CERC 2017: Presentation of solutions

University of Zagreb

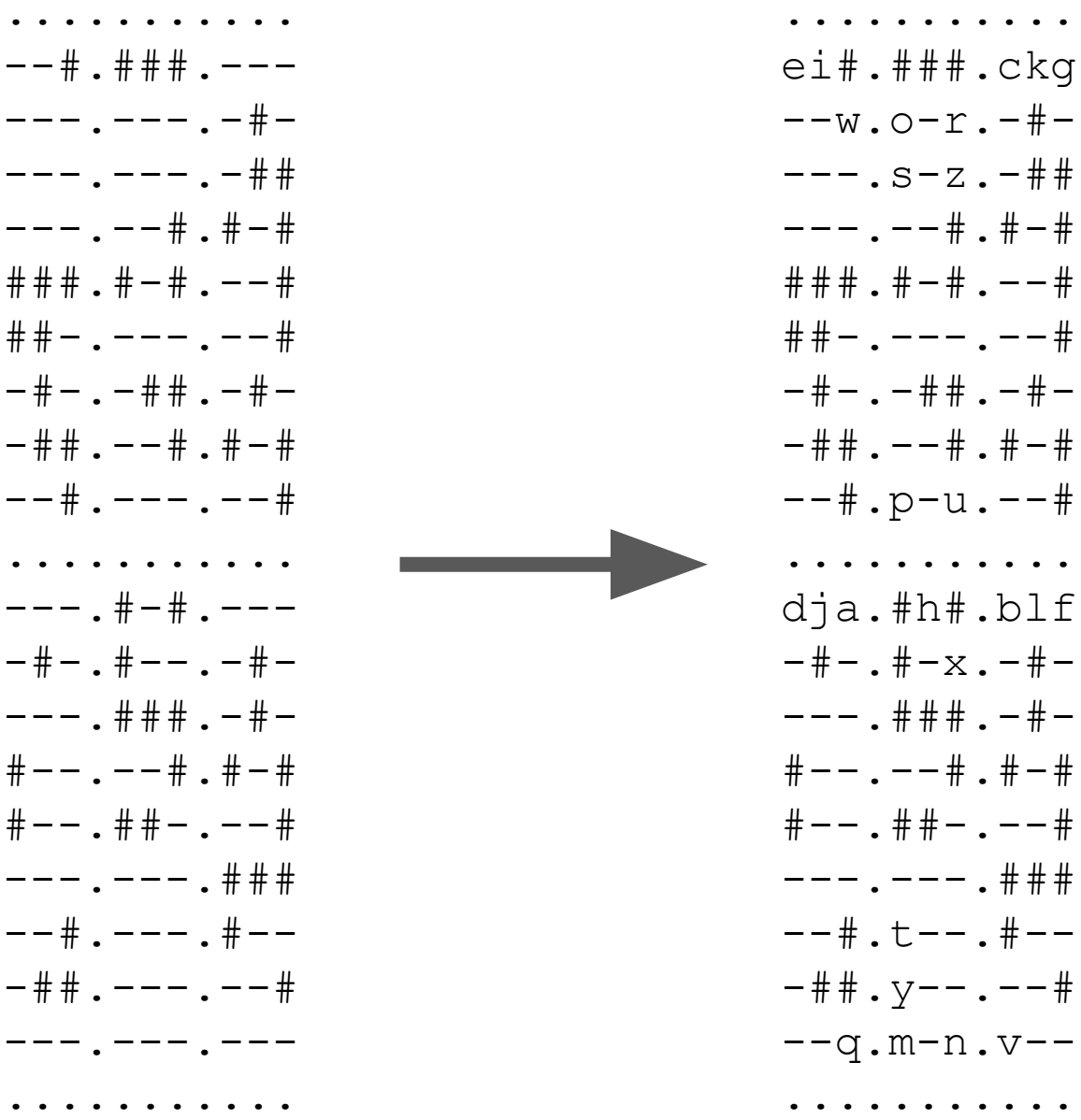
A:	Assignment Algorithm	Easy
B:	Buffalo Barricades	Hard
C:	Cumulative Code	Hard
D:	Donut Drone	Medium
E:	Embedding Enumeration	Hard
F:	Faulty Factorial	Easy
G:	Gambling Guide	Medium
H:	Hidden Hierarchy	Easy
I:	Intrinsic Interval	Hard
J:	Justified Jungle	Easy
K:	Kitchen Knobs	Hard
L:	Lunar Landscape	Medium

Problem A
Assignment Algorithm

Submits: 97
Accepted: at least 56

First solved by: FI MUNI 01
Masaryk University
(Fabík, Pokorný, Priessnitz)
00:37:18

Author: Ivan Paljak


....
--#.###.---
---.---.-#-
---.---.-##
---.--#.#-#
###.#-#.--#
##-.---.--#
-#-.-##.-#-
-##.--#.#-#
--#.---.--#
....
---.#-#.---
-#-.#--.-#-
---.###.-#-
#--.--#.#-#
#--.##-.--#
---.---.###
--#.---.#--
-##.---.--#
---.---.---
....
....
ei#.###.ckg
--w.o-r.-#-
---.s-z.-##
---.--#.#-#
###.#-#.--#
##-.---.--#
-#-.-##.-#-
-##.--#.#-#
--#.p-u.--#
....
dja.#h#.blf
-#-.#-x.-#-
---.###.-#-
#--.--#.#-#
#--.##-.--#
---.---.###
--#.t--.#--
-##.y--.--#
--q.m-n.v--
....

Implement the rules carefully.

Break down the complex algorithm into smaller simple pieces that are easy to implement.

Tip: Use helper functions.

- NumEmptySeats(row)
- SelectRow()
- GetSeatPriority(column)
- GetPlaneBalance()
- SelectSeat(row)
- ...

Problem H
Hidden Hierarchy

Submits: 95
Accepted: at least 52

First solved by: MFF3
Charles University in Prague
(Konečný, Madaj, Rozhoň)
00:22:48

Author: Luka Kalinovčić

Files

```
/sys/kernel/notes 100
/cerc/problems/a/testdata/in 1000000
/cerc/problems/a/testdata/out 8
/cerc/problems/a/luka.cc 500
/cerc/problems/a/zuza.cc 5000
/cerc/problems/b/testdata/in 15
/cerc/problems/b/testdata/out 4
/cerc/problems/b/kale.cc 100
/cerc/documents/rules.pdf 4000
```

Directory tree

```
- / 1009727
- /cerc/ 1009627
  /cerc/documents/ 4000
- /cerc/problems/ 1005627
- /cerc/problems/a/ 1005508
  /cerc/problems/a/testdata/ 1000008
- /cerc/problems/b/ 119
  /cerc/problems/b/testdata/ 19
- /sys/ 100
  /sys/kernel/ 100
```

Files

```
/sys/kernel/notes 100
/cerc/problems/a/testdata/in 1000000
/cerc/problems/a/testdata/out 8
/cerc/problems/a/luka.cc 500
/cerc/problems/a/zuza.cc 5000
/cerc/problems/b/testdata/in 15
/cerc/problems/b/testdata/out 4
/cerc/problems/b/kale.cc 100
/cerc/documents/rules.pdf 4000
```

Directory tree

```
- / 1009727
- /cerc/ 1009627
  /cerc/documents/ 4000
- /cerc/problems/ 1005627
- /cerc/problems/a/ 1005508
  /cerc/problems/a/testdata/ 1000008
+ /cerc/problems/b/ 119
- /sys/ 100
  /sys/kernel/ 100
```

Files

```
/sys/kernel/notes 100
/cerc/problems/a/testdata/in 1000000
/cerc/problems/a/testdata/out 8
/cerc/problems/a/luka.cc 500
/cerc/problems/a/zuza.cc 5000
/cerc/problems/b/testdata/in 15
/cerc/problems/b/testdata/out 4
/cerc/problems/b/kale.cc 100
/cerc/documents/rules.pdf 4000
```

Directory tree

```
- / 1009727
- /cerc/ 1009627
  /cerc/documents/ 4000
- /cerc/problems/ 1005627
- /cerc/problems/a/ 1005508
  /cerc/problems/a/testdata/ 1000008
+ /cerc/problems/b/ 119
+ /sys/ 100
```

Step 1: Build the directory tree.
For each file:
    Make a list p of parent directories up to the root
    For each dir in list p:
        Add file size to dir size
    For each adjacent dir_A, dir_B in list p:
        Add dir_B to the set of dir_A's subdirectories

Step 2: Find directories to collapse.
Collapse a dir if:
a) It has subdirectories, and
b) size of each subdirectory is below threshold.

Step 3: Print the directory tree recursively.

Tip: Consider Python.

Problem F
Faulty Factorial

Submits: 229
Accepted: at least 32

First solved by: UW3
University of Warsaw
(Hołubowicz, Paluszek, Tabaszewski)
00:38:14

Author: Lovro Pužar

Faulty factorial: Take any factor of a factorial and make it smaller, but keep it positive.

Factorial: $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8$

Faulty factorial: $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \mathbf{2} \cdot 7 \cdot 8$

Problem: Find any faulty factorial of length n that gives reminder r when divided by prime number p.

Case r = 0:

If n < p:

None of the factors is divisible by p: impossible.

Problem: Find any faulty factorial of length n that gives reminder r when divided by prime number p.

Case r = 0:

If n < p:

None of the factors is divisible by p: impossible.

Else:

The factorial is already divisible by p, just don’t mess it up. Impossible when n = p = 2.

Problem: Find any faulty factorial of length n that gives reminder r when divided by prime number p.

Case r > 0:

If n >= 2p:

Two factors divisible by p, we can’t make both smaller: impossible.

Problem: Find any faulty factorial of length n that gives reminder r when divided by prime number p.

Case r > 0:

- If n >= 2p:
  - Two factors divisible by p, we can't make both smaller: impossible.
- Else if n >= p:
  - We need to change the factor p, if possible.

Problem: Find any faulty factorial of length n that gives reminder r when divided by prime number p.

Case r > 0:

- If n >= 2p:
  - Two factors divisible by p, we can't make both smaller: impossible.
- Else if n >= p:
  - We need to change the factor p, if possible.
- Else:
  - n < p <= 10000000, so we can try each factor.

Problem: Find a faulty factorial of length $n < p$, with a fault at position i, that gives reminder $r > 0$ when divided by prime number p.

We are looking for x such that:

$$n! / i \cdot x \equiv r \qquad \text{(modulo p)}$$

Problem: Find a faulty factorial of length $n < p$, with a fault at position i, that gives reminder $r > 0$ when divided by prime number p.

We are looking for x such that:

$$n! / i \cdot x \equiv r \quad \text{(modulo } p\text{)}$$

$$x \equiv r \cdot i / n! \quad \text{(modulo } p\text{)}$$

Problem: Find a faulty factorial of length $n < p$, with a fault at position i, that gives reminder $r > 0$ when divided by prime number p.

We are looking for x such that:

$$n! / i \cdot x \equiv r \quad \text{(modulo p)}$$
$$x \equiv r \cdot i / n! \quad \text{(modulo p)}$$
$$x \equiv r \cdot i \cdot n!^{-1} \quad \text{(modulo p)}$$

Problem: Find a faulty factorial of length $n < p$, with a fault at position i, that gives reminder $r > 0$ when divided by prime number p.

We are looking for x such that:

$$n! / i \cdot x \equiv r \quad \text{(modulo } p\text{)}$$
$$x \equiv r \cdot i / n! \quad \text{(modulo } p\text{)}$$
$$x \equiv r \cdot i \cdot n!^{-1} \quad \text{(modulo } p\text{)}$$
$$x \equiv r \cdot i \cdot n!^{p-2} \quad \text{(modulo } p\text{)}$$

Compute x, and check whether $x < i$.

