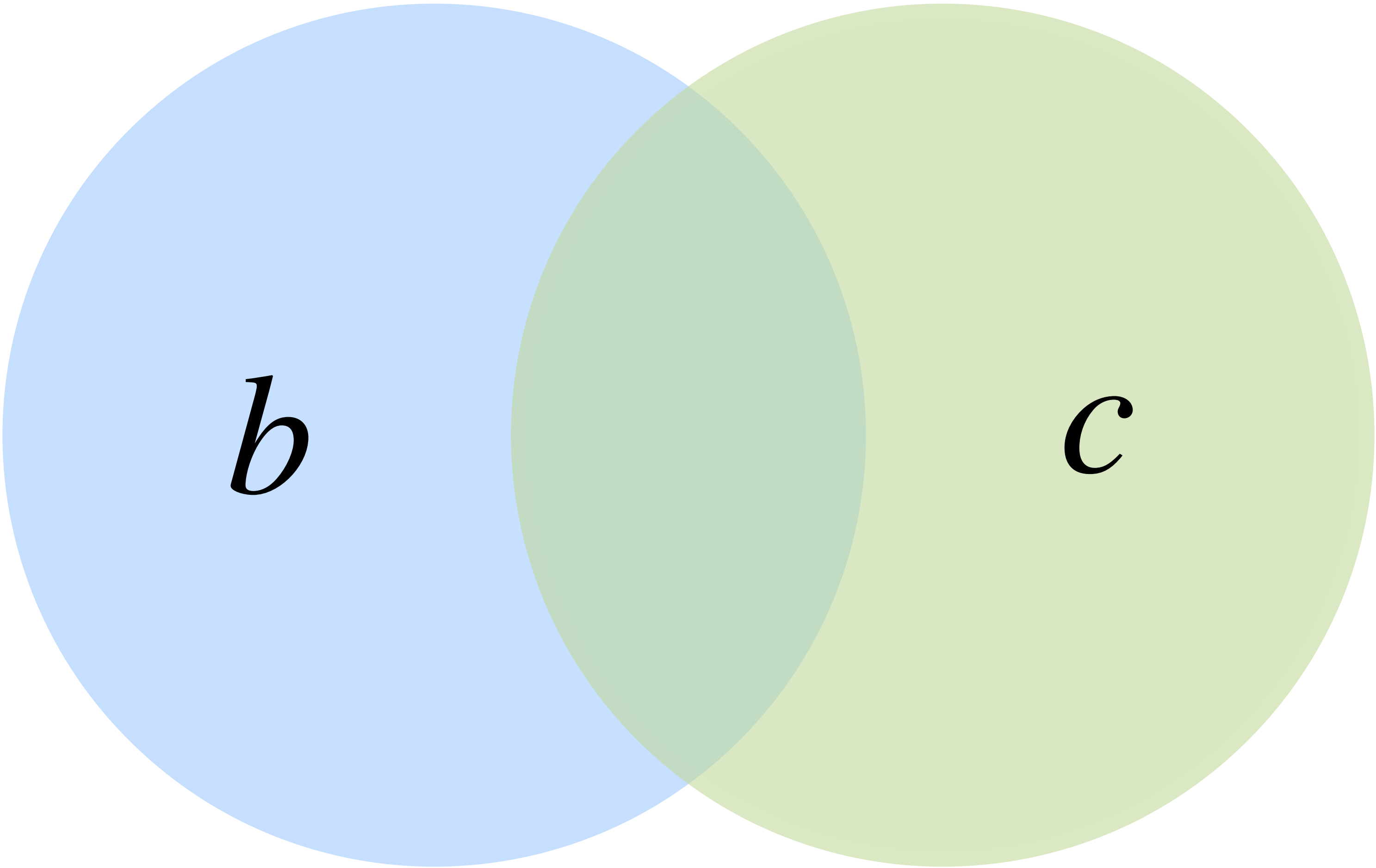
$$a_i = b_i + c_i$$

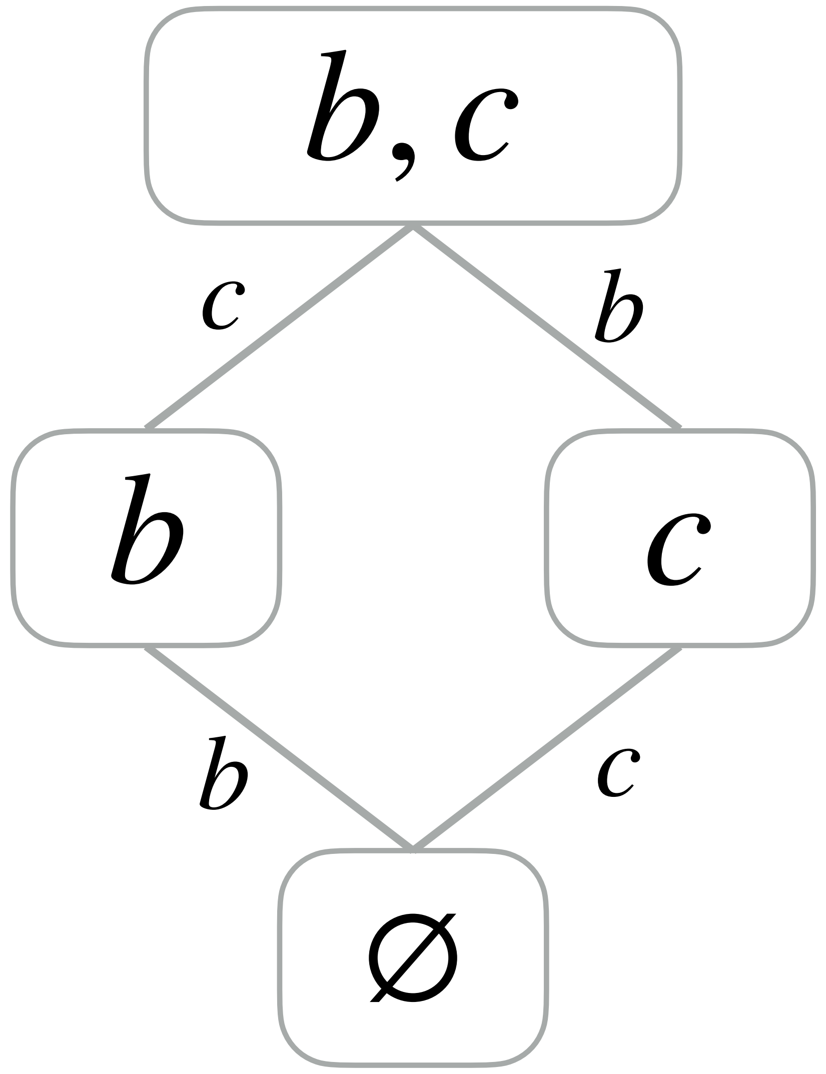


$$b \cup c = (b \cap c) \cup b \cup c$$

Iteration lattice for additions

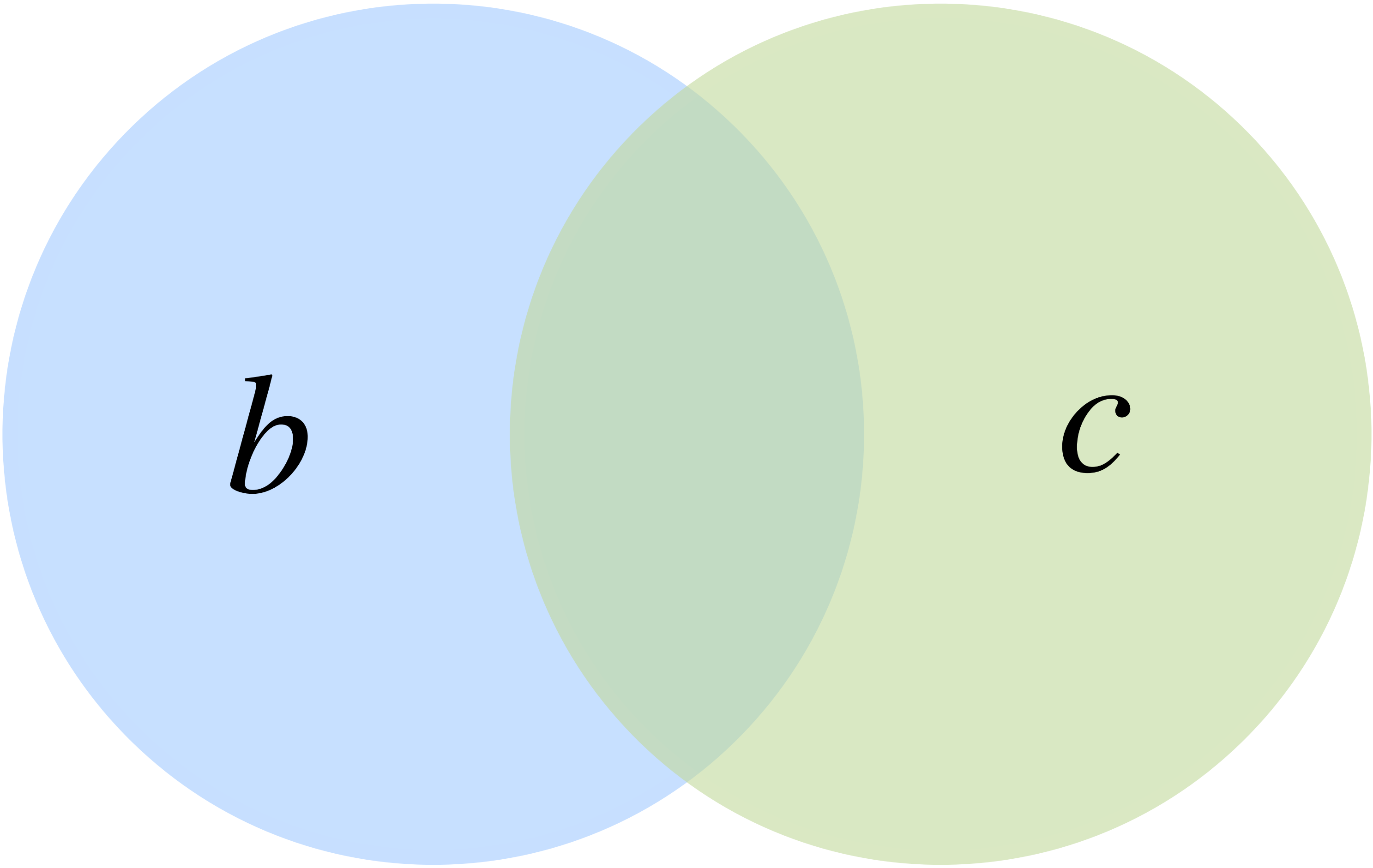
$$a_i = b_i + c_i$$

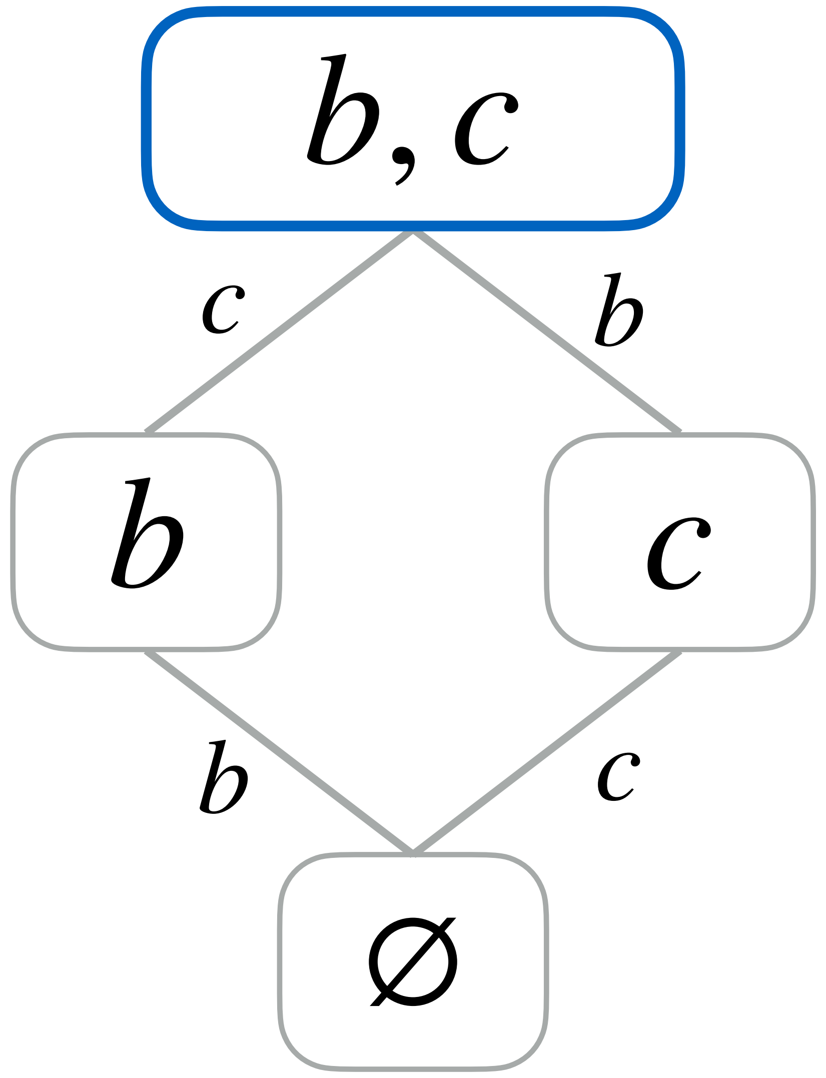




Iteration lattice for additions