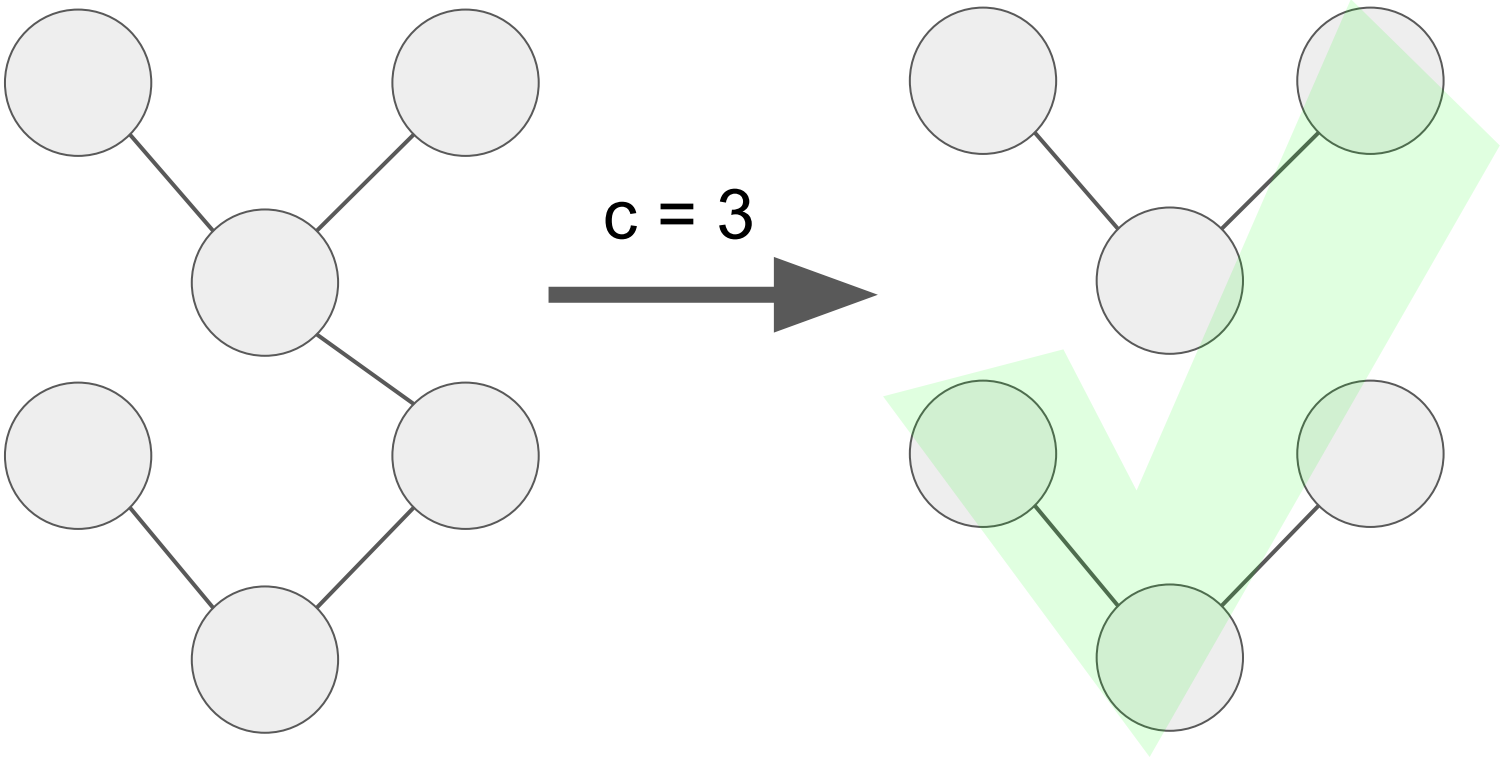
Problem J
Justified Jungle

Submits: 203
Accepted: at least 17

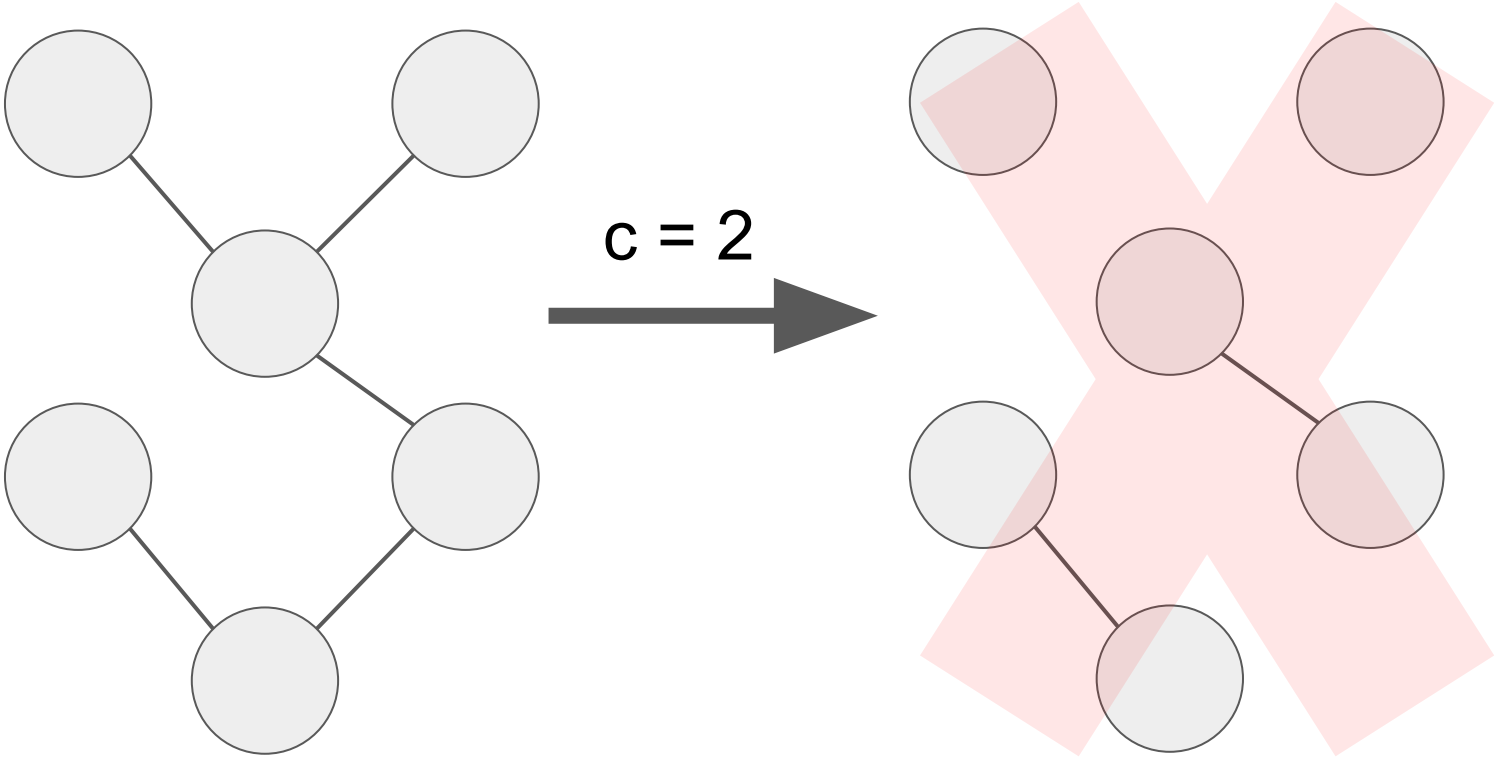
First solved by: Jagiellonian 1
Jagiellonian University in Krakow
(Hlembotskyi, Stokowacki, Zieliński)
00:16:32

Author: Luka Kalinovčić, Ivan Katanić

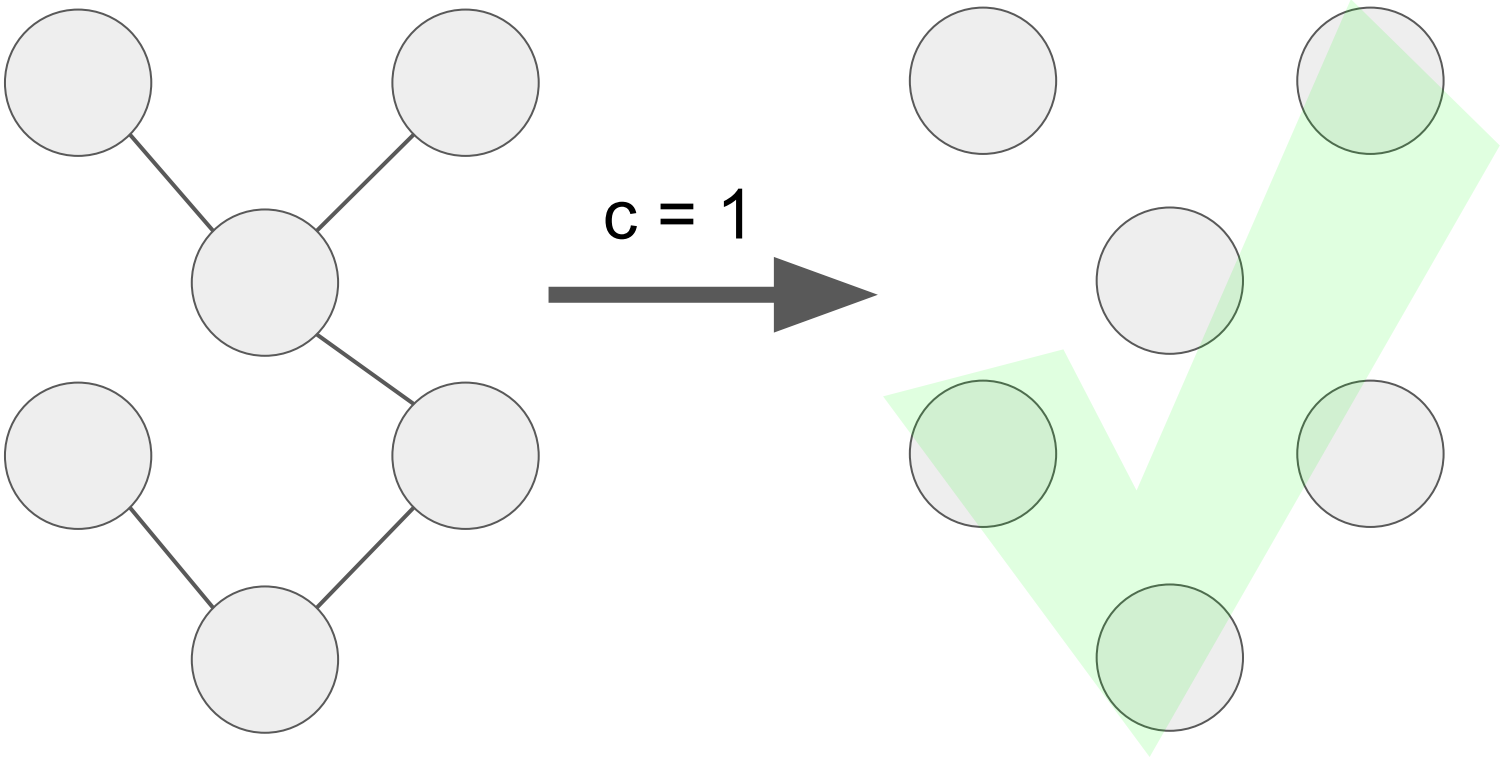
Problem: Given a tree, find all integers c, such that we can cut a tree into components of size c.



Problem: Given a tree, find all integers c, such that we can cut a tree into components of size c.



Problem: Given a tree, find all integers c, such that we can cut a tree into components of size c.



Problem: Given a tree, find all integers c, such that we can cut a tree into components of size c.

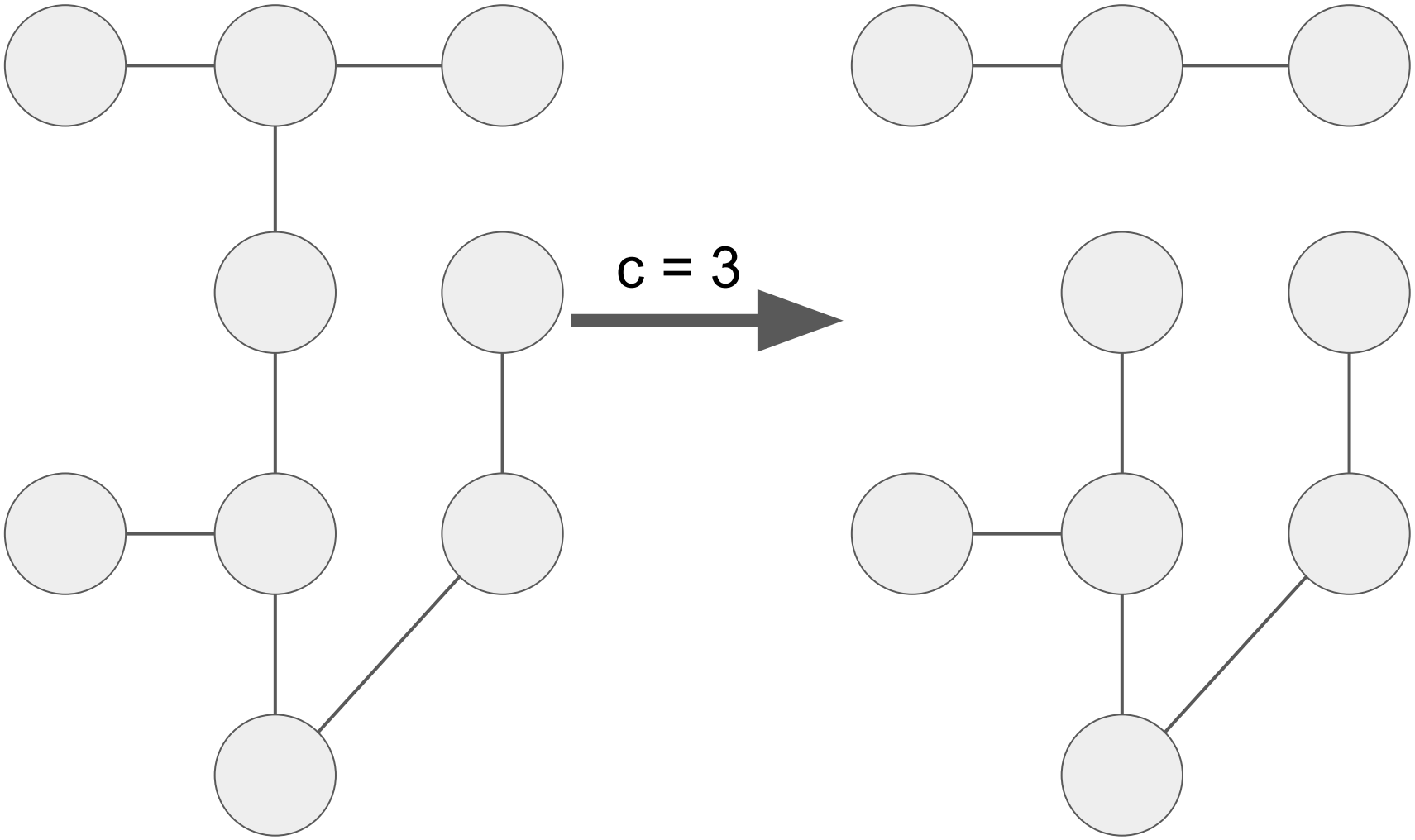
The tree size needs to be divisible by c.
There aren't that many divisors: worst case 240 for n=720720.
We can try each divisor separately.

Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?