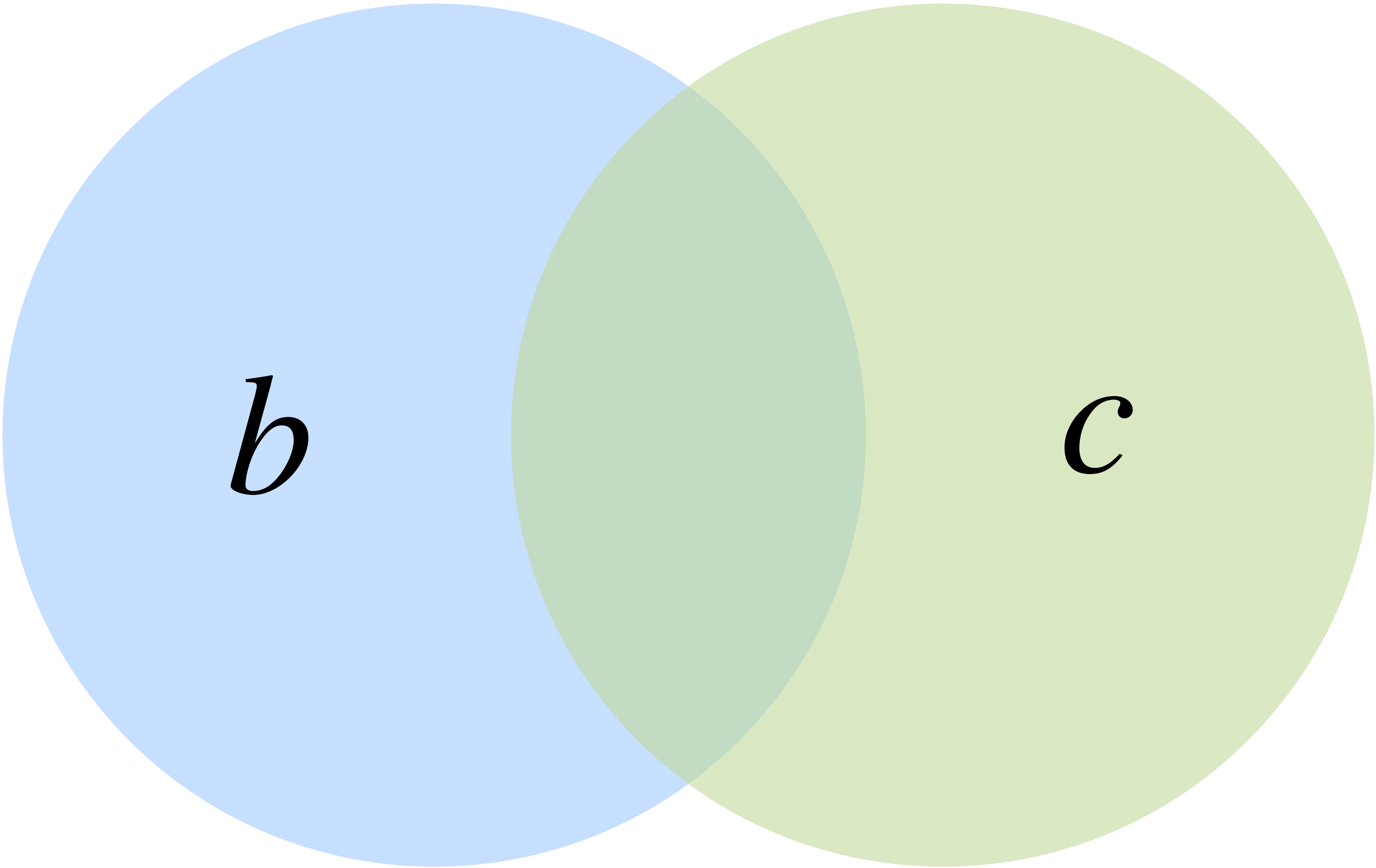
$$a_i = b_i + c_i$$

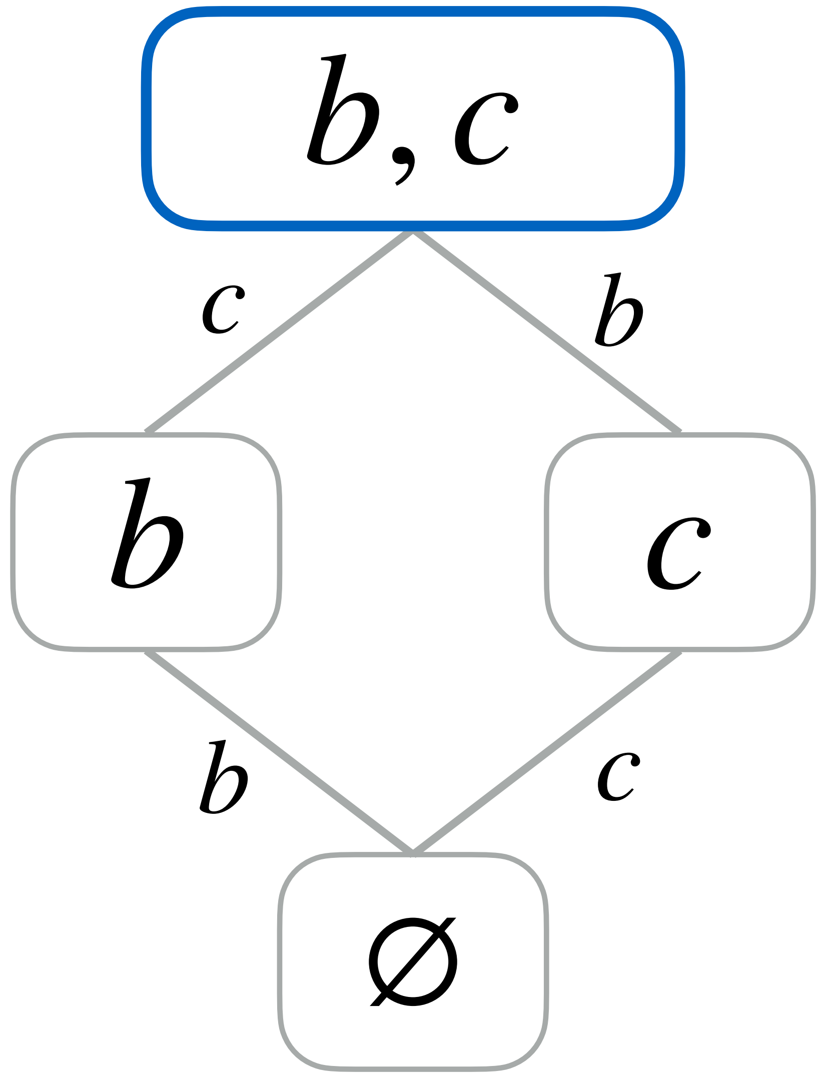




Iteration lattice for additions

$$a_i = b_i + c_i$$

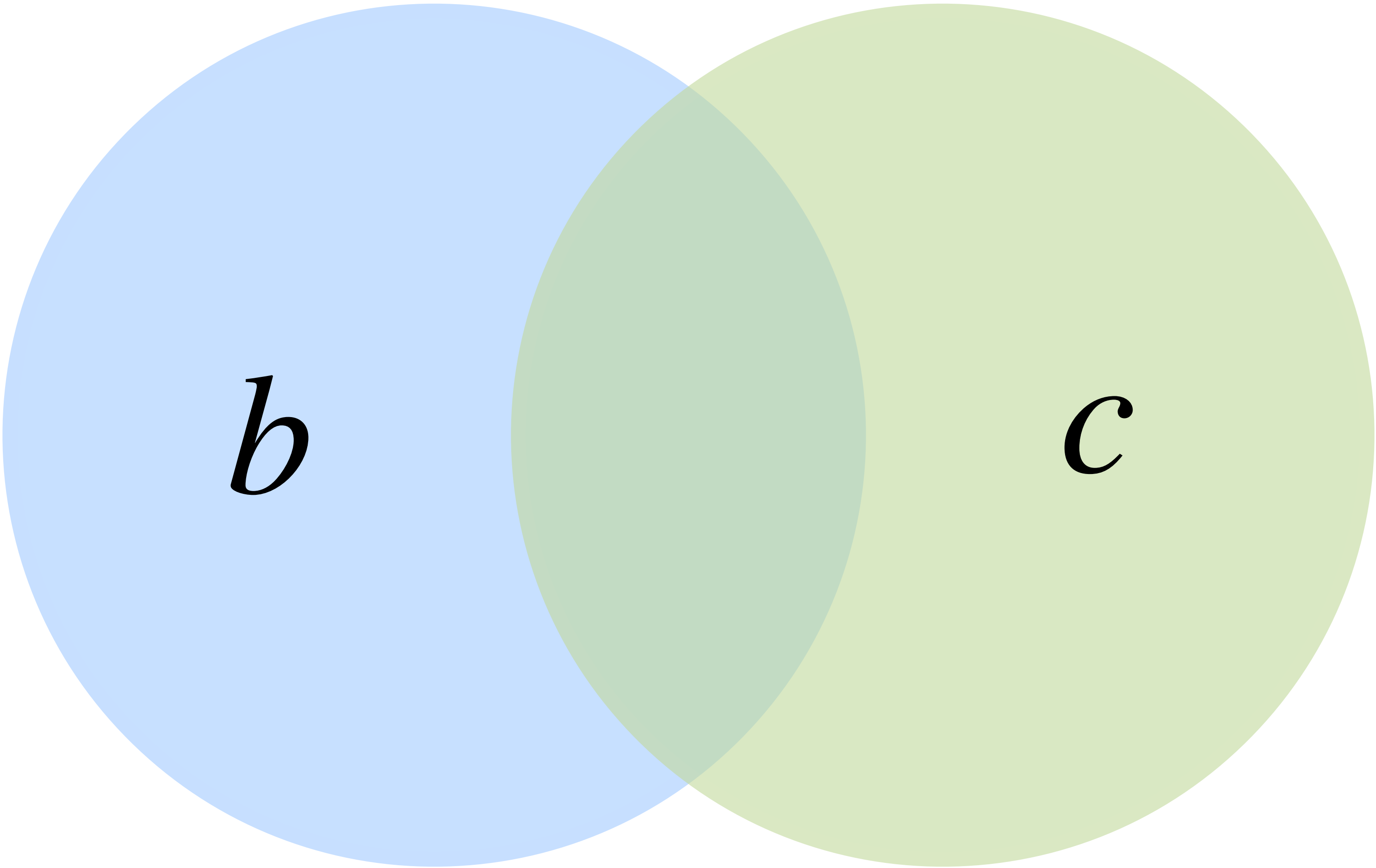


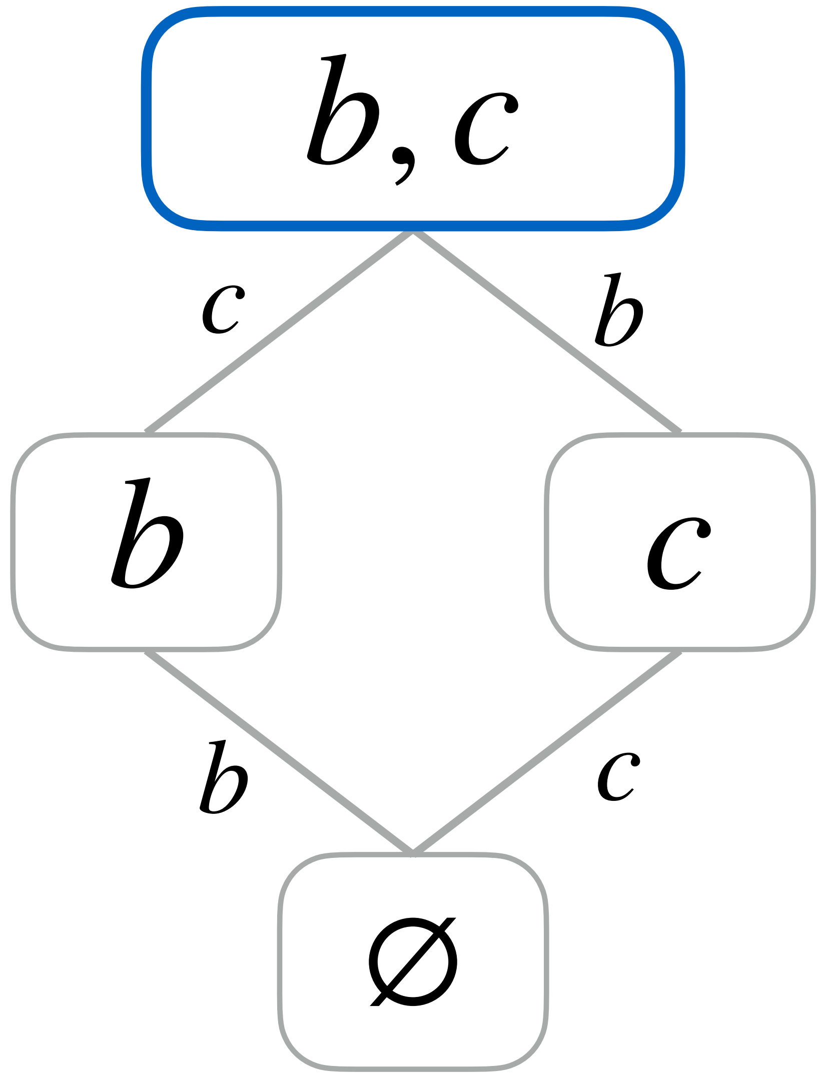



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);



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

Iteration lattice for additions

$$a_i = b_i + c_i$$

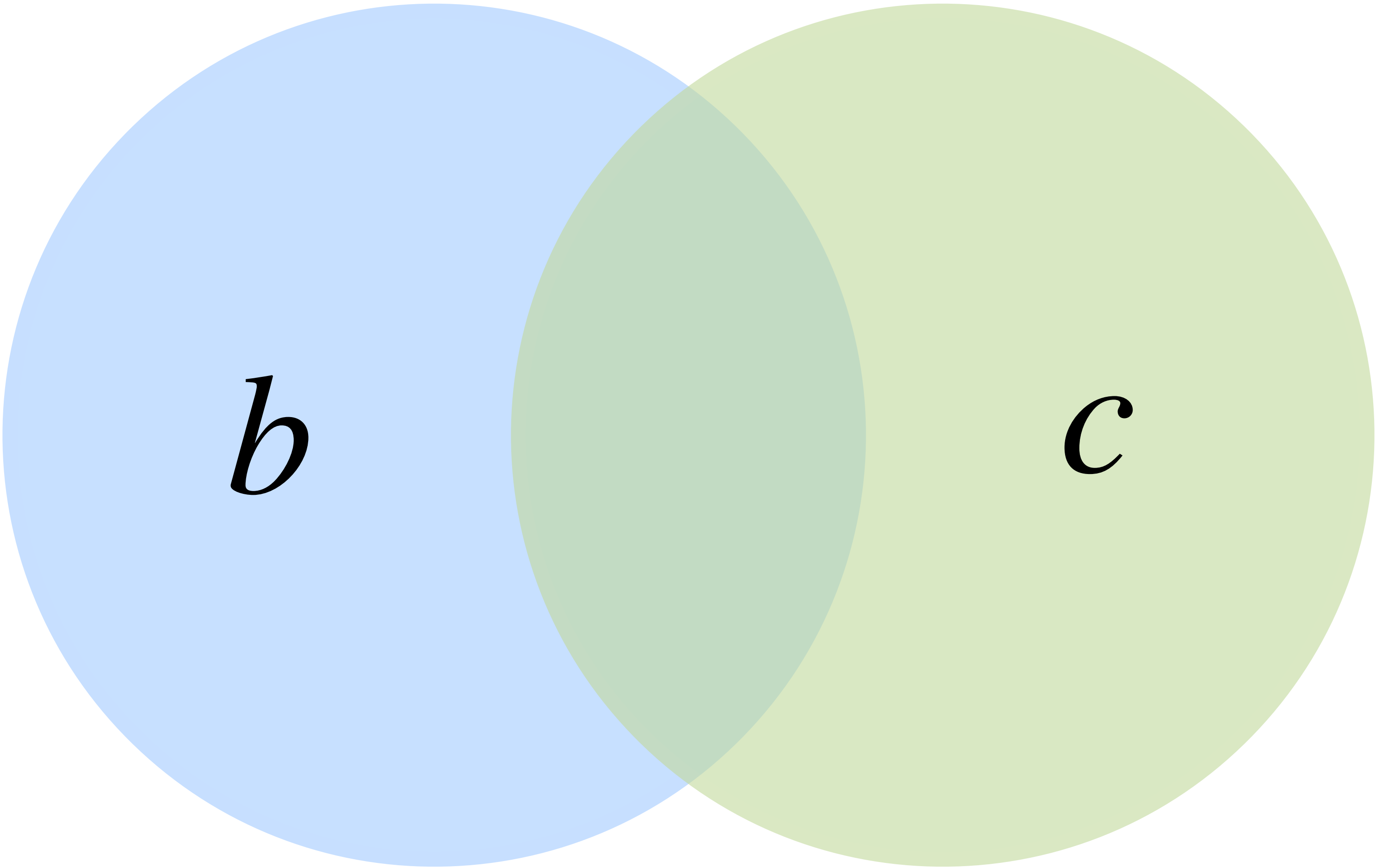


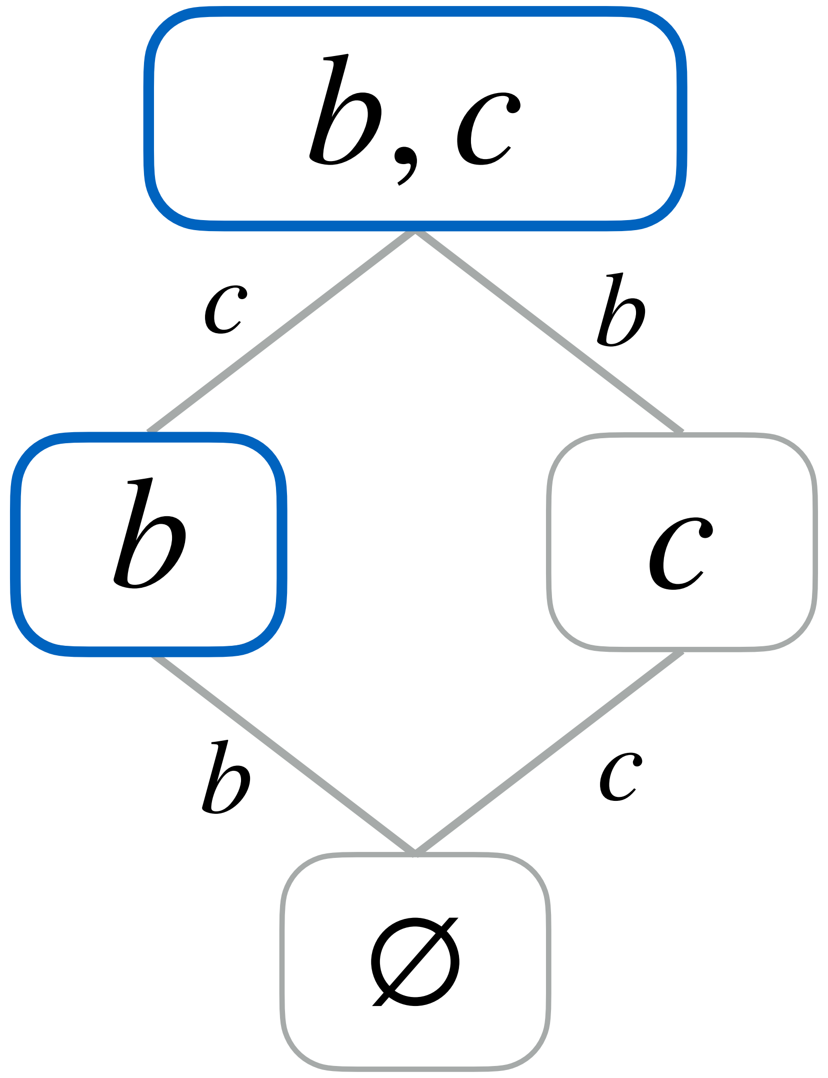



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }




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

Iteration lattice for additions

$$a_i = b_i + c_i$$

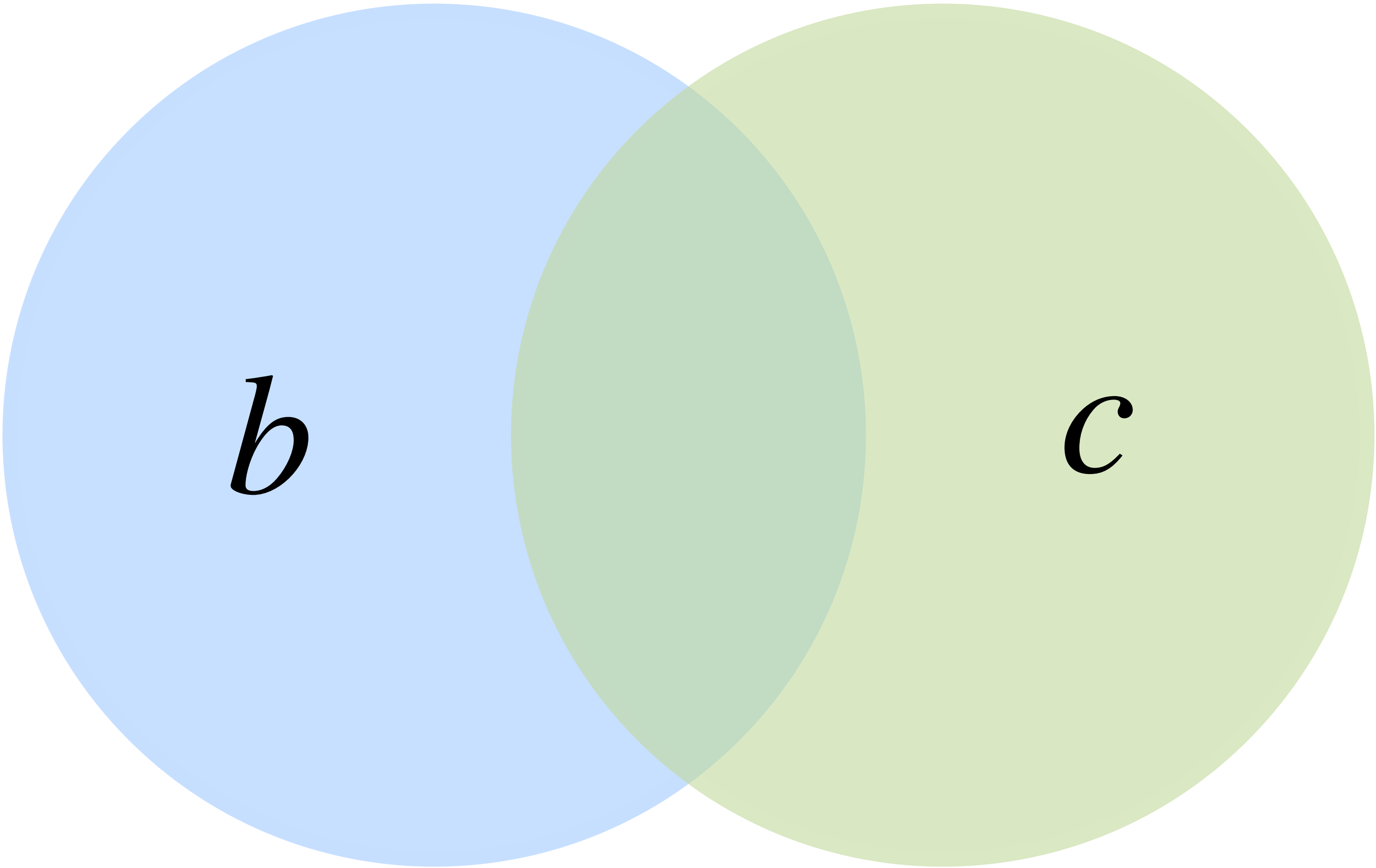


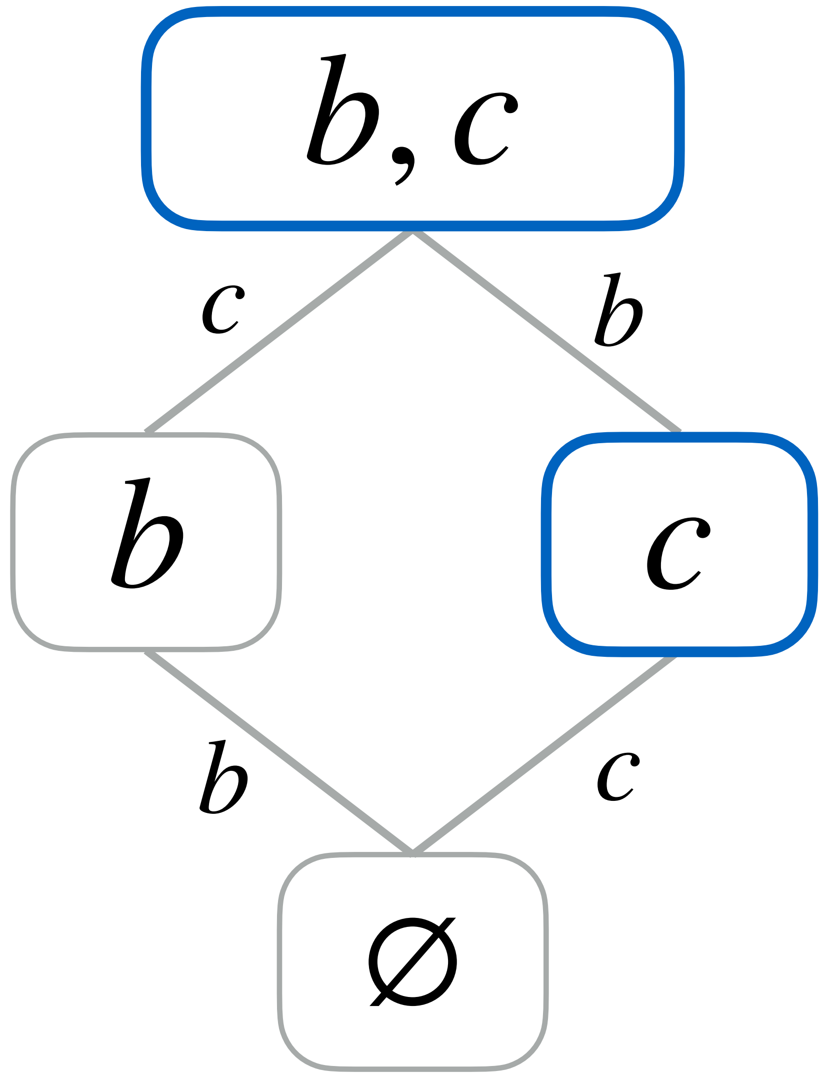



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }



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