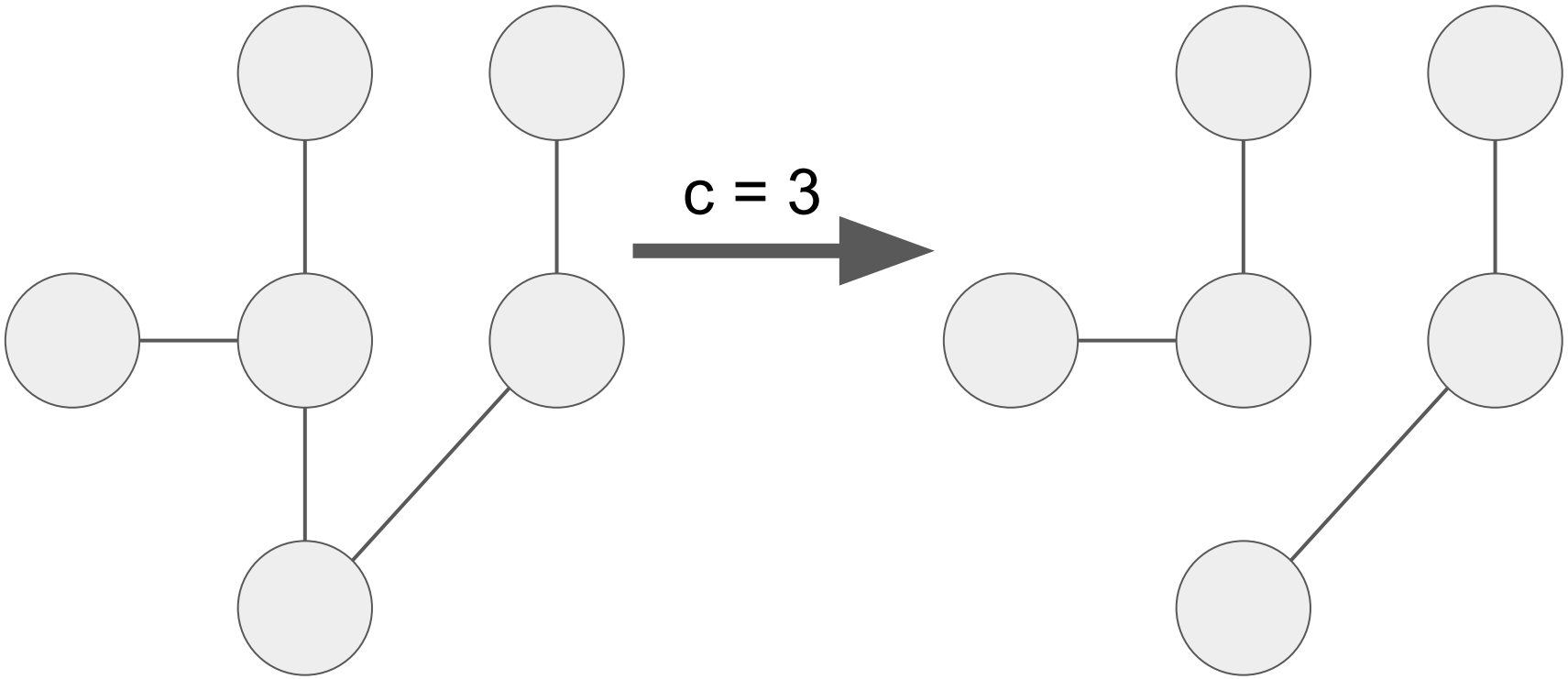
Iterative algorithm:
If n = c: done.
Otherwise:

- Find an edge that divides the tree into subtrees of sizes c and n − c.
- If there is no such edge: impossible.
- Otherwise: Cut the edge and repeat the algorithm on the subtree of size n − c.

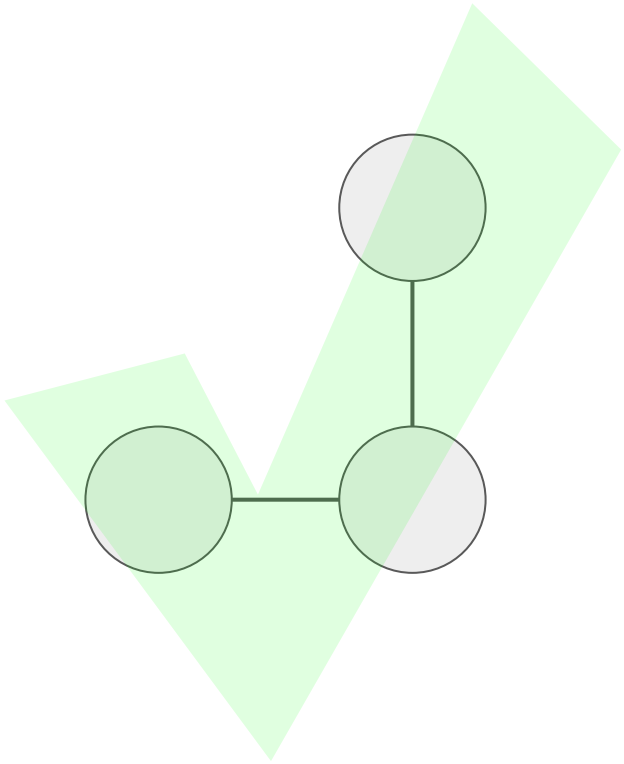
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



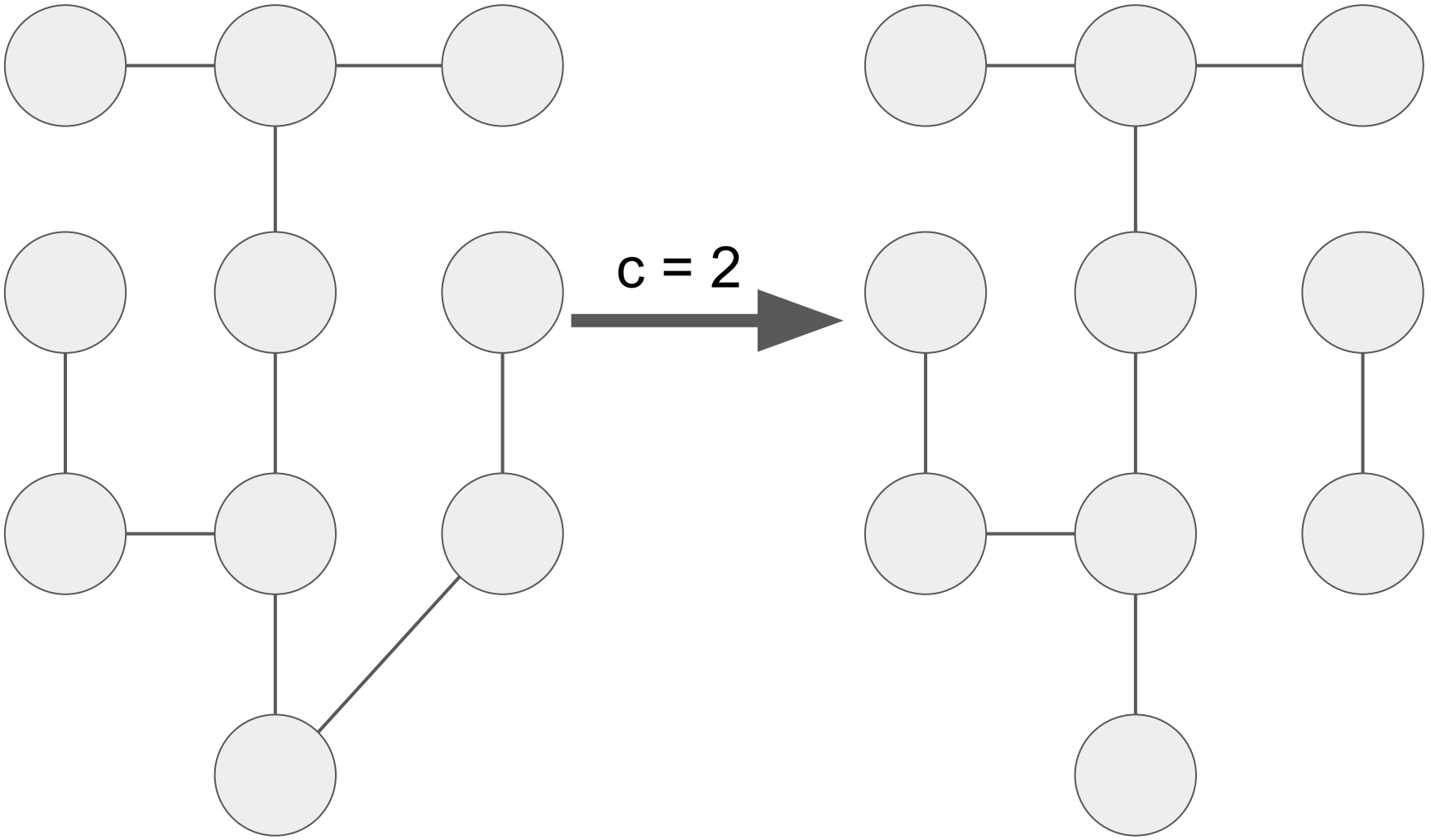
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



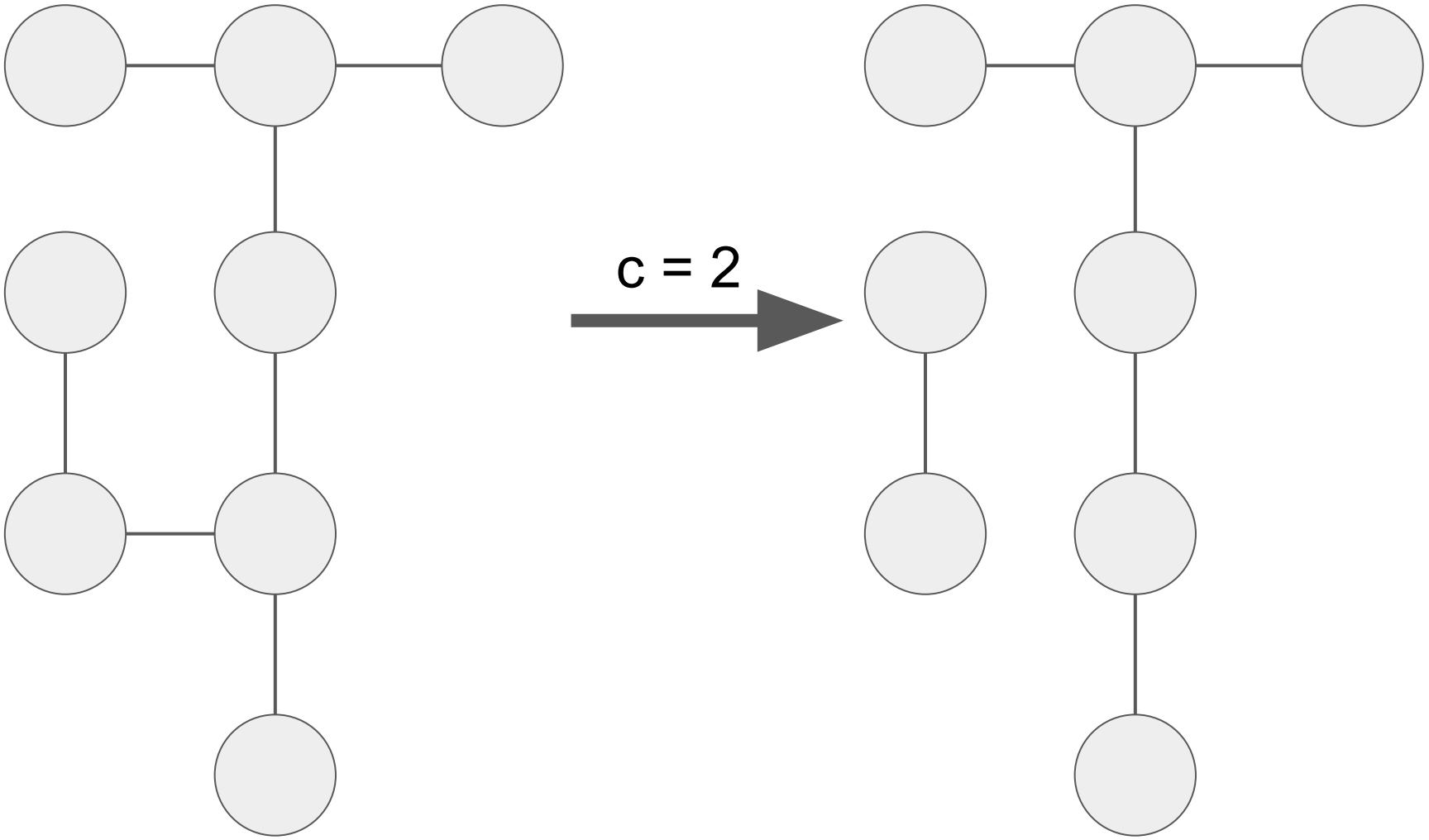
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