Iteration lattice for additions

$$a_i = b_i + c_i$$

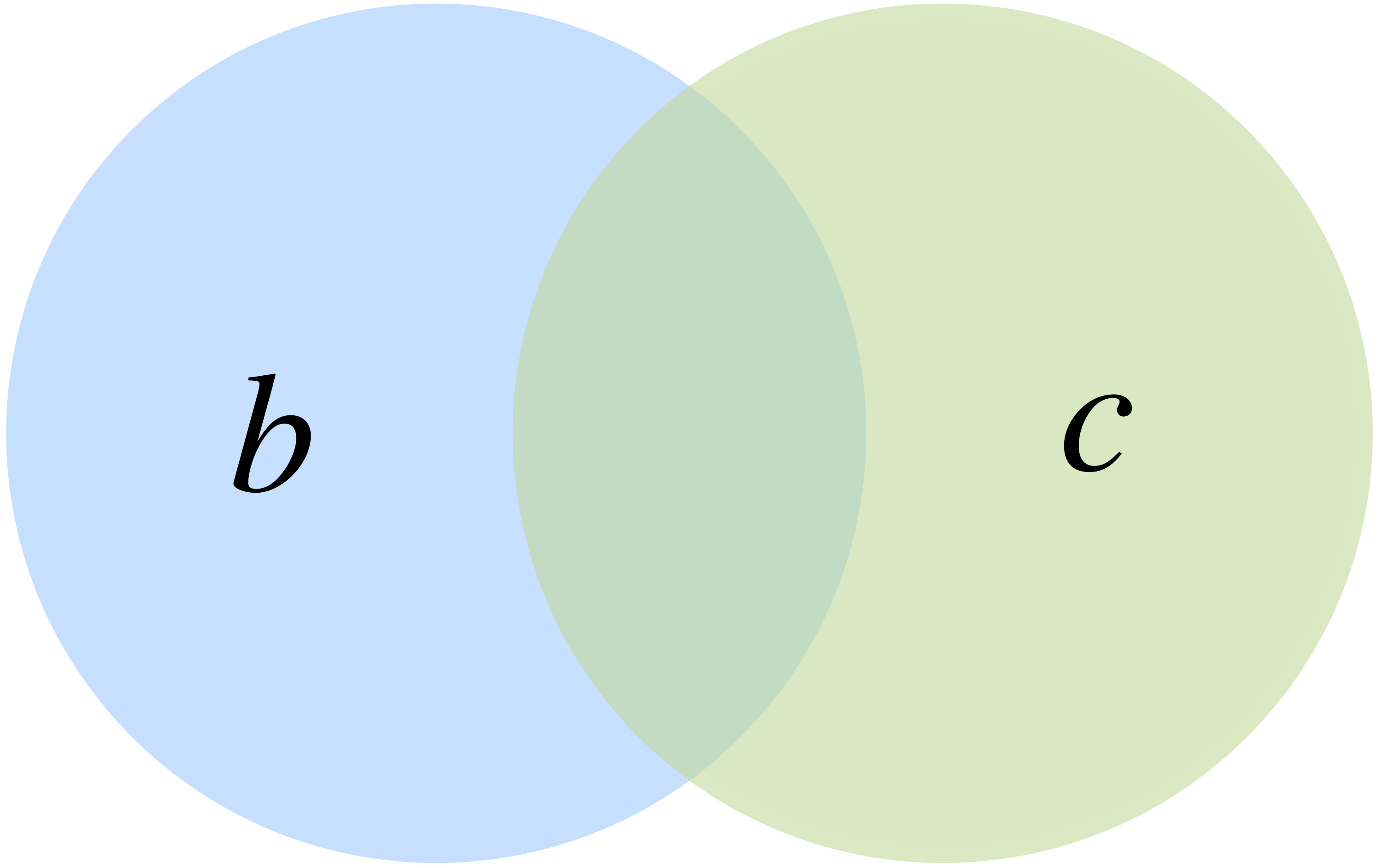


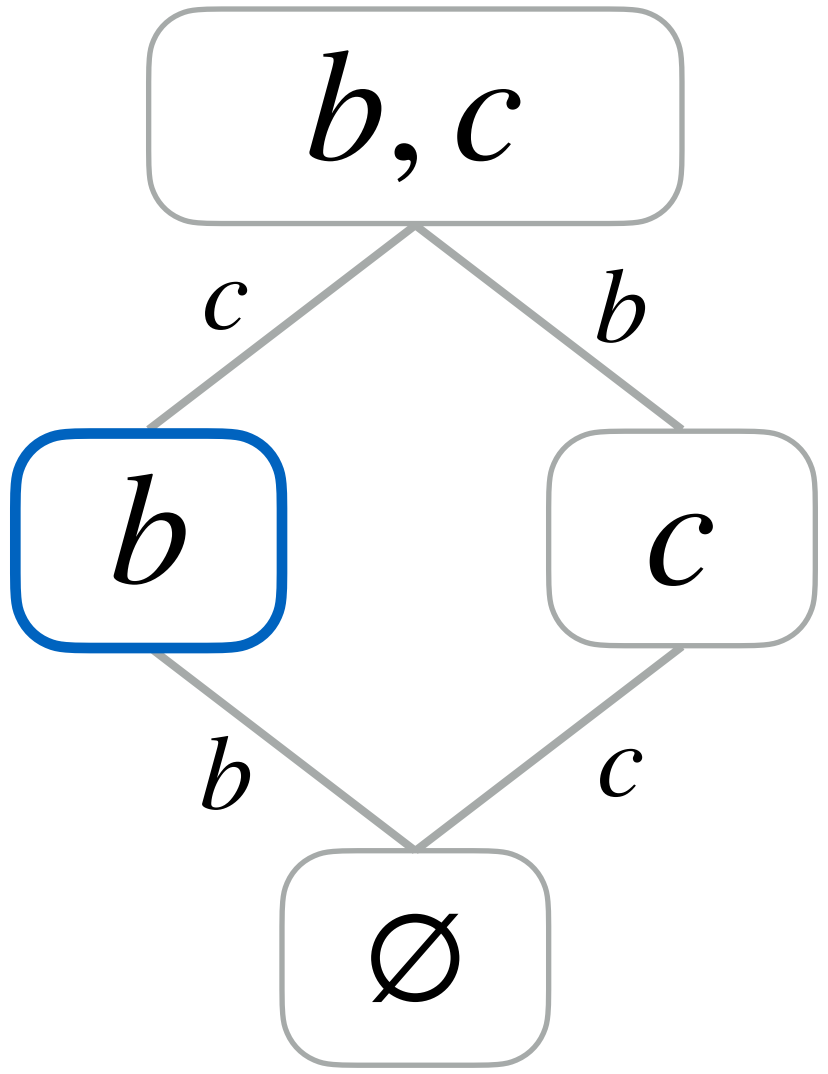



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = c[pc1];
  }
 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

Iteration lattice for additions

$$a_i = b_i + c_i$$

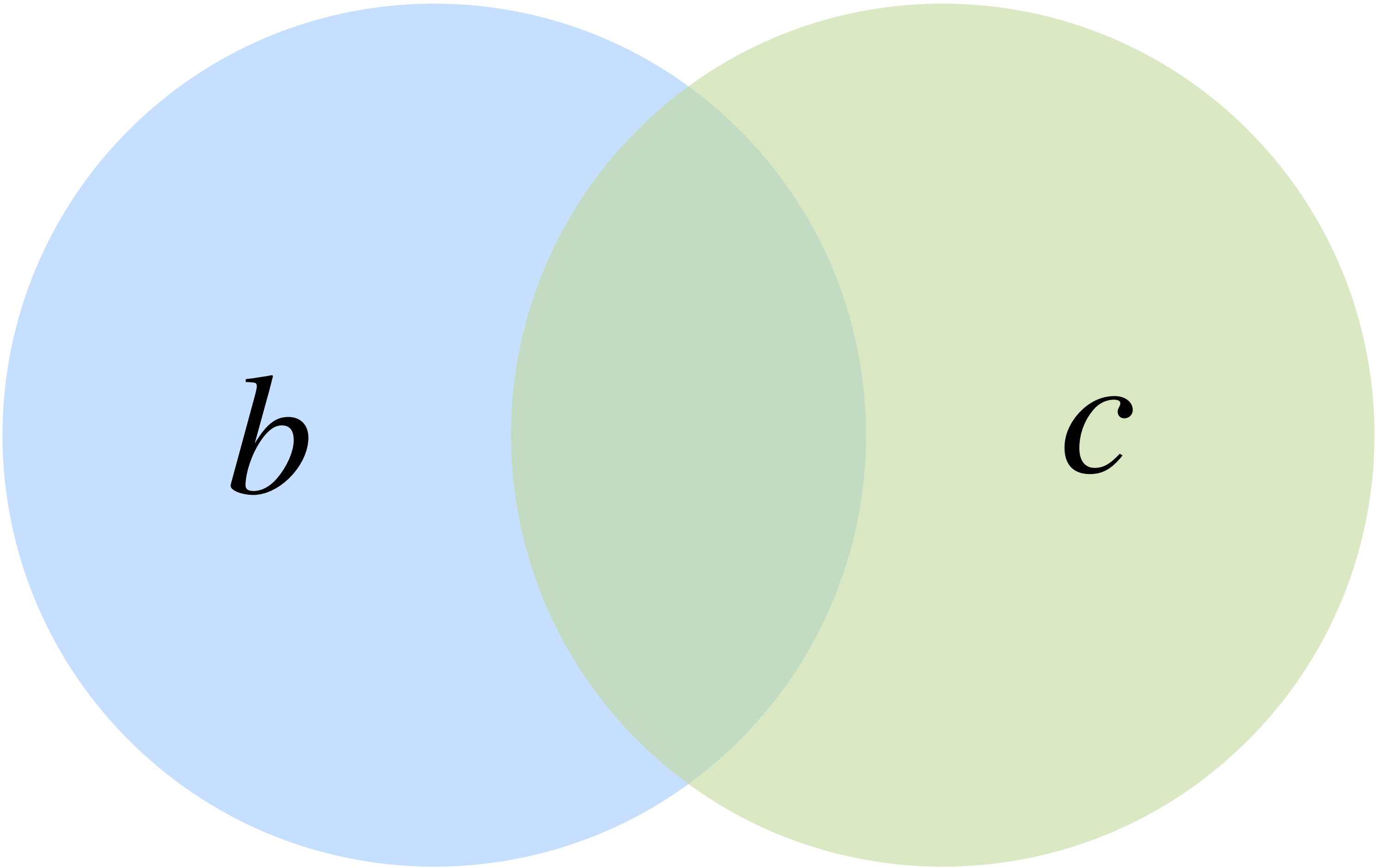


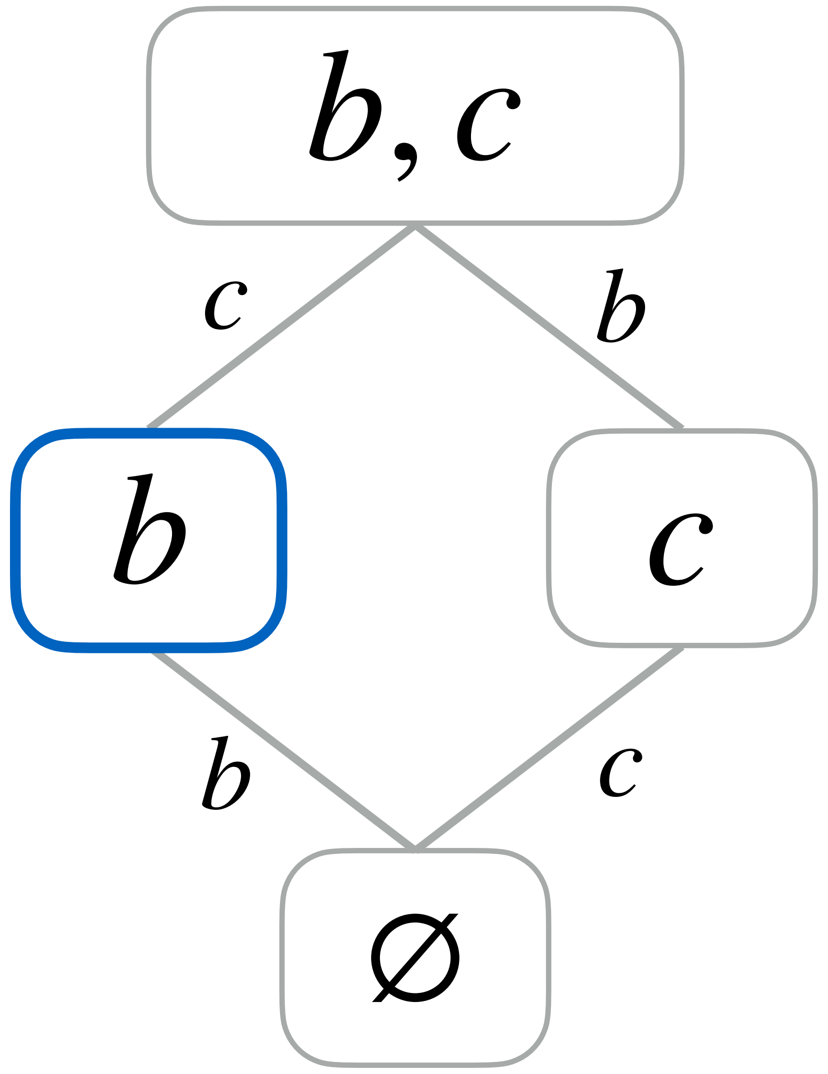



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = c[pc1];
  }
 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

Iteration lattice for additions

$$a_i = b_i + c_i$$

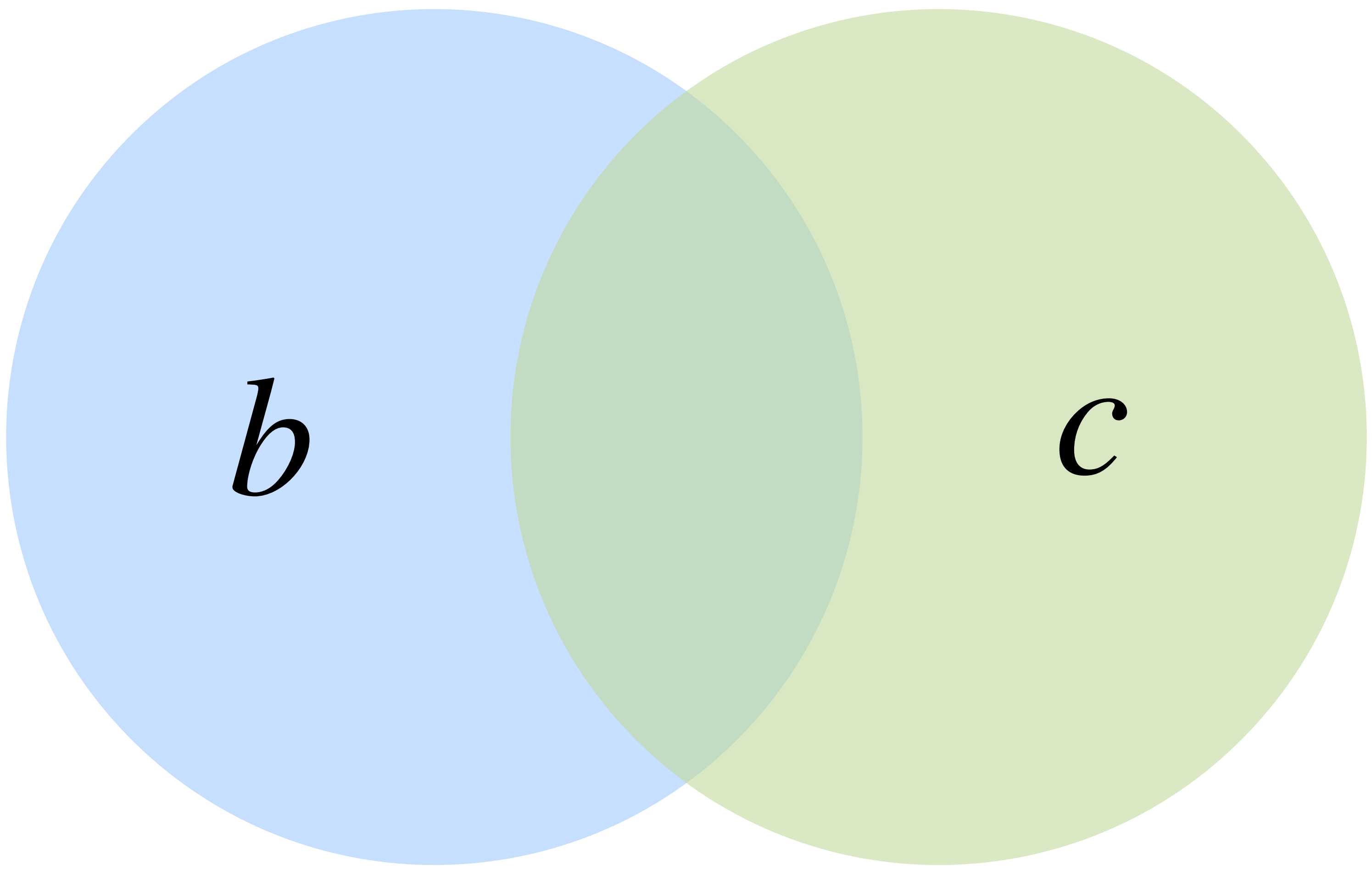


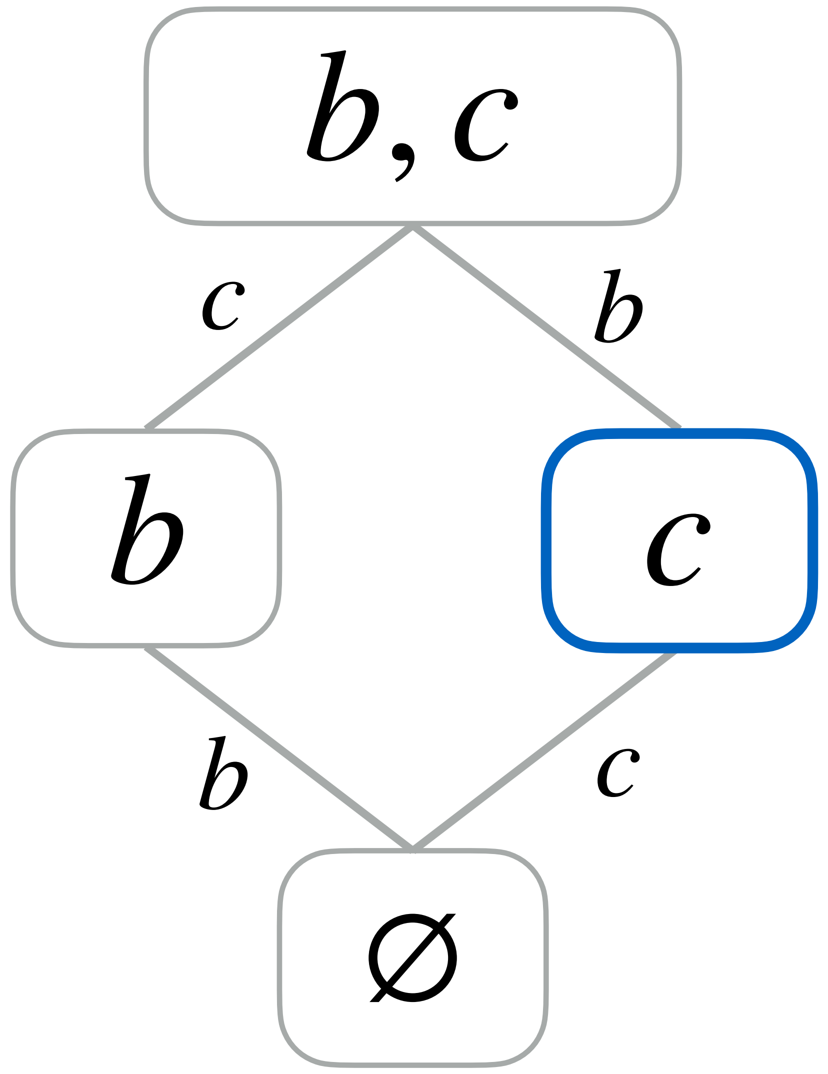



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = c[pc1];
  }
 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

$$a_i = b_i + c_i$$

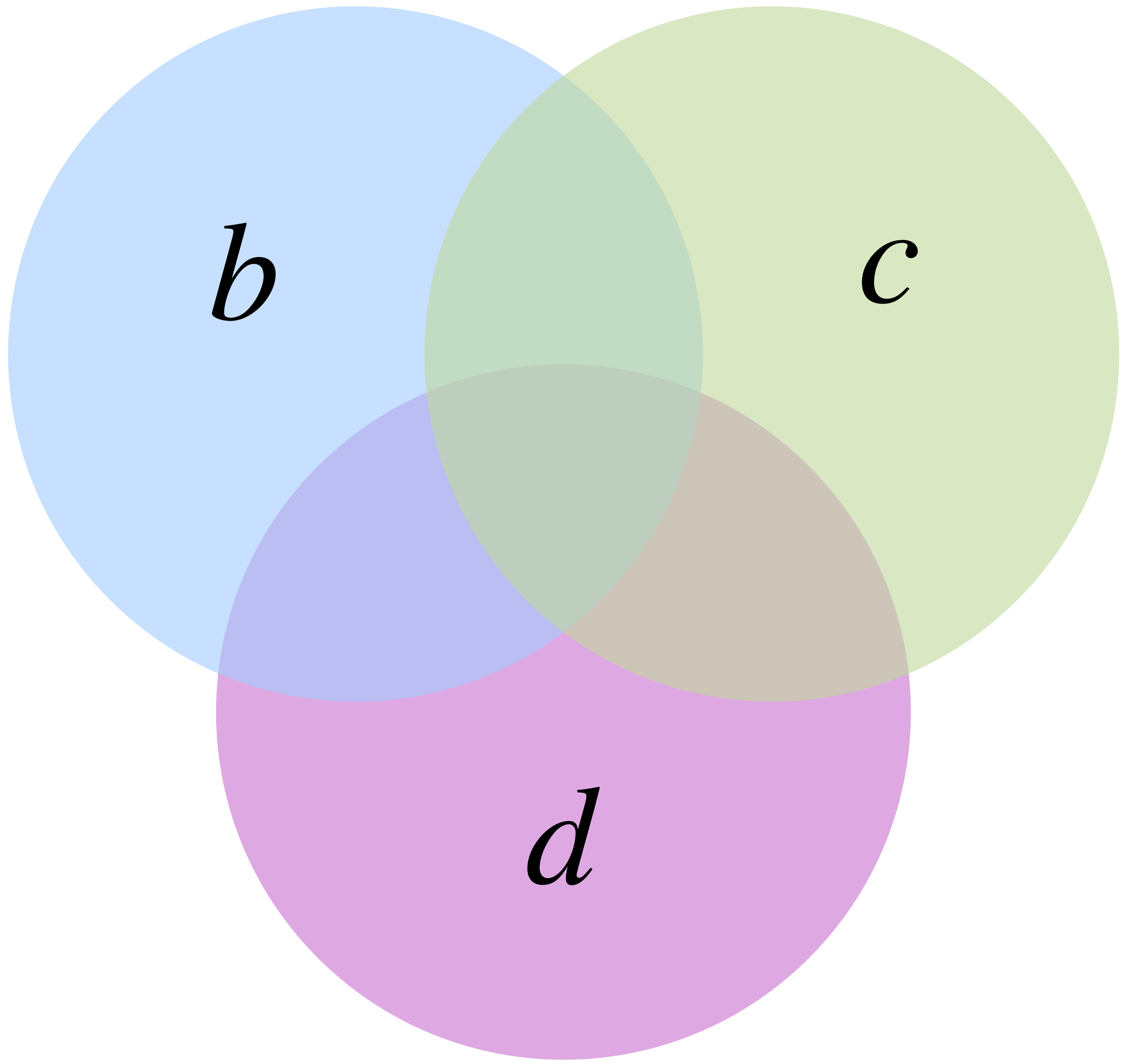


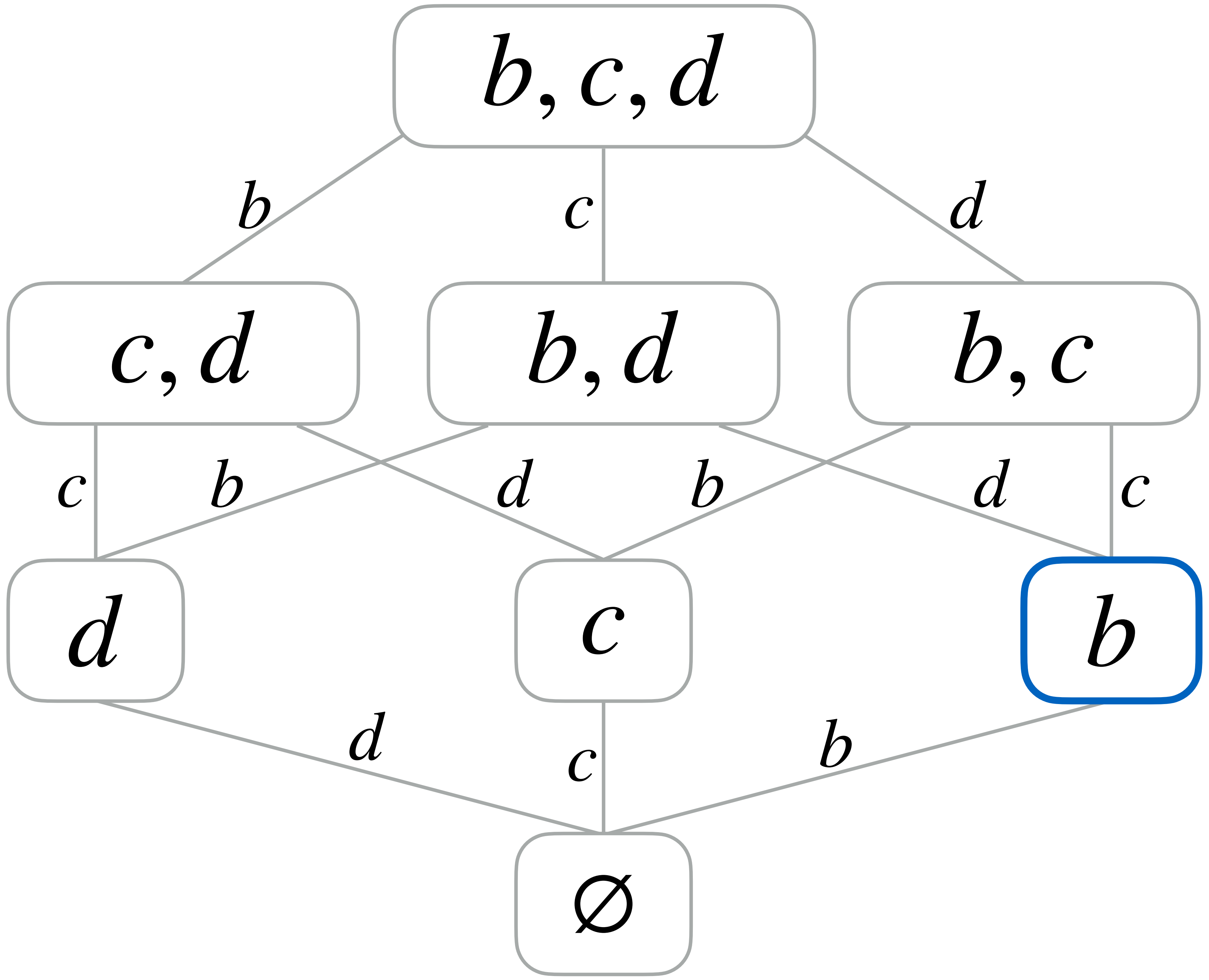



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    a[i] = b[pb1] + c[pc1];
  }
  else if (ib == i) {
    a[i] = b[pb1];
  }
  else {
    a[i] = c[pc1];
  }
 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