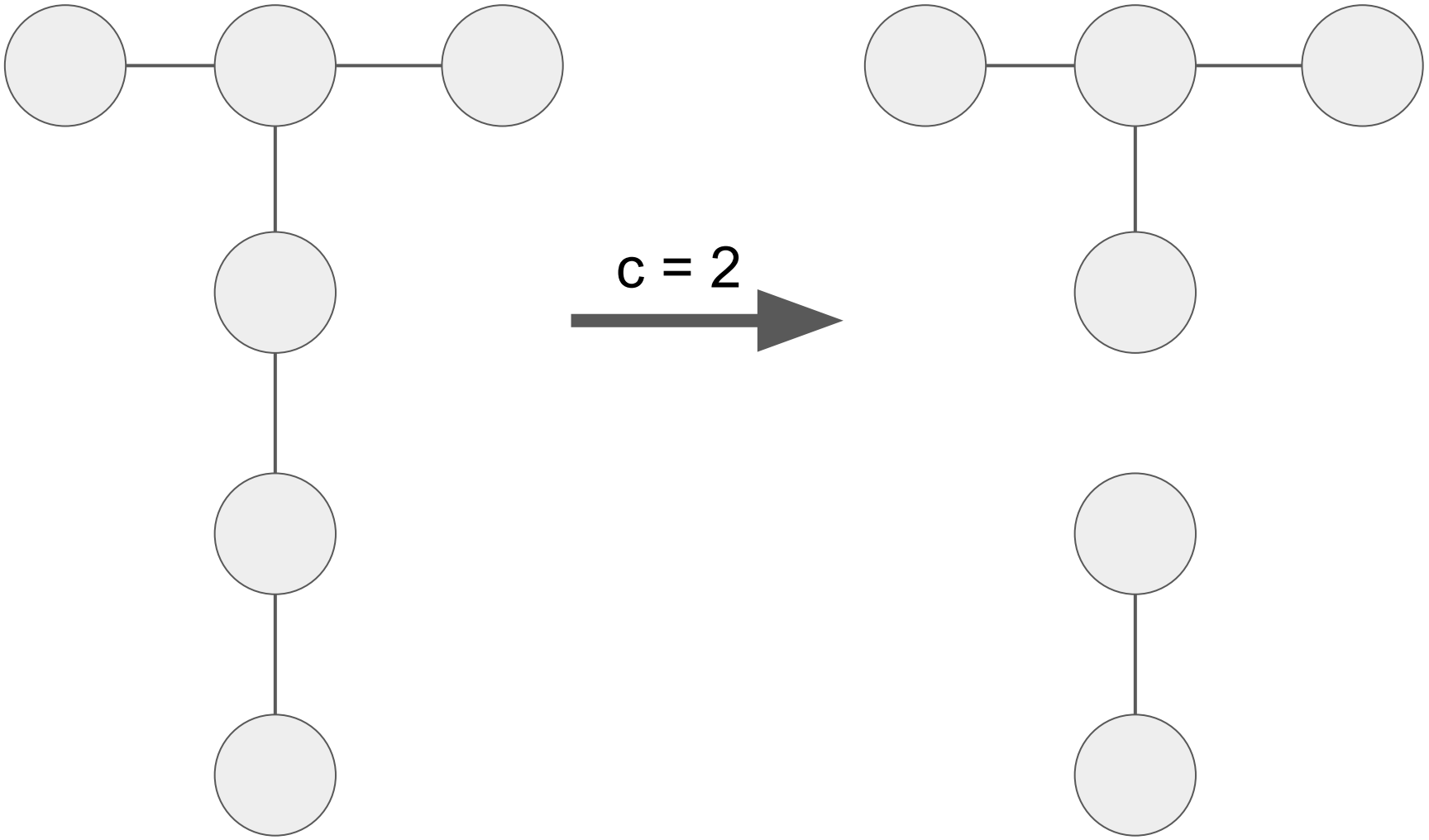
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



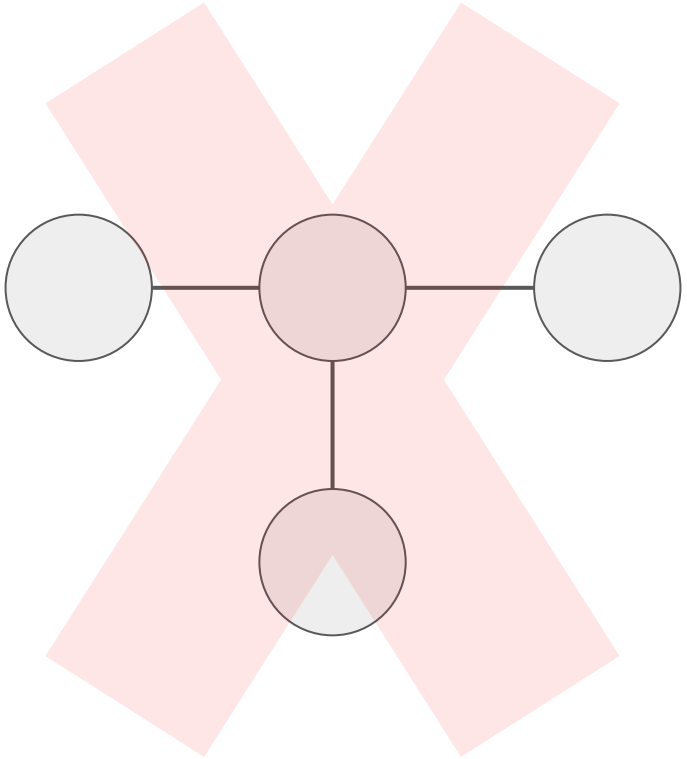
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



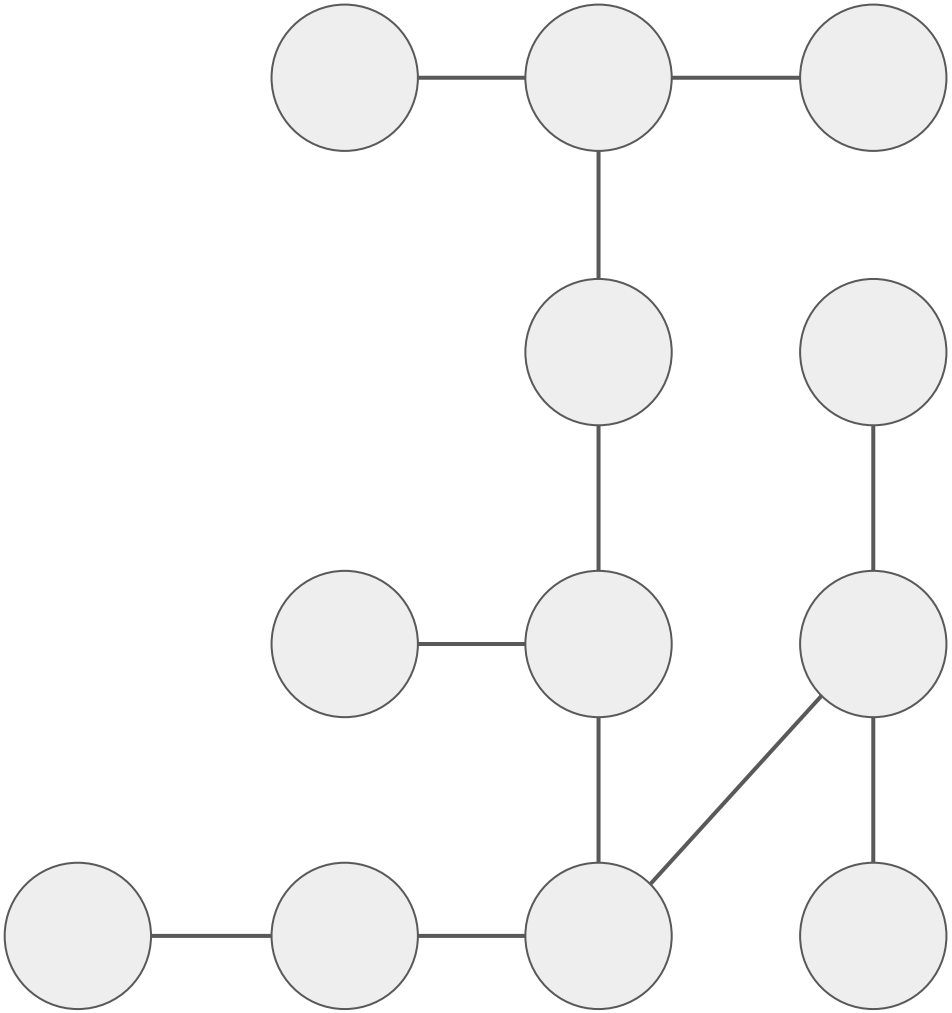
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?

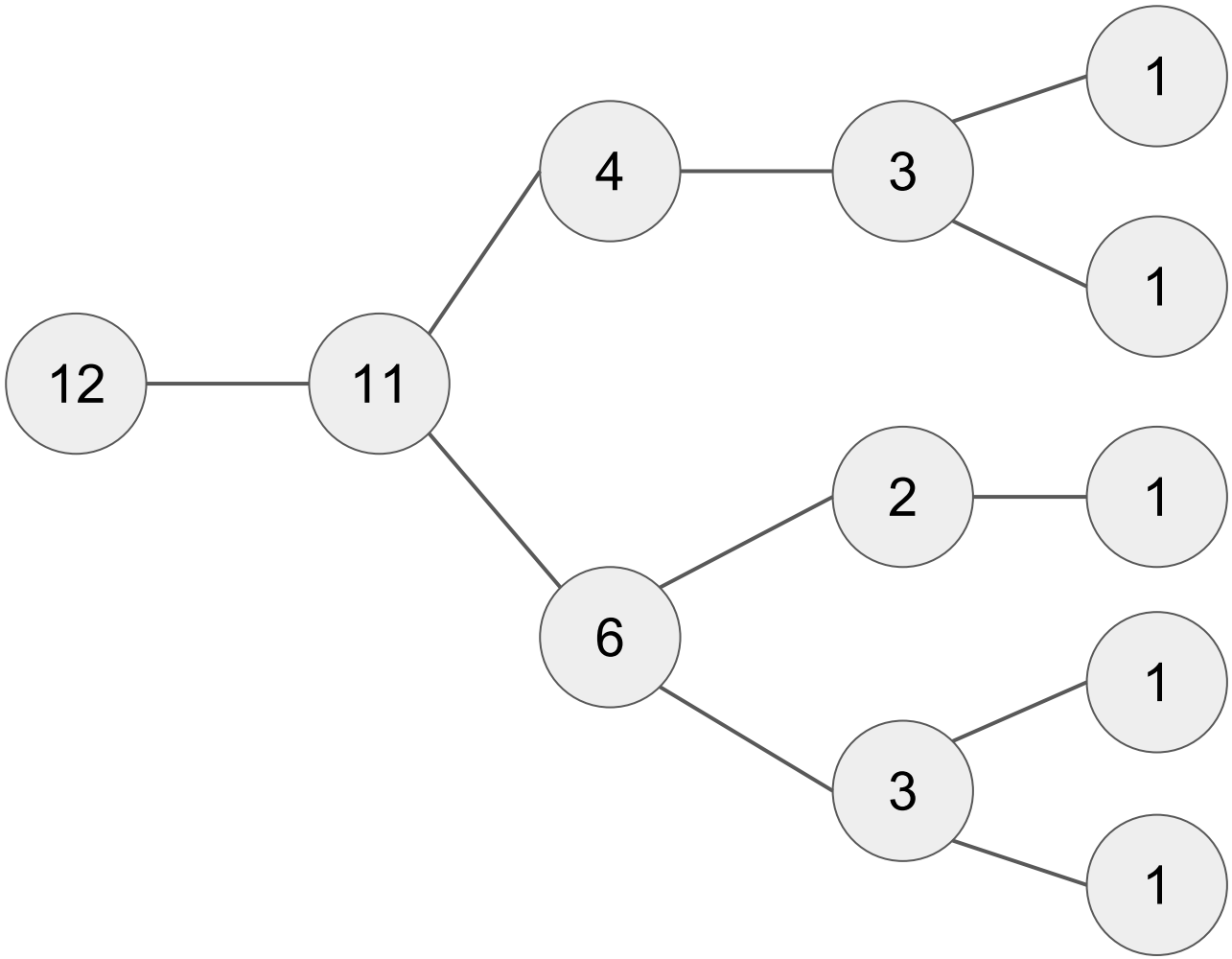
Iterative algorithm is difficult to implement in O(n), and might time out.

Simplified algorithm:
Root the tree and compute the size of each subtree (only once, no need to repeat for each divisor).

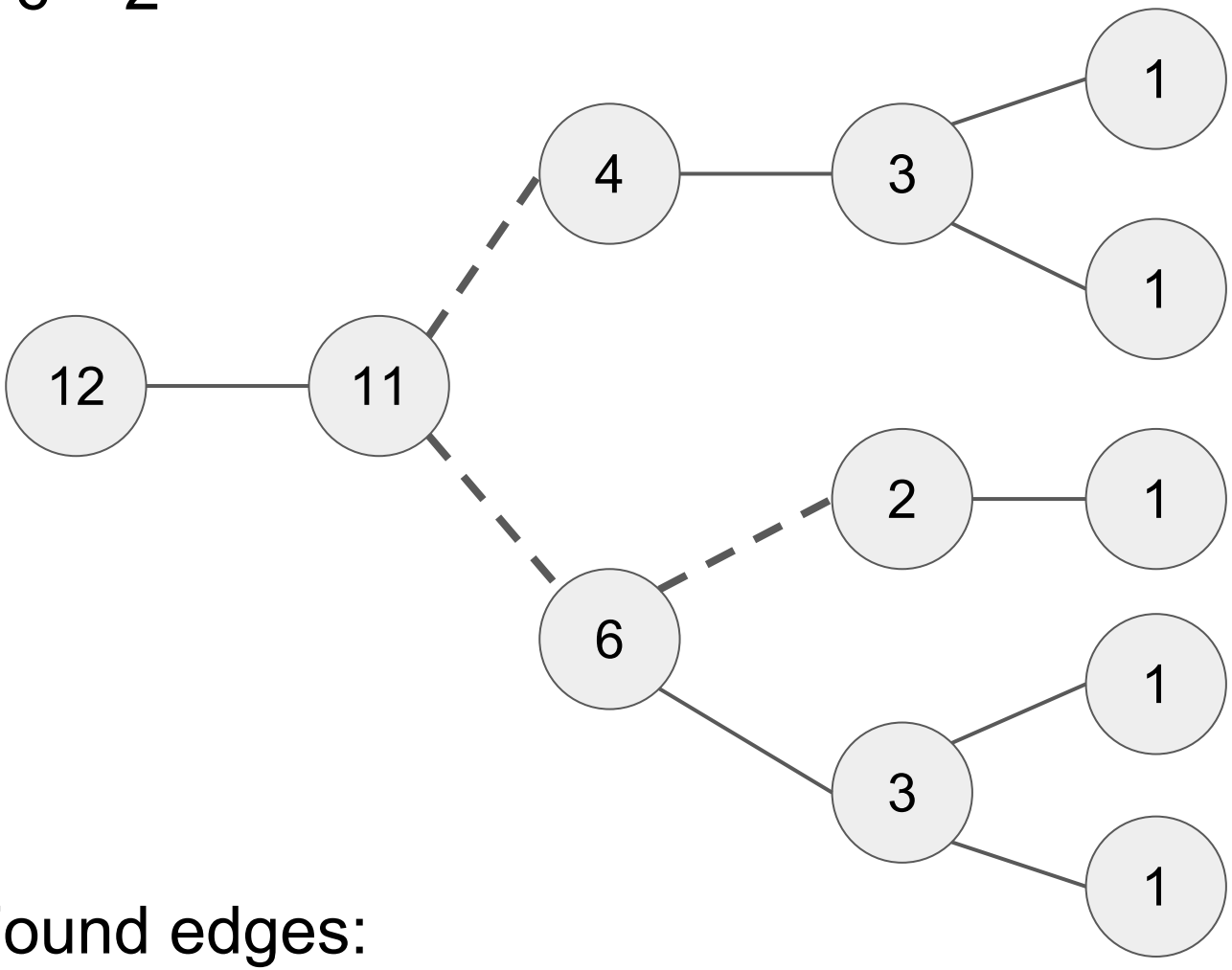
Find edges with subtrees sizes equal to a multiple of c. Those are the ones we'll end up cutting.

If the number of found edges is equal to n / c − 1: yes!
Otherwise: no!




12
11
4
3
1
1
6
2
1
3
1
1

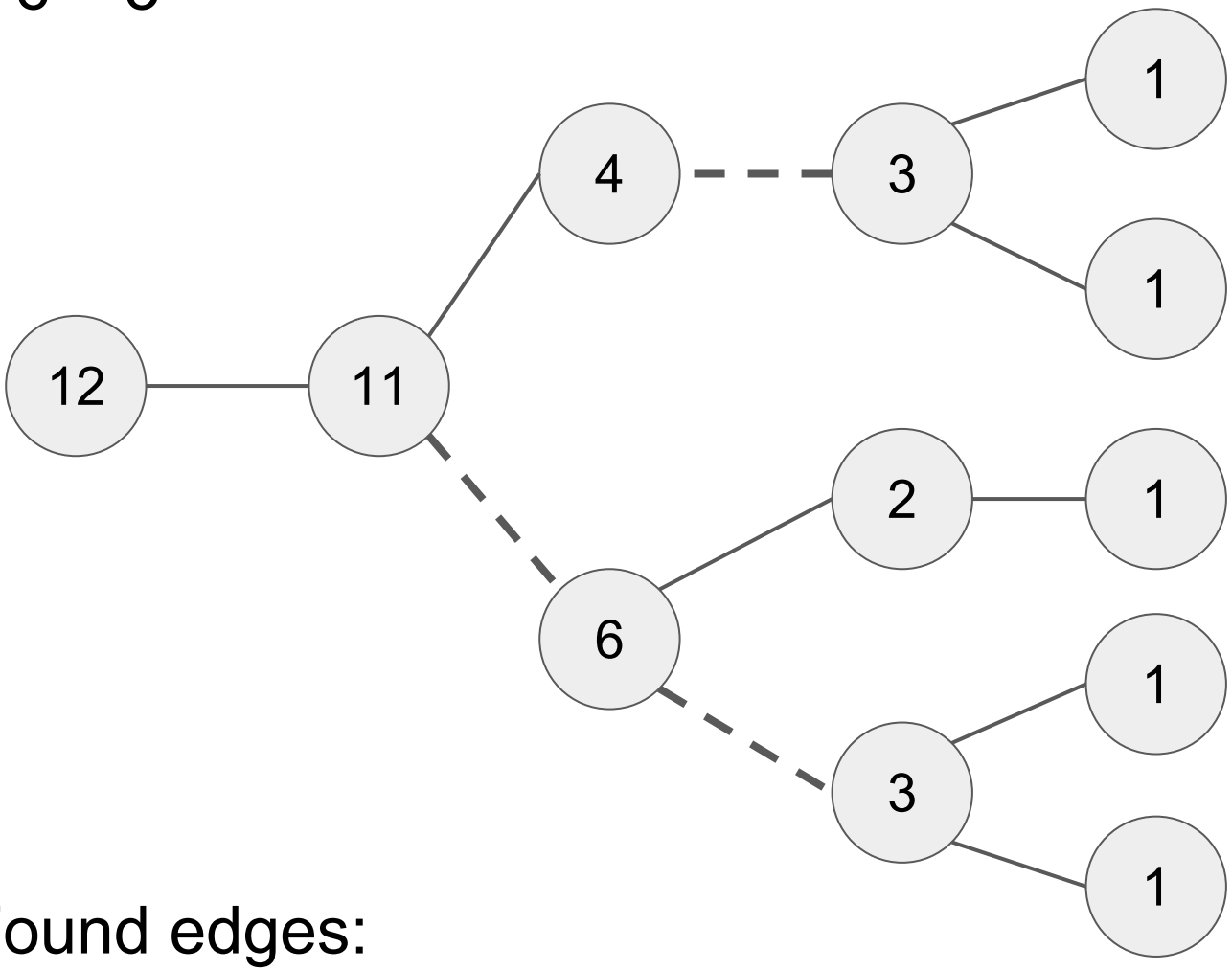
c = 2



Found edges:

$3 \neq n / c - 1 \rightarrow$ NO

c = 3



Found edges:

3 = n / c − 1 → YES

Overall complexity: $O(n \cdot \sigma(n))$, where $\sigma(n)$ is the number of divisors of n.

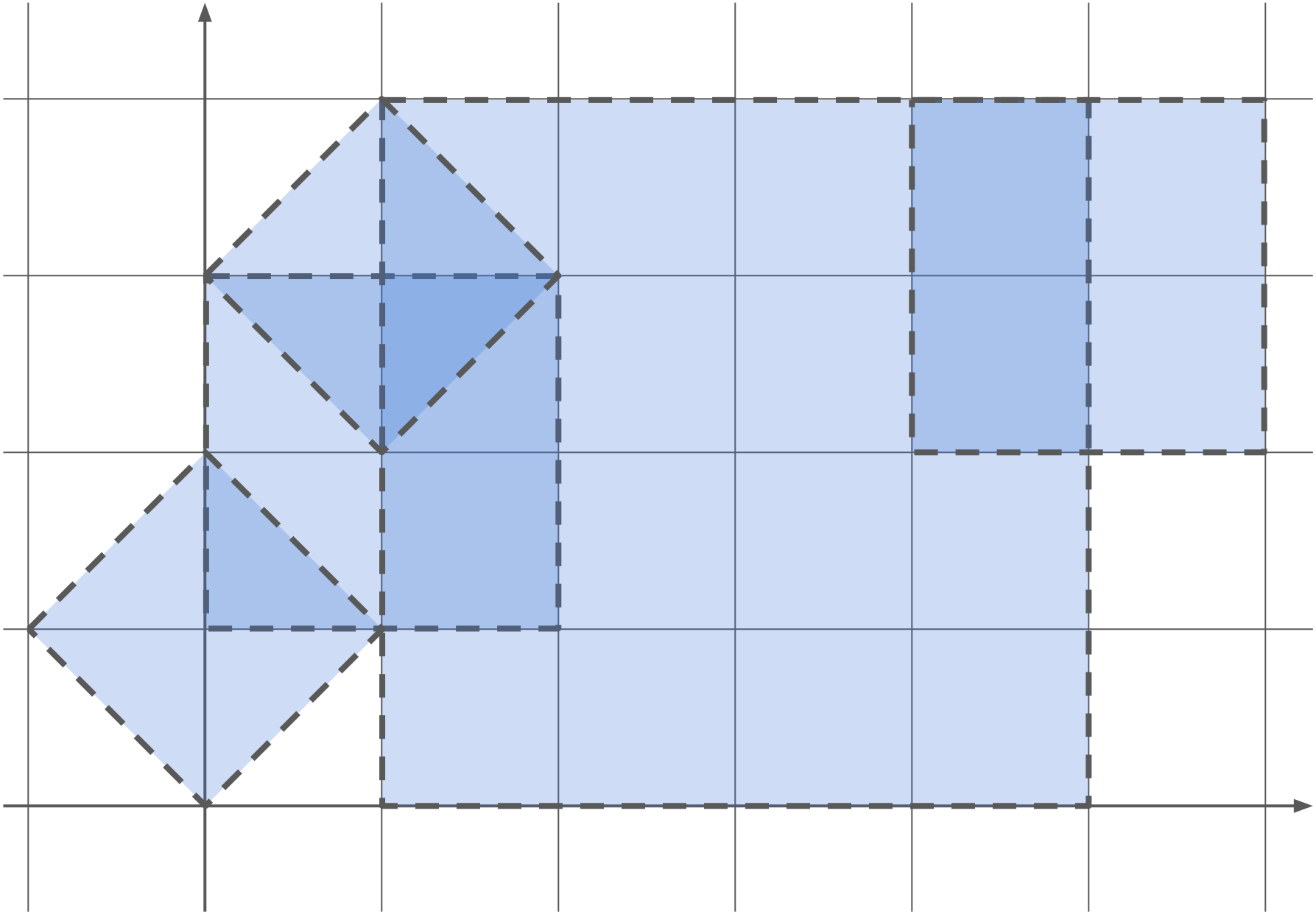
$O(n + \sigma(n)^2)$ is possible with an extra insight.