$$a_i = b_i + c_i + d_i$$



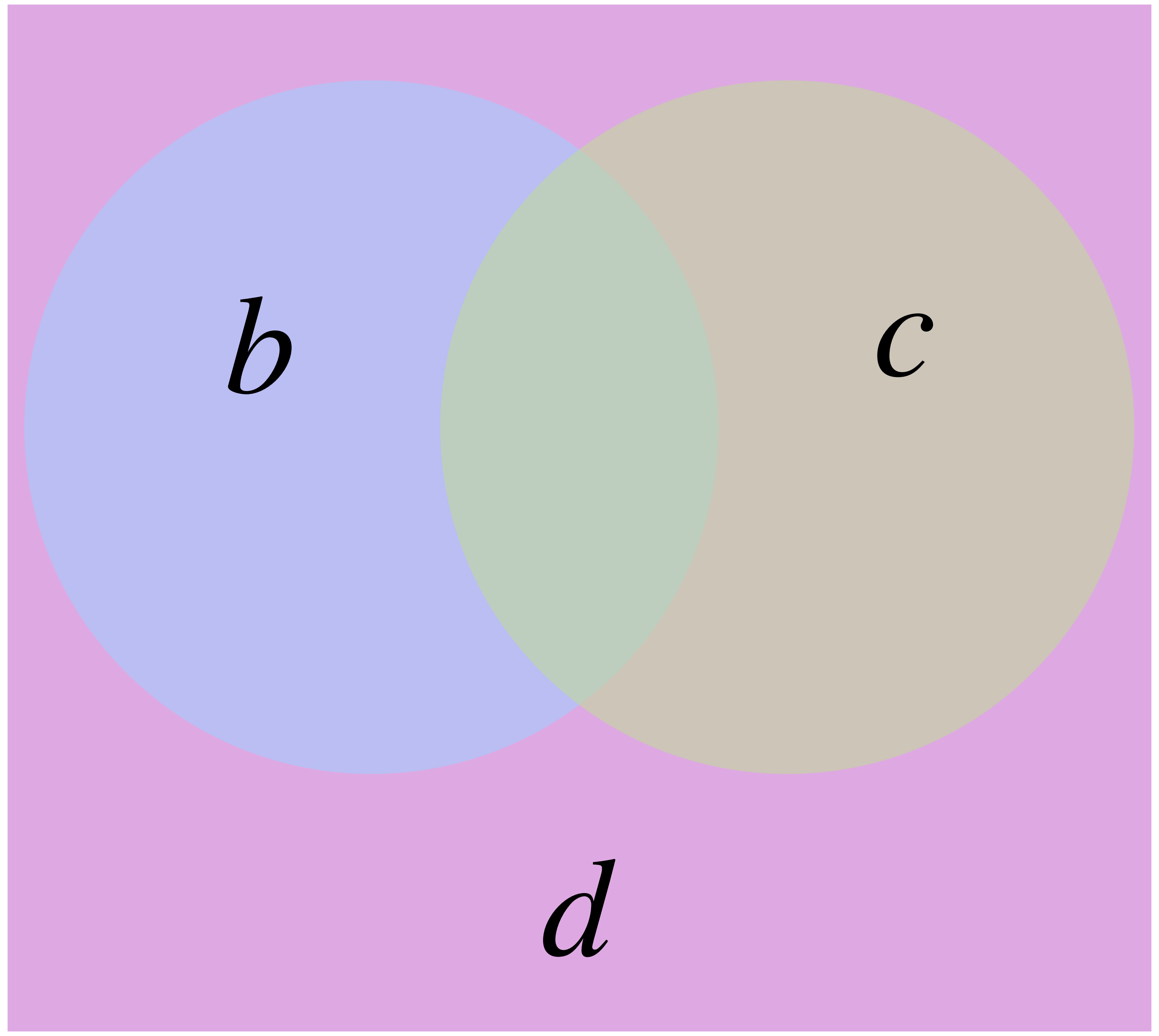


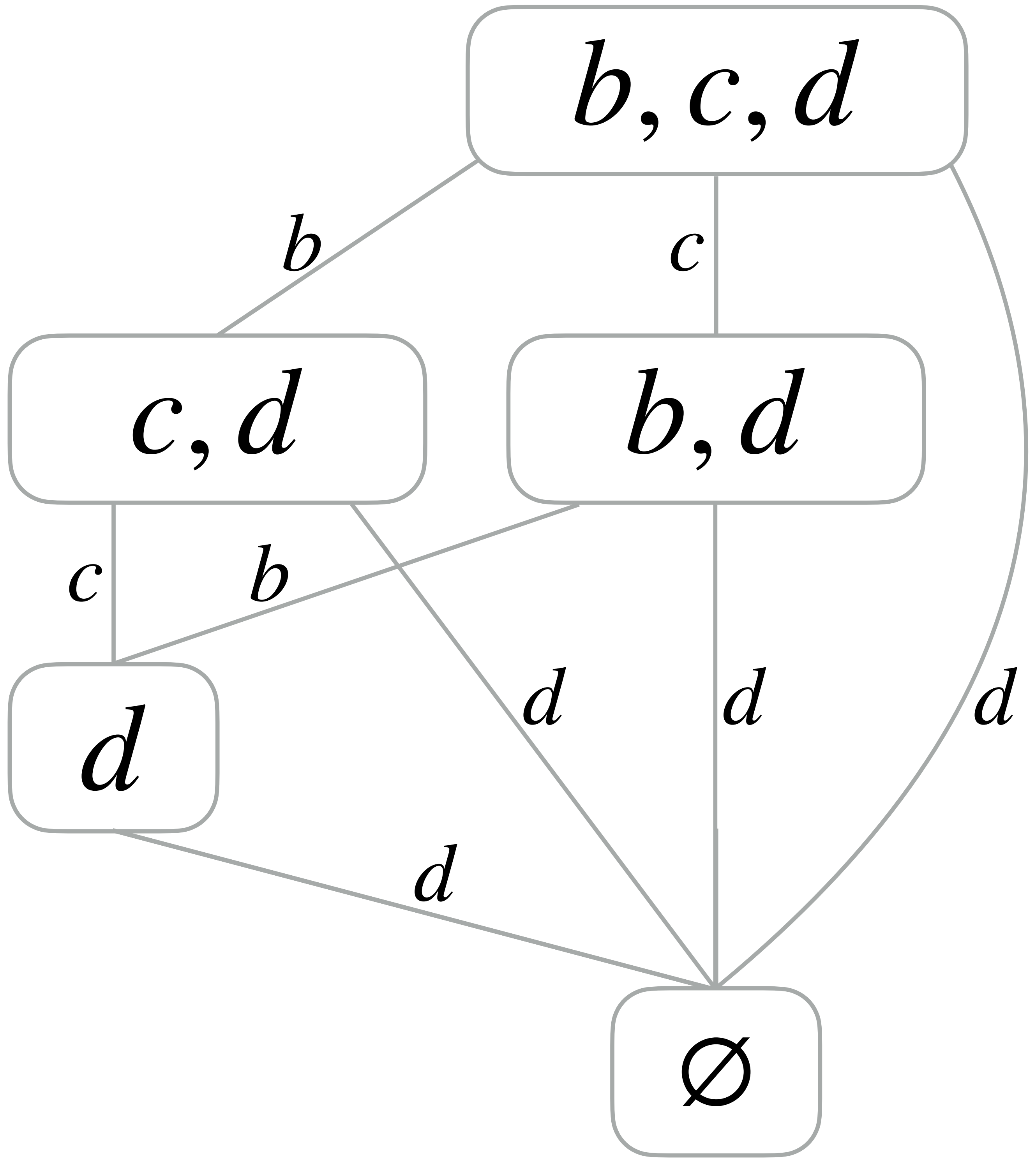

```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int id = d1_crd[pd1];
  int i = min(ib, ic, id);
  if (ib == i && ic == i && id == i) {
    a[i] = b[pb1] + c[pc1] + d[pd1];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

$$a_i = b_i + c_i + d_i$$

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++;
}
```

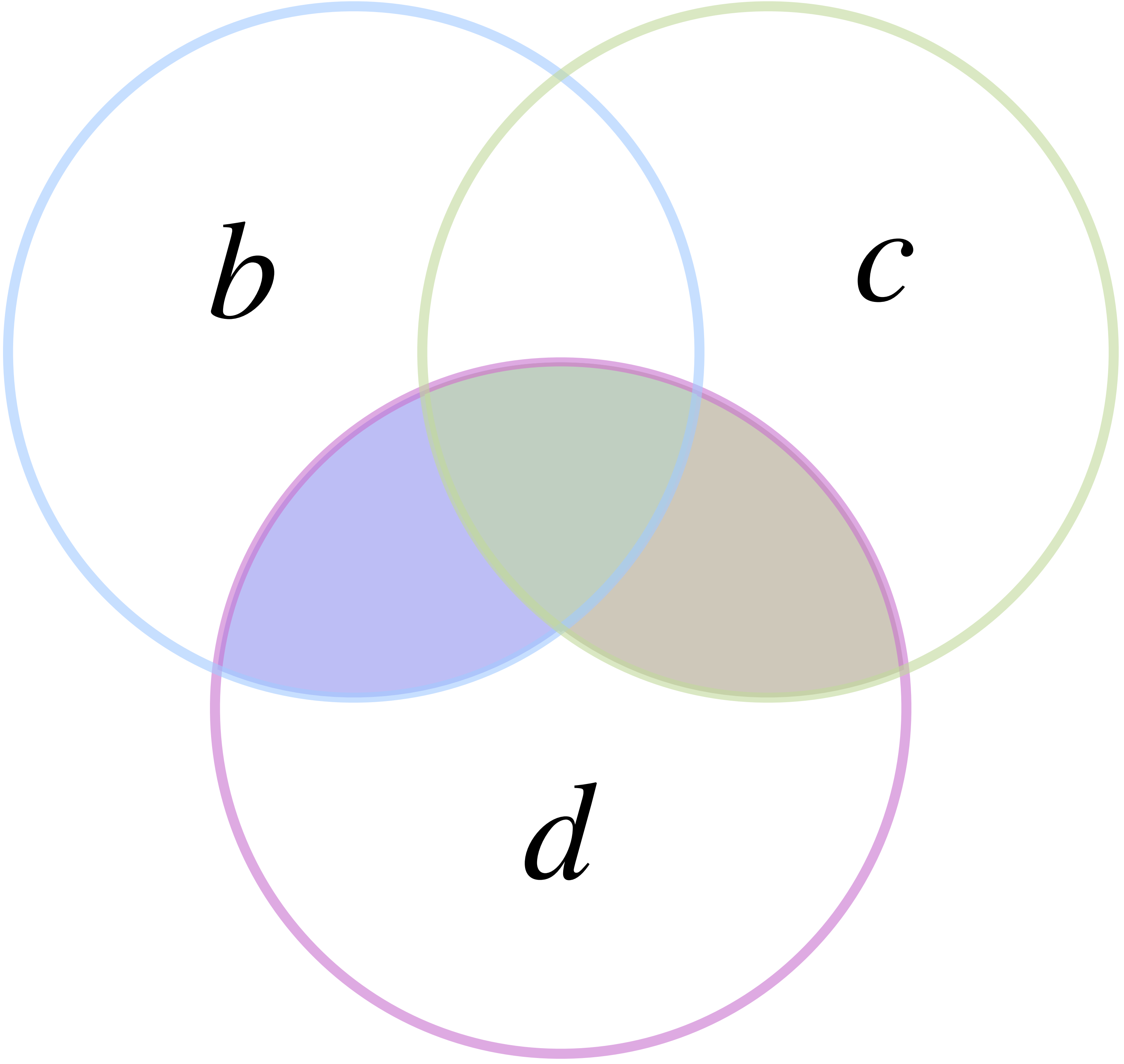
```
while (pc1 < c1_pos[1]) {
  int ic = c1_crd[pc1];
  int pd1 = id;
  int pa1 = id;
  if (ic == id) {
    a[pa1] = c[pc1] + d[pd1];
  }
  else {
    a[pa1] = d[pd1];
  }
  if (ic == id) pc1++;
  id++;
}
```