Problem L
Lunar Landscape

Submits: 41
Accepted: at least 5

First solved by: UW2
University of Warsaw
(Boguta, Czajka, Farbiś)
02:02:13

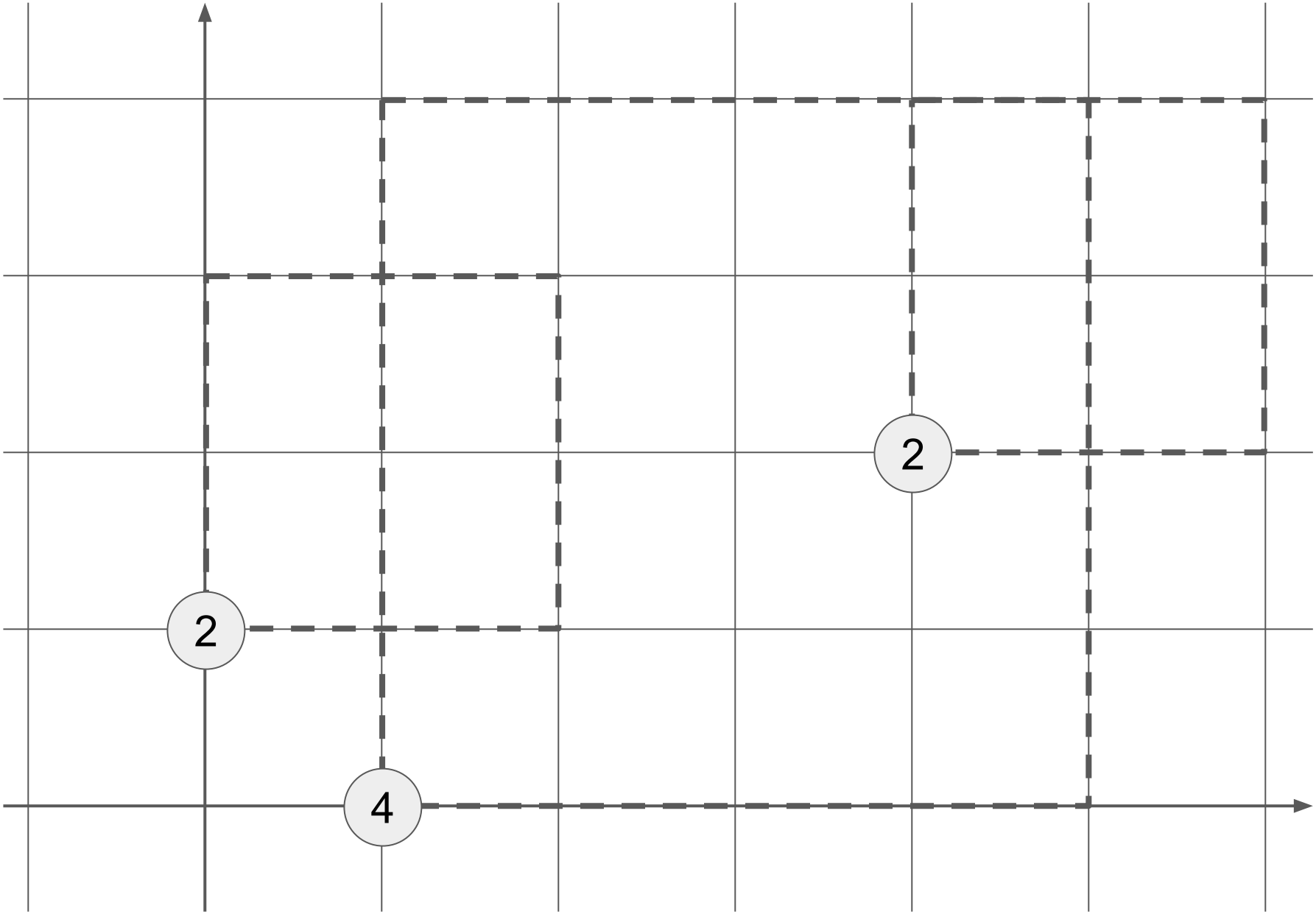
Author: Luka Kalinovčić

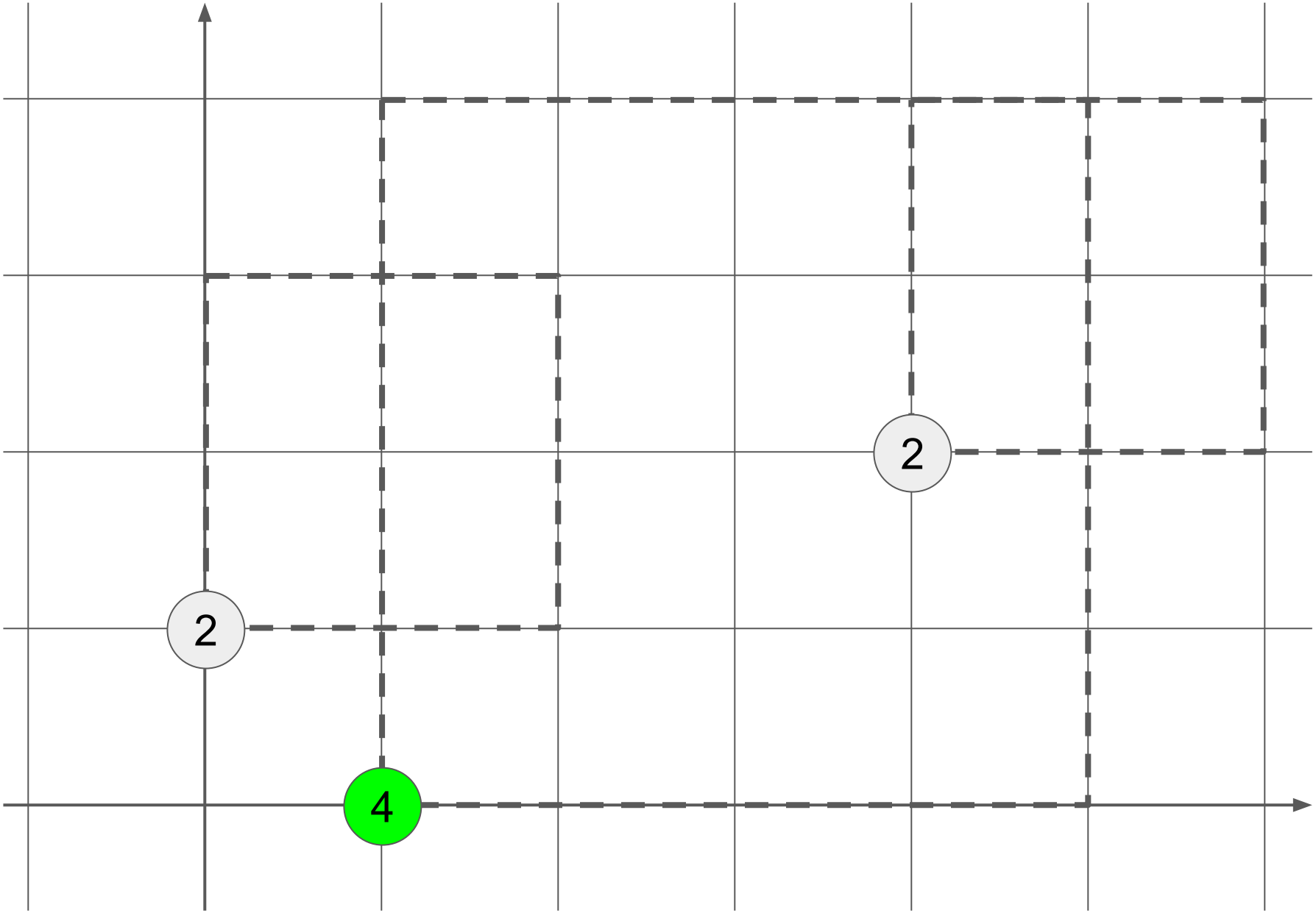


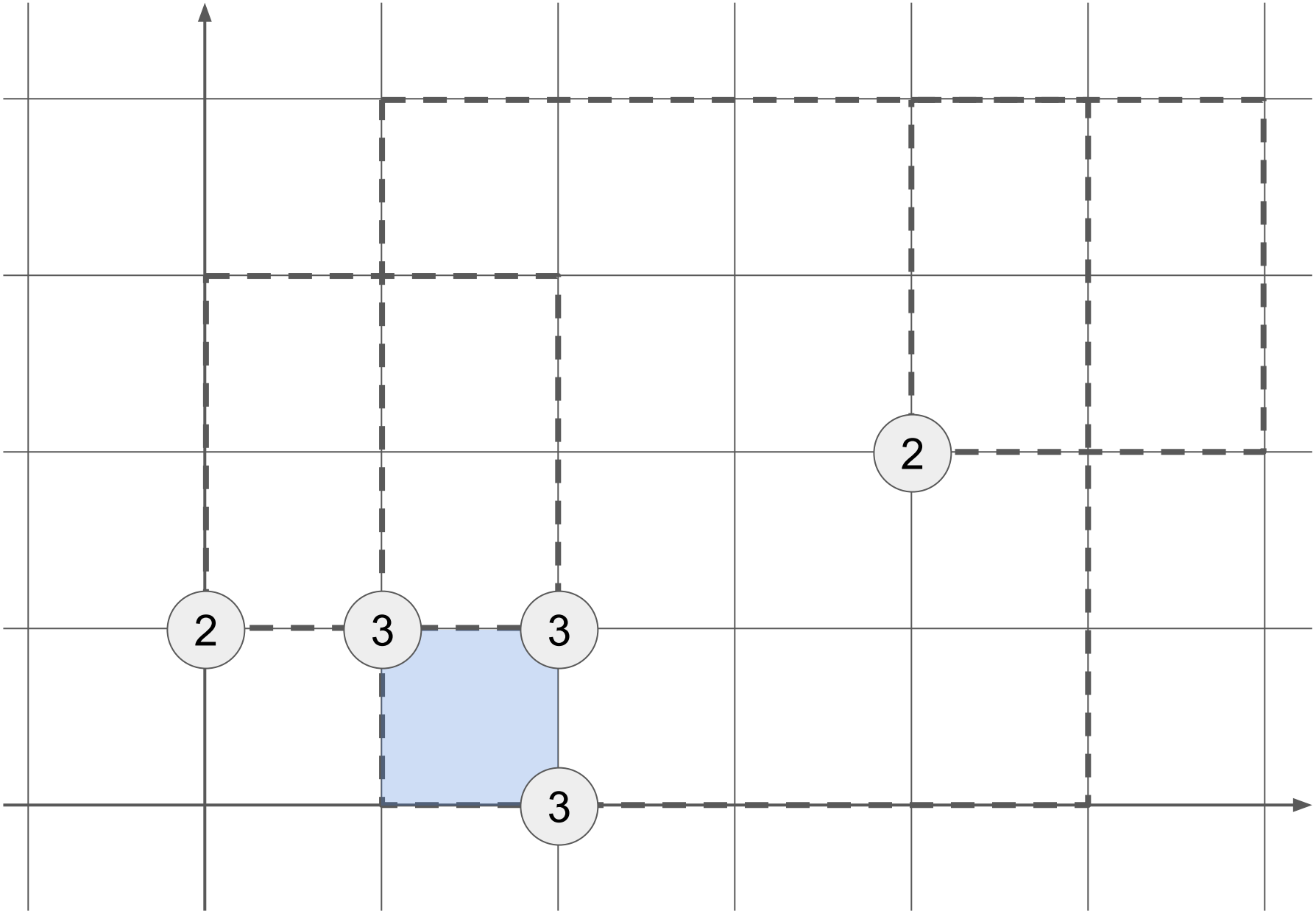
Key observation: The grid is small enough to iterate over each unit square and “paint it blue” in memory.

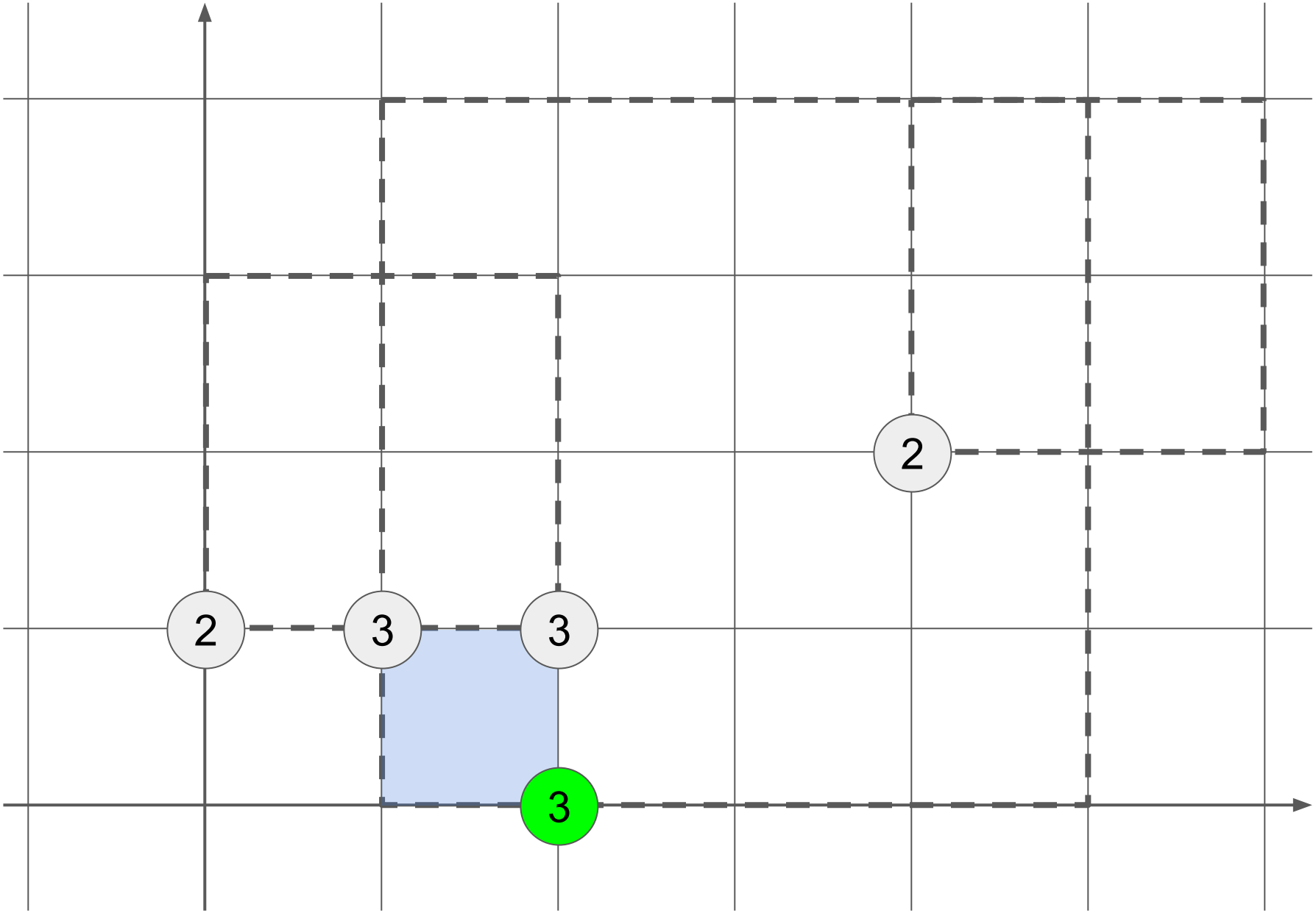
However, the naive algorithm that iterates through each unit square of each frame is too slow.

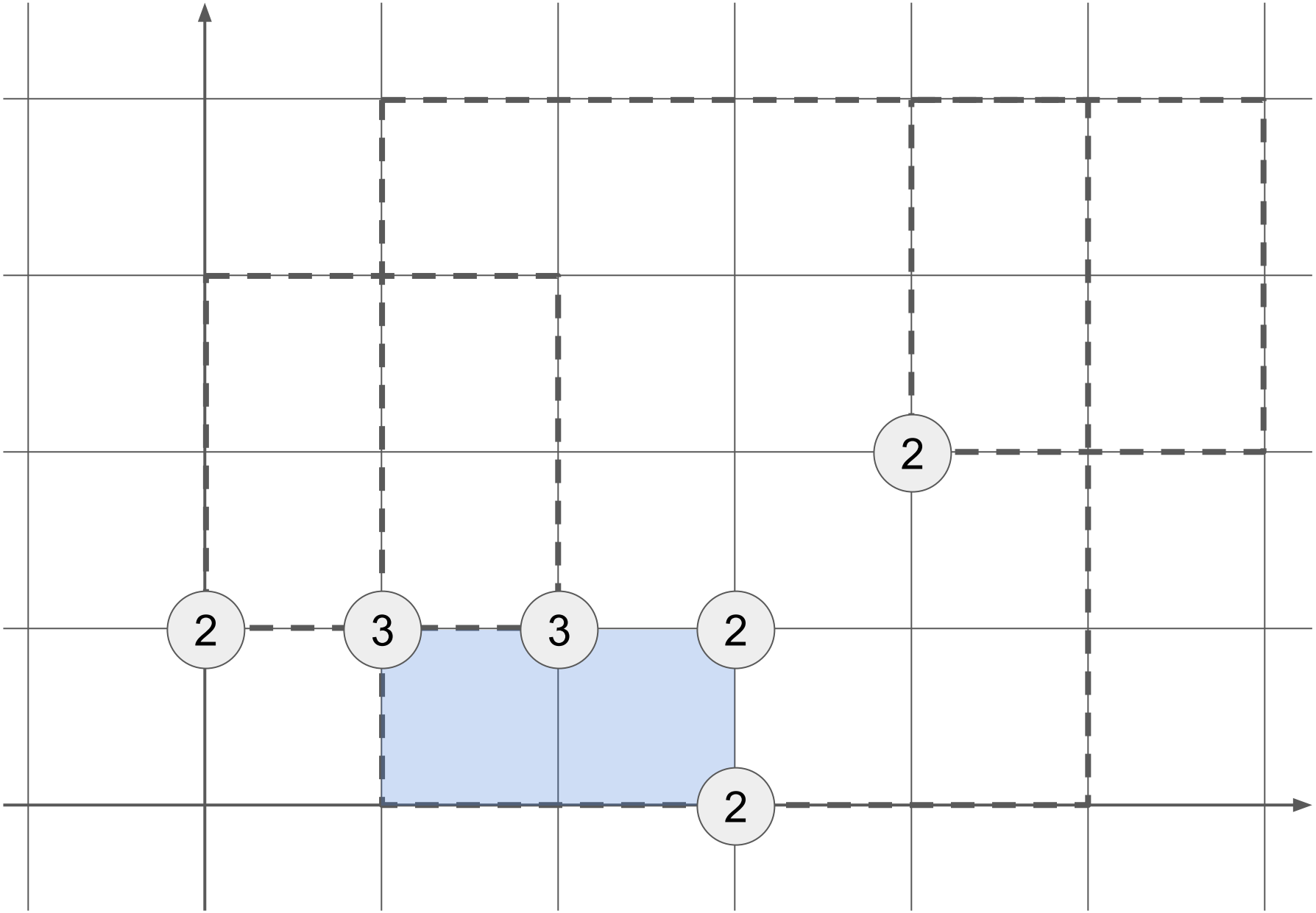
Instead, let’s first place a “bucket full of paint” in a corner of each frame.
Then, we’ll sweep across the grid and whenever we encounter a bucket, we’ll paint one unit square and propagate the bucket to neighbouring squares.

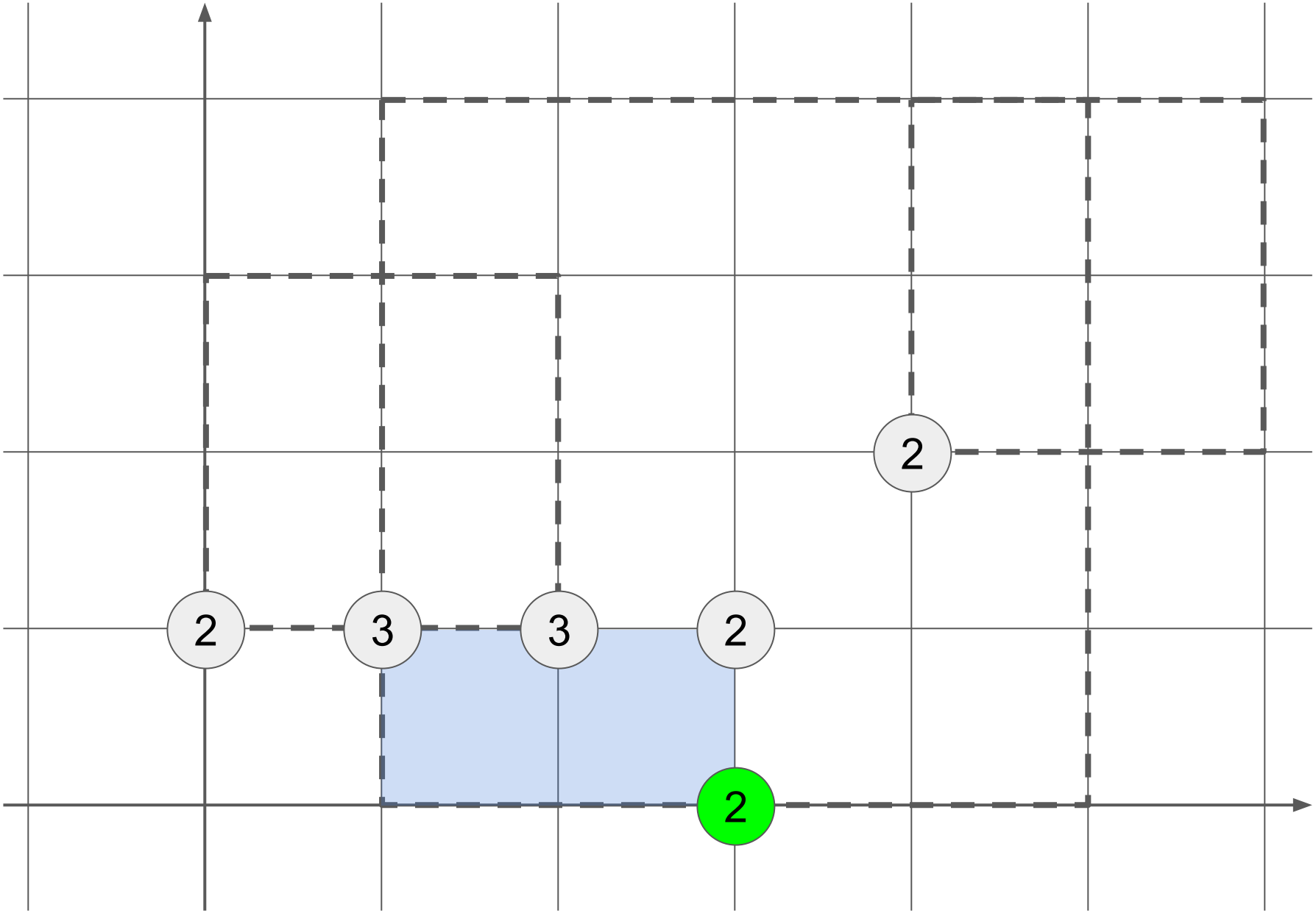

2
2
4

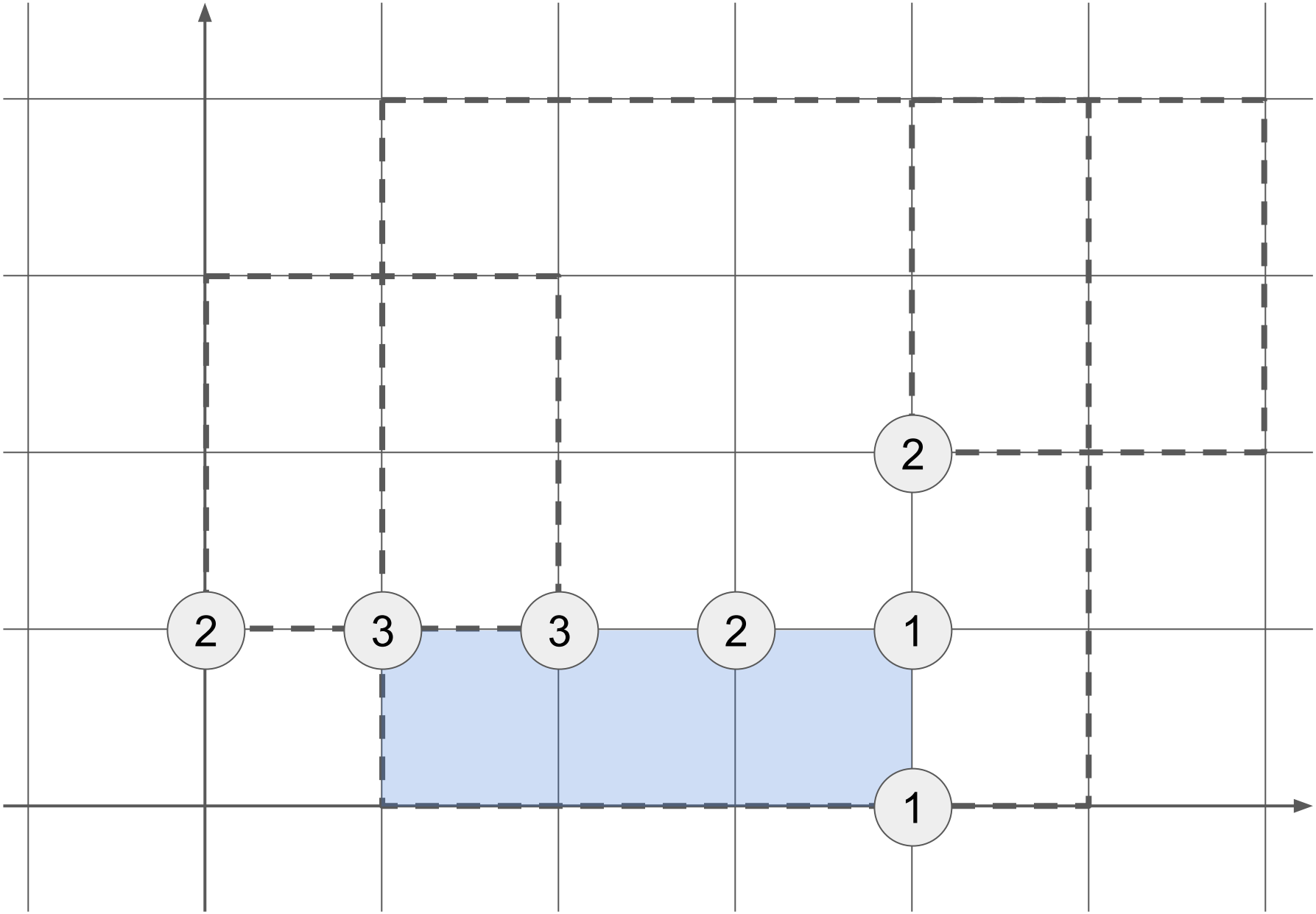

2
2
4

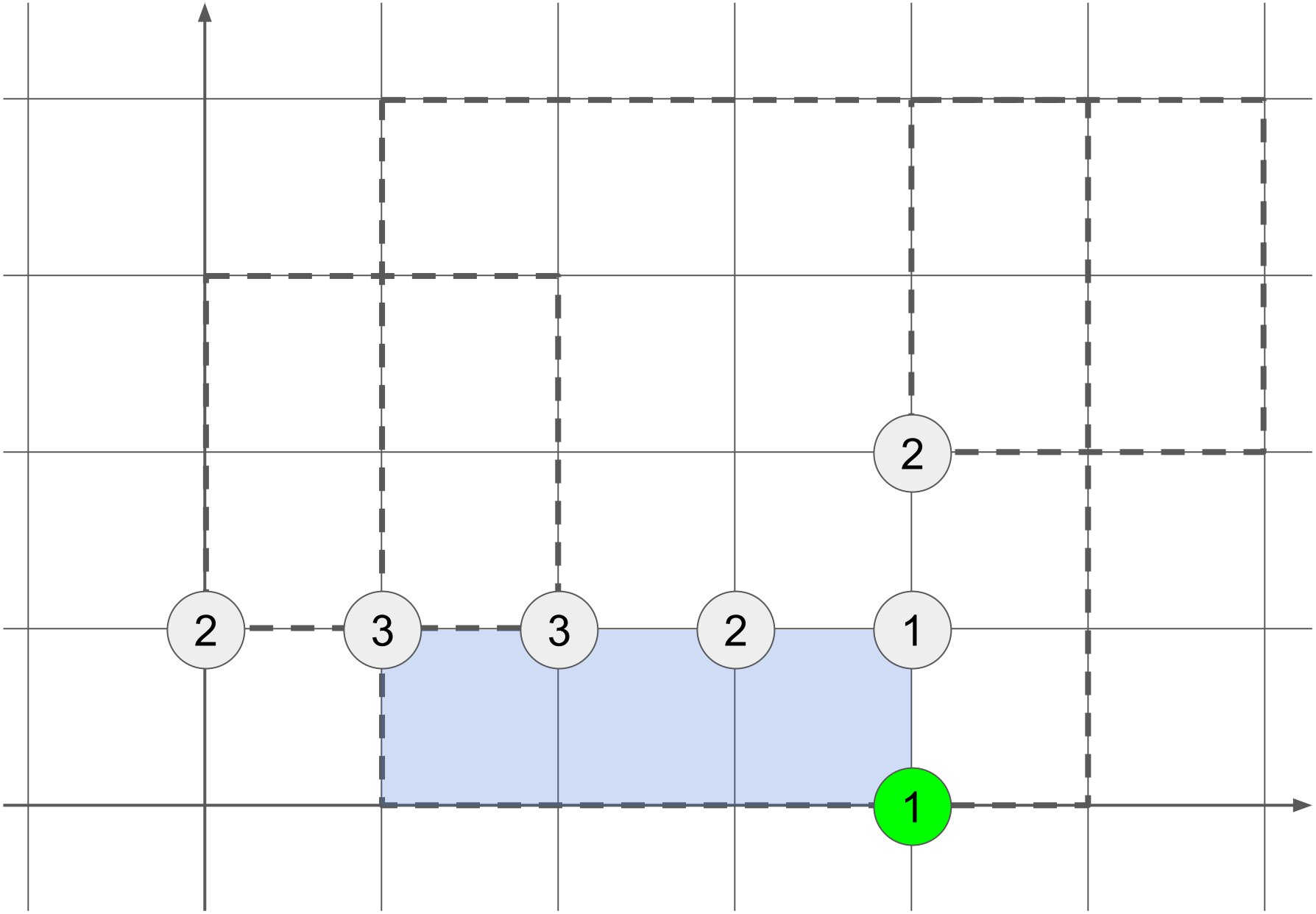

2
2
3
3
3

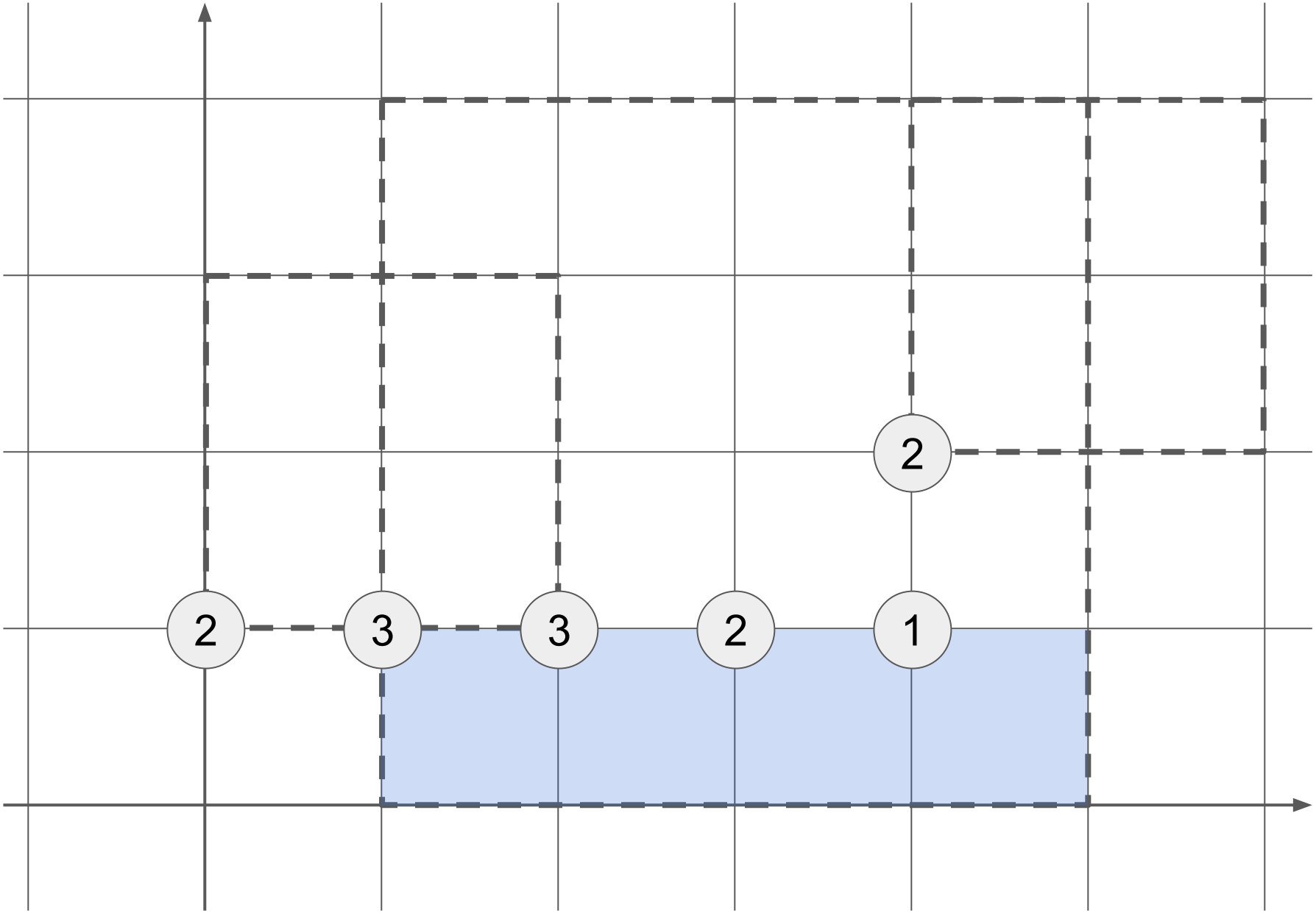