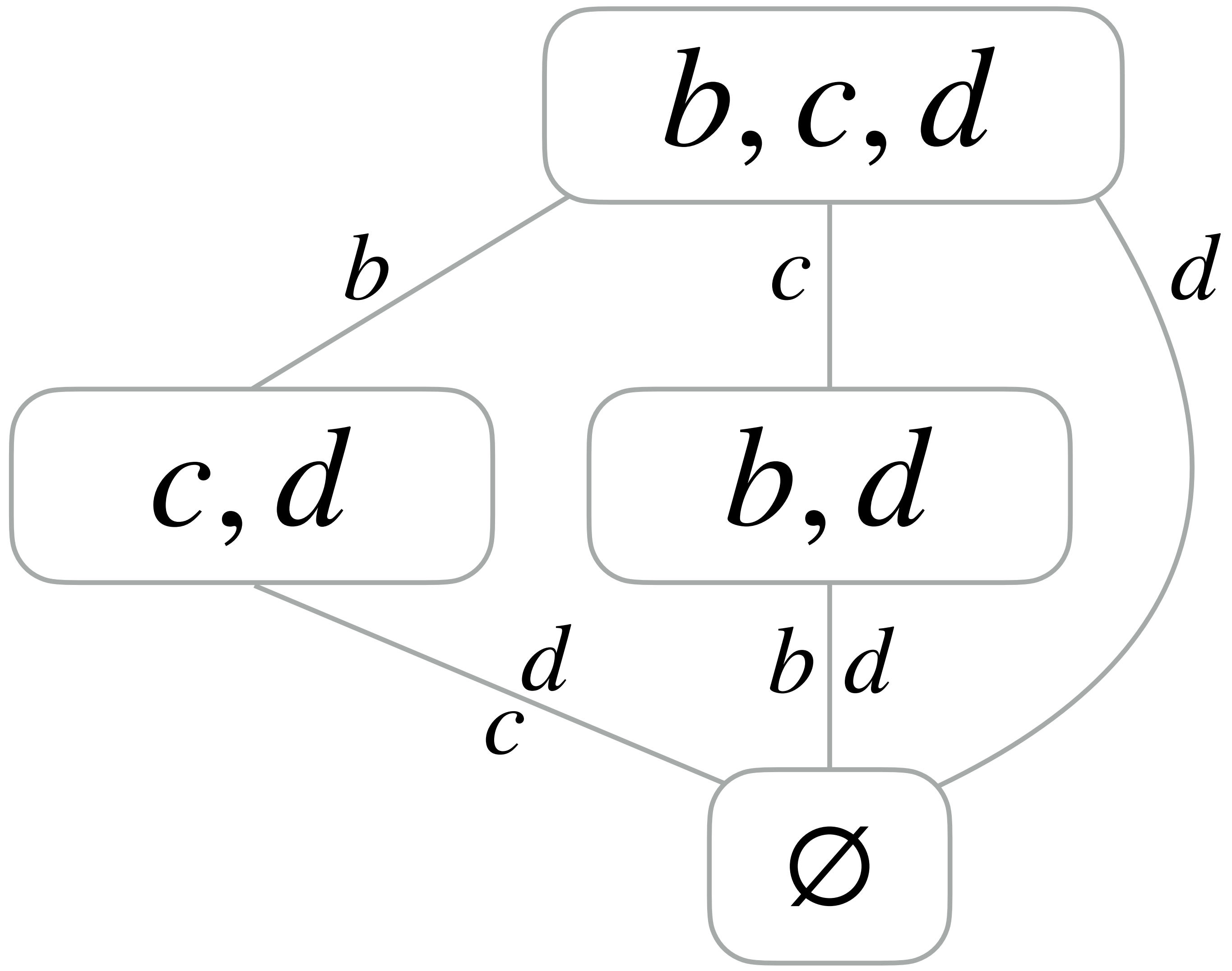
```
while (pb1 < b1_pos[1]) {
  int ib = b1_crd[pb1];
  int pd1 = id;
  int pa1 = id;
  if (ib == id) {
    a[pa1] = b[pb1] + d[pd1];
  }
  else {
    a[pa1] = d[pd1];
  }
  if (ib == id) pb1++;
  id++;
}
```

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

Iteration lattice for a compound expression

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



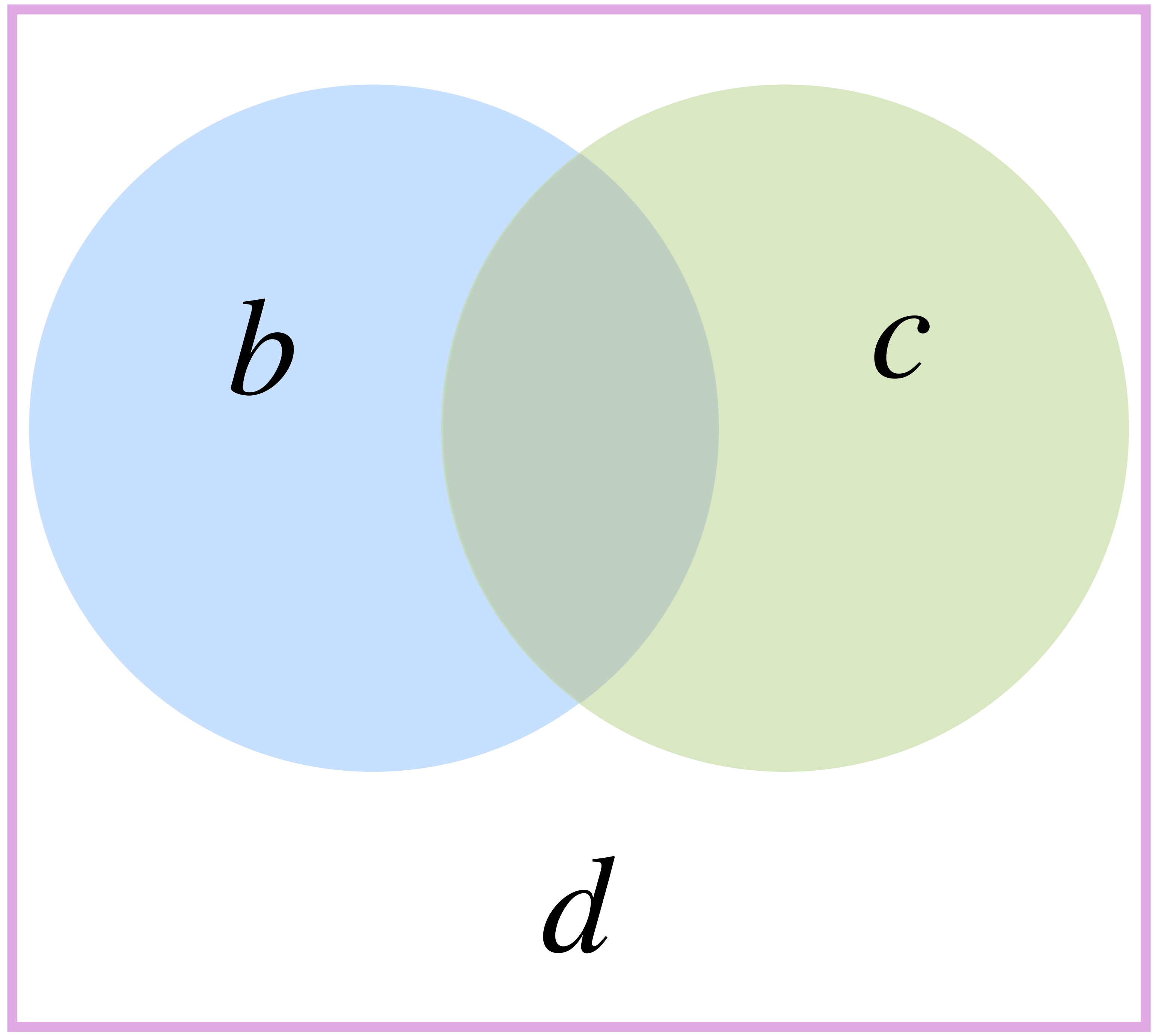


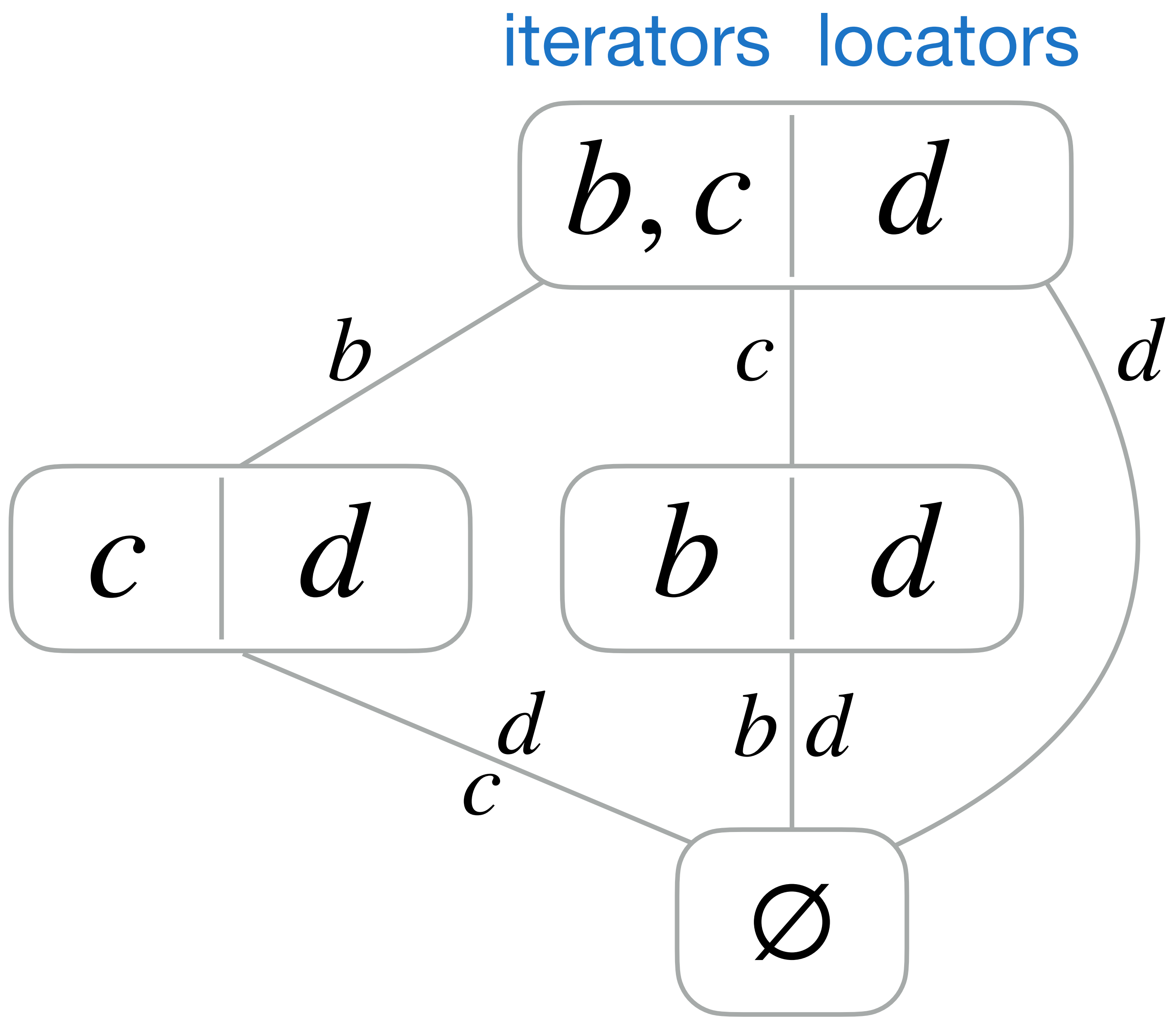

```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
  int ib = b1_crd[pb1];
  int ic = c1_crd[pc1];
  int id = d1_crd[pd1];
  int i = min(ib, ic, id);
  if (ib == i && ic == i && id == i) {
    a[i] = (b[pb1] + c[pc1]) * d[pd1];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

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

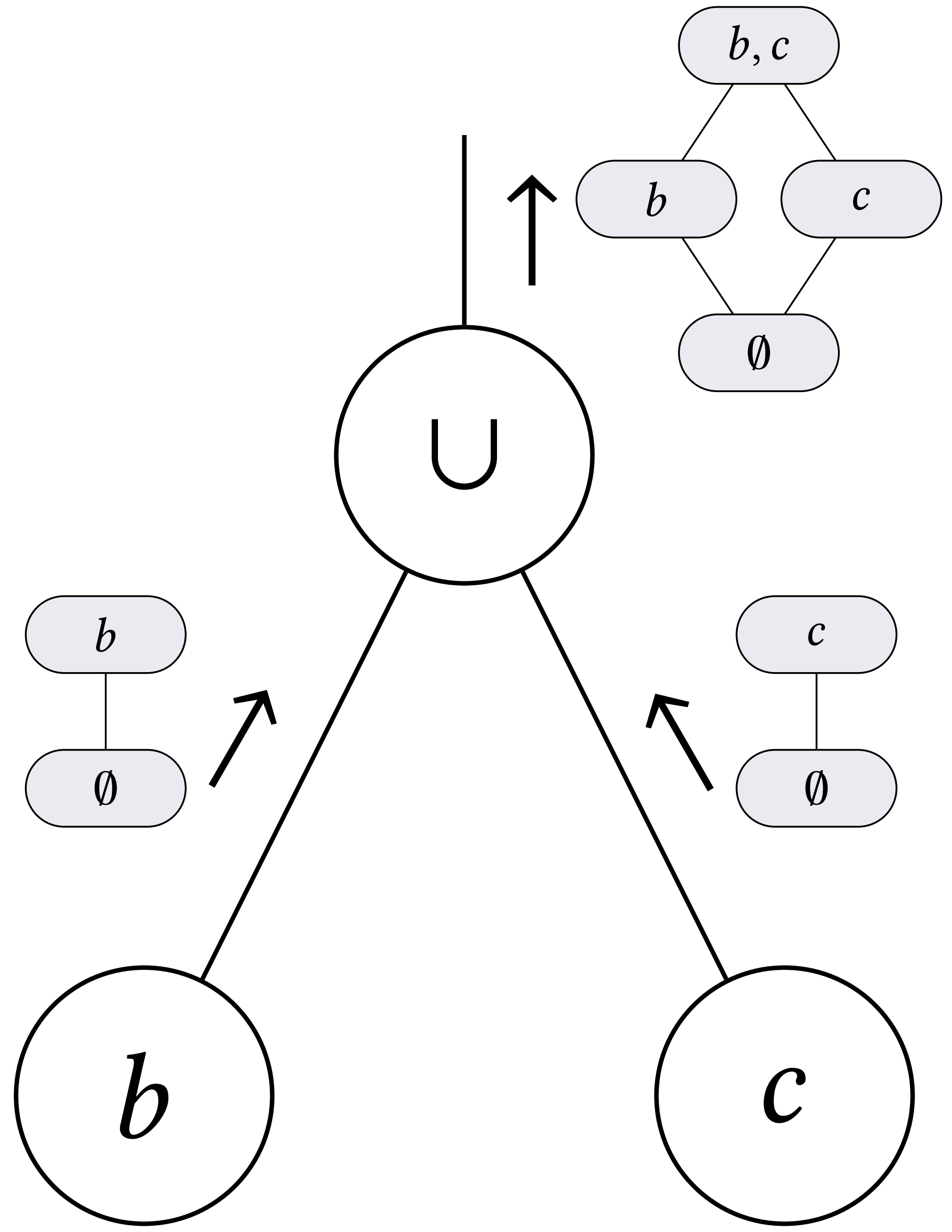
Dense






```
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++;
}
```

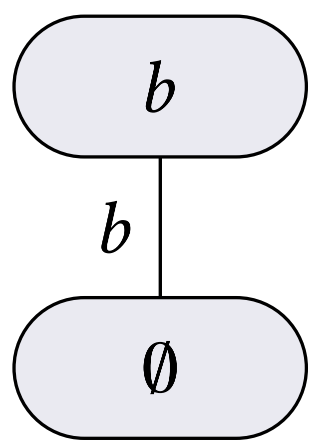
Iteration lattice construction



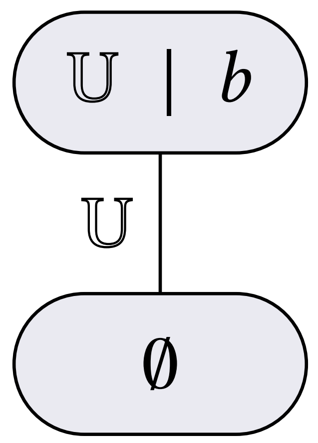
$b_i \cup c_i$

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

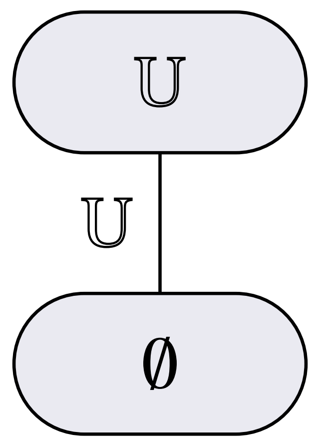
Base cases:



b has an iterator

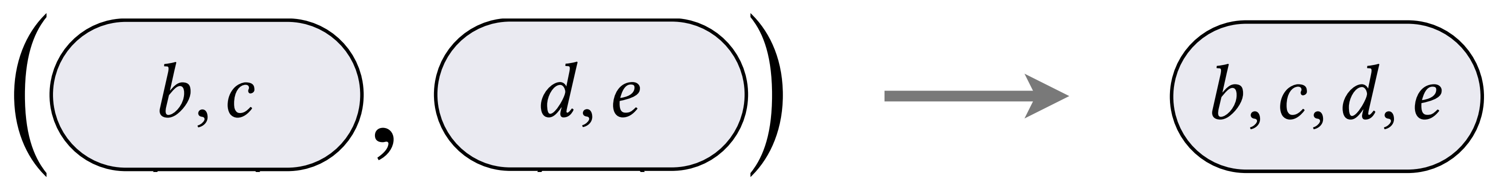


b does not have an iterator,
but supports locate



b is the set universe

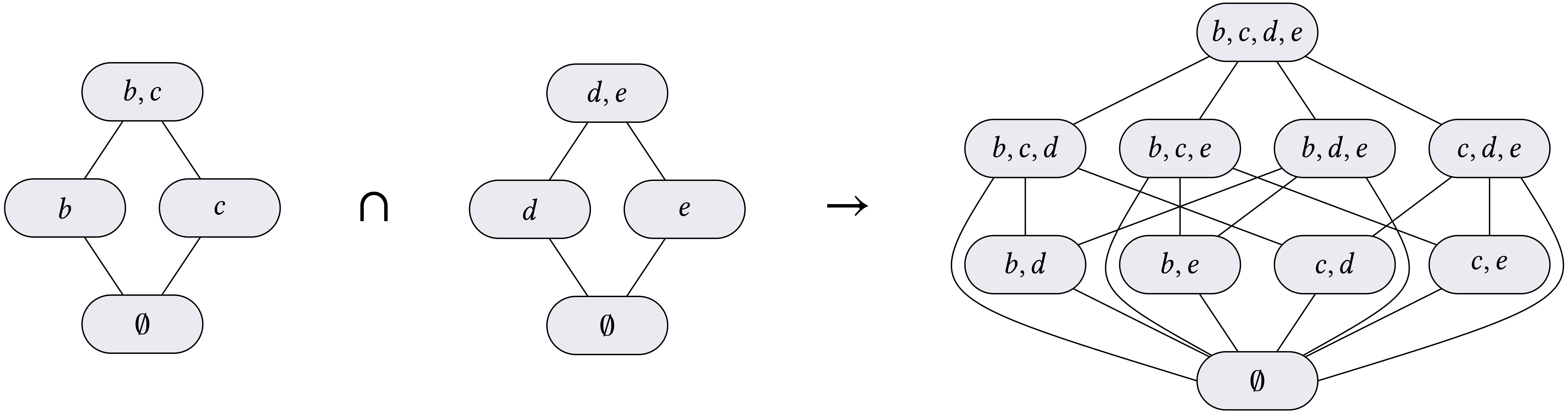
Lattice point merging:



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

Iteration lattice construction

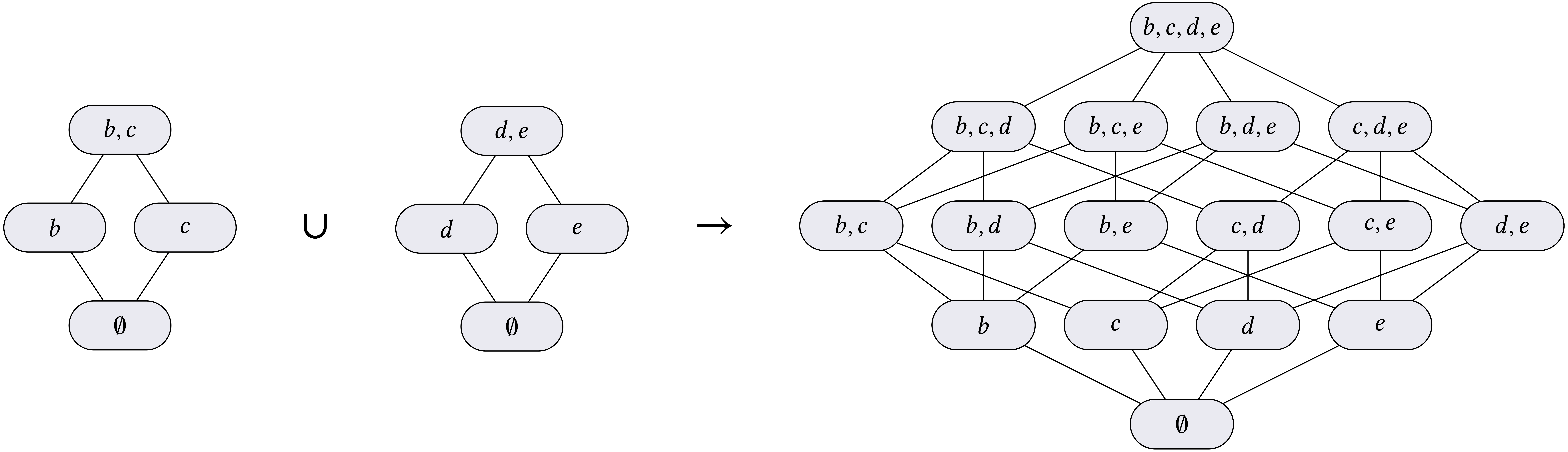
Intersection



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

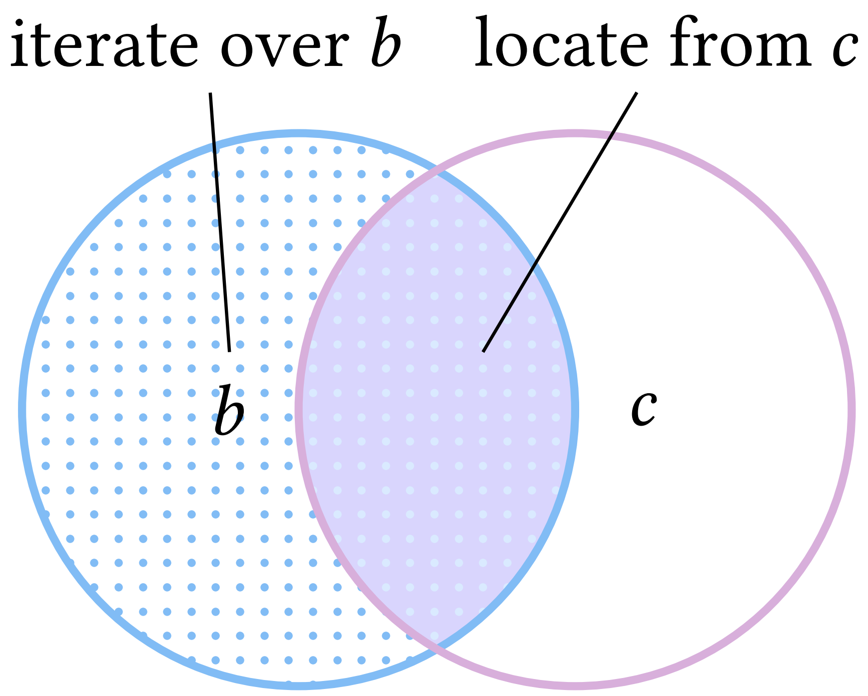
Iteration lattice construction

Union

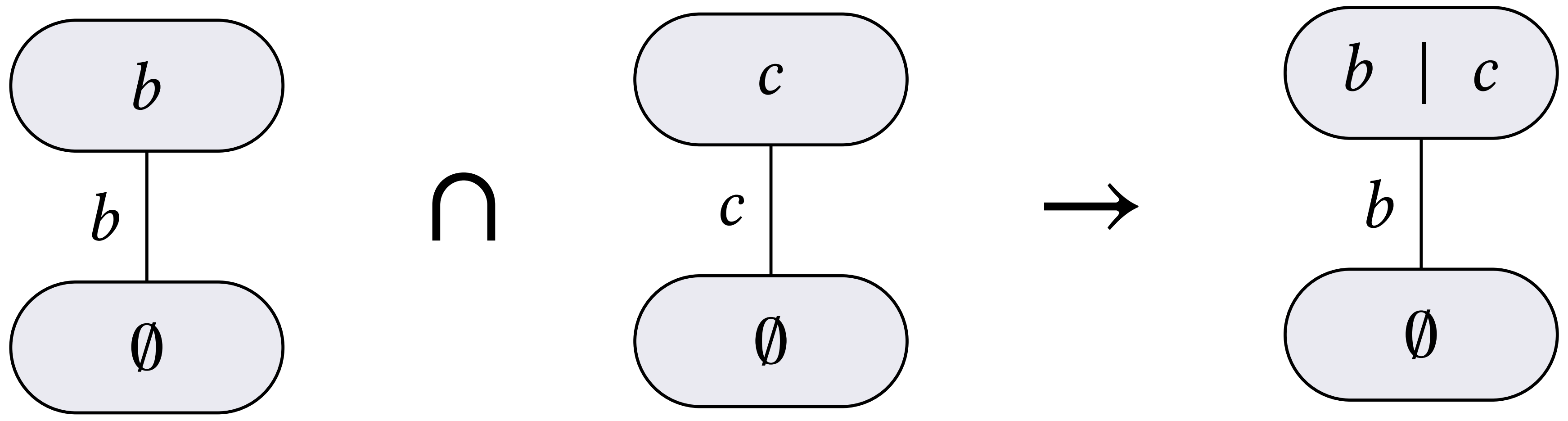


The union of two lattices is computed by first merging the lattice point pairs in the Cartesian combination of their lattice points. The union of the lattices is then the union of the result and the two initial lattices.

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.

Overview of lectures in the coming weeks