2
2
3
3
3

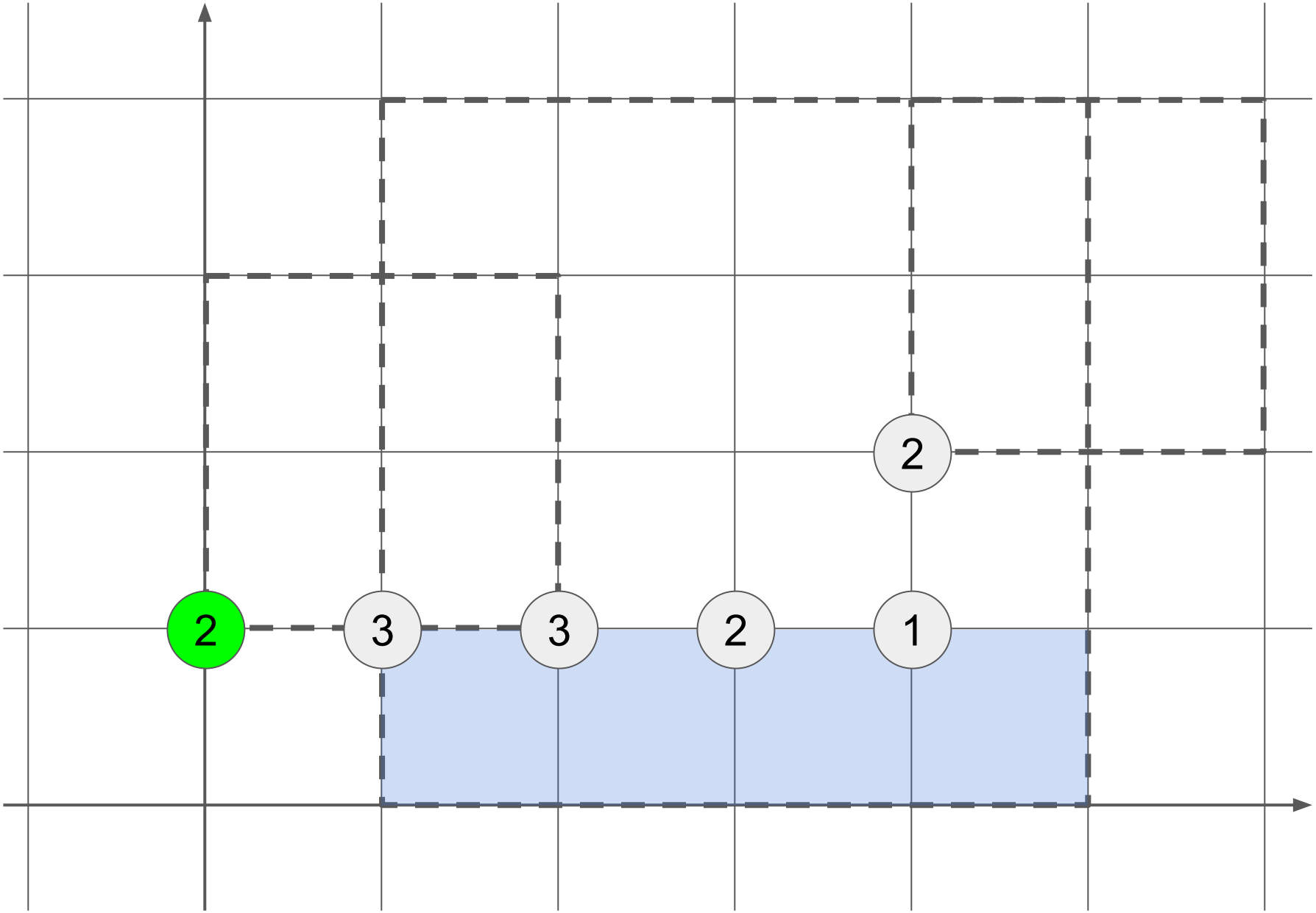

2
2
3
3
2
2

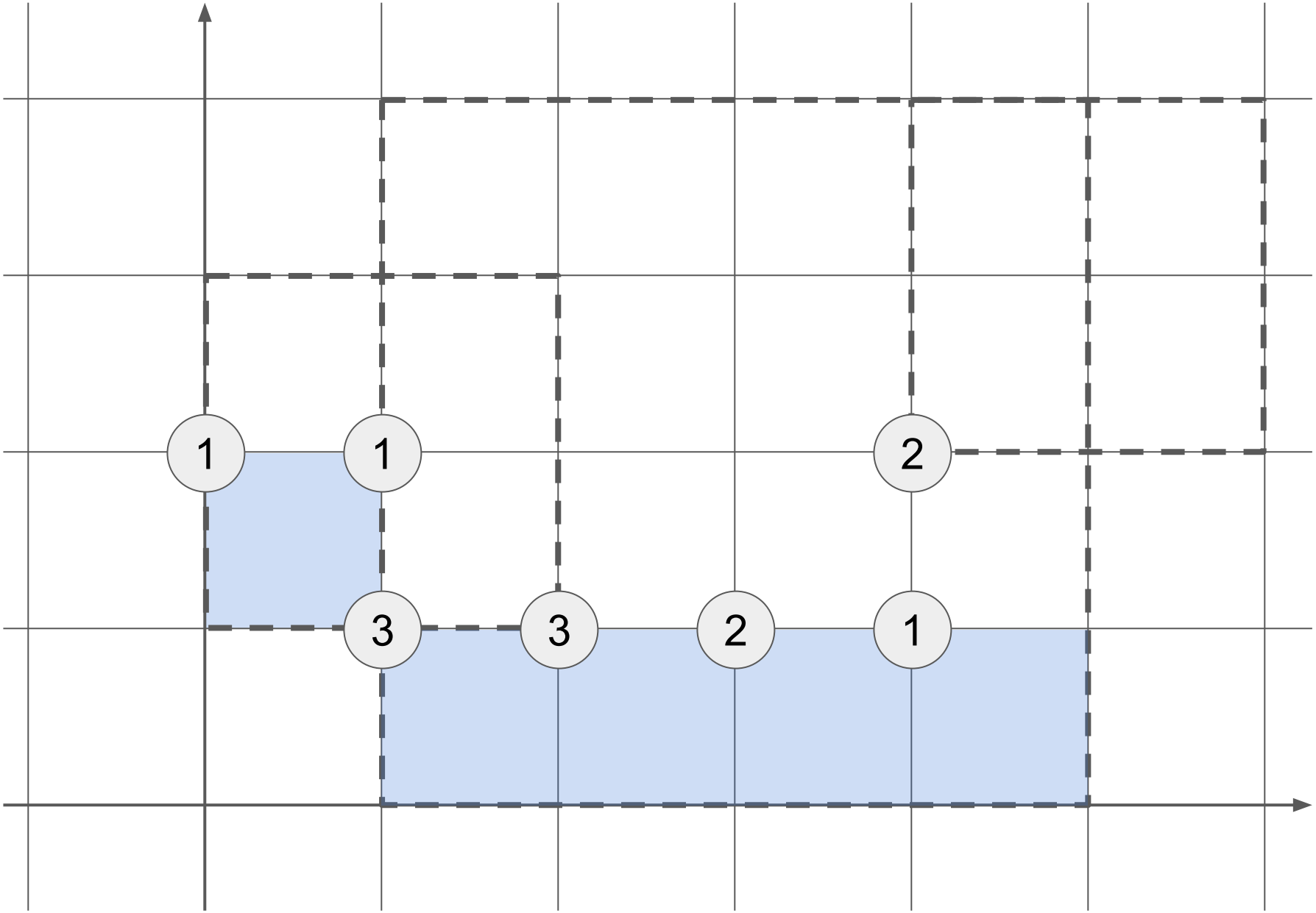

2
2
3
3
2
2

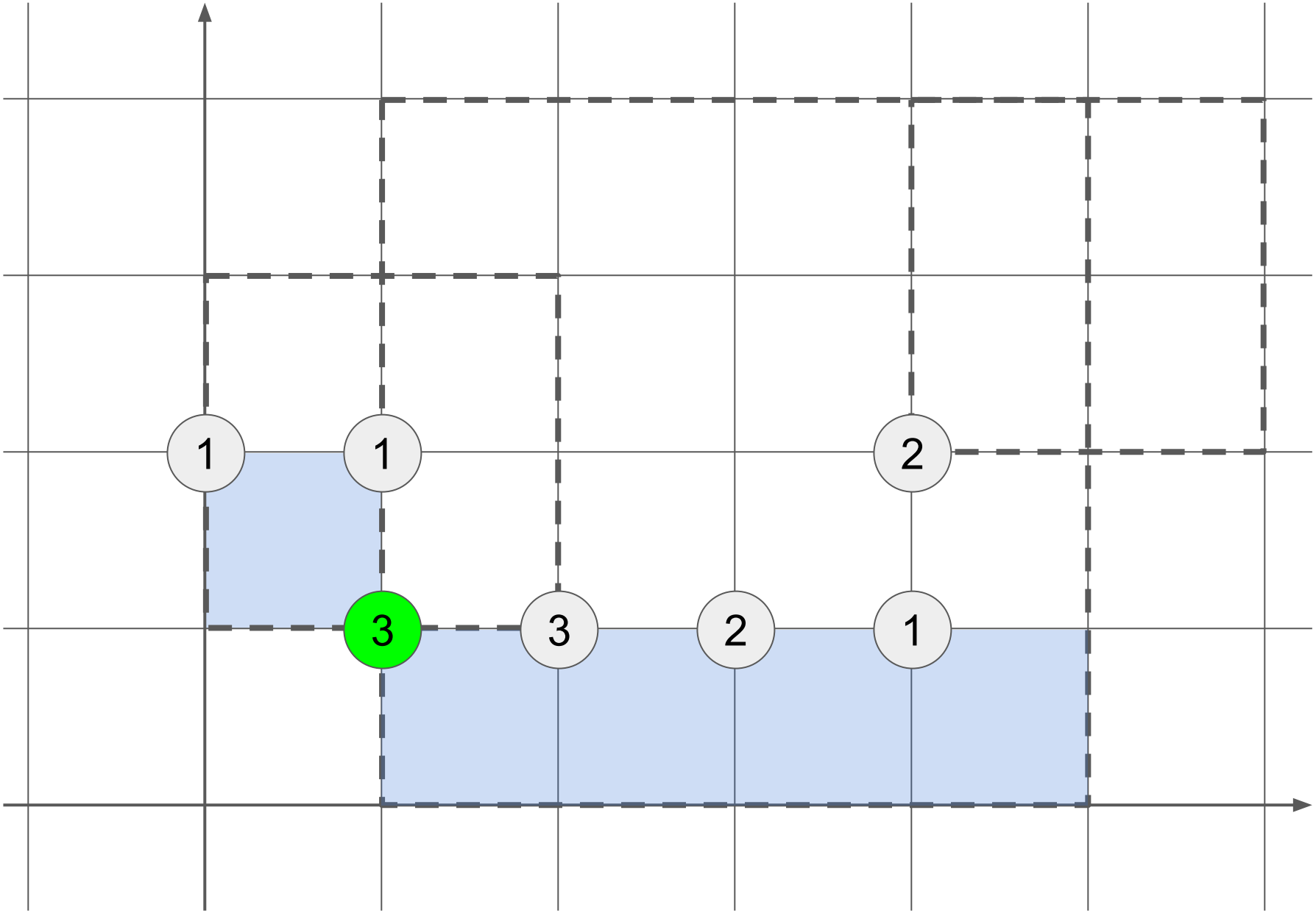

2
2
3
3
2
1
1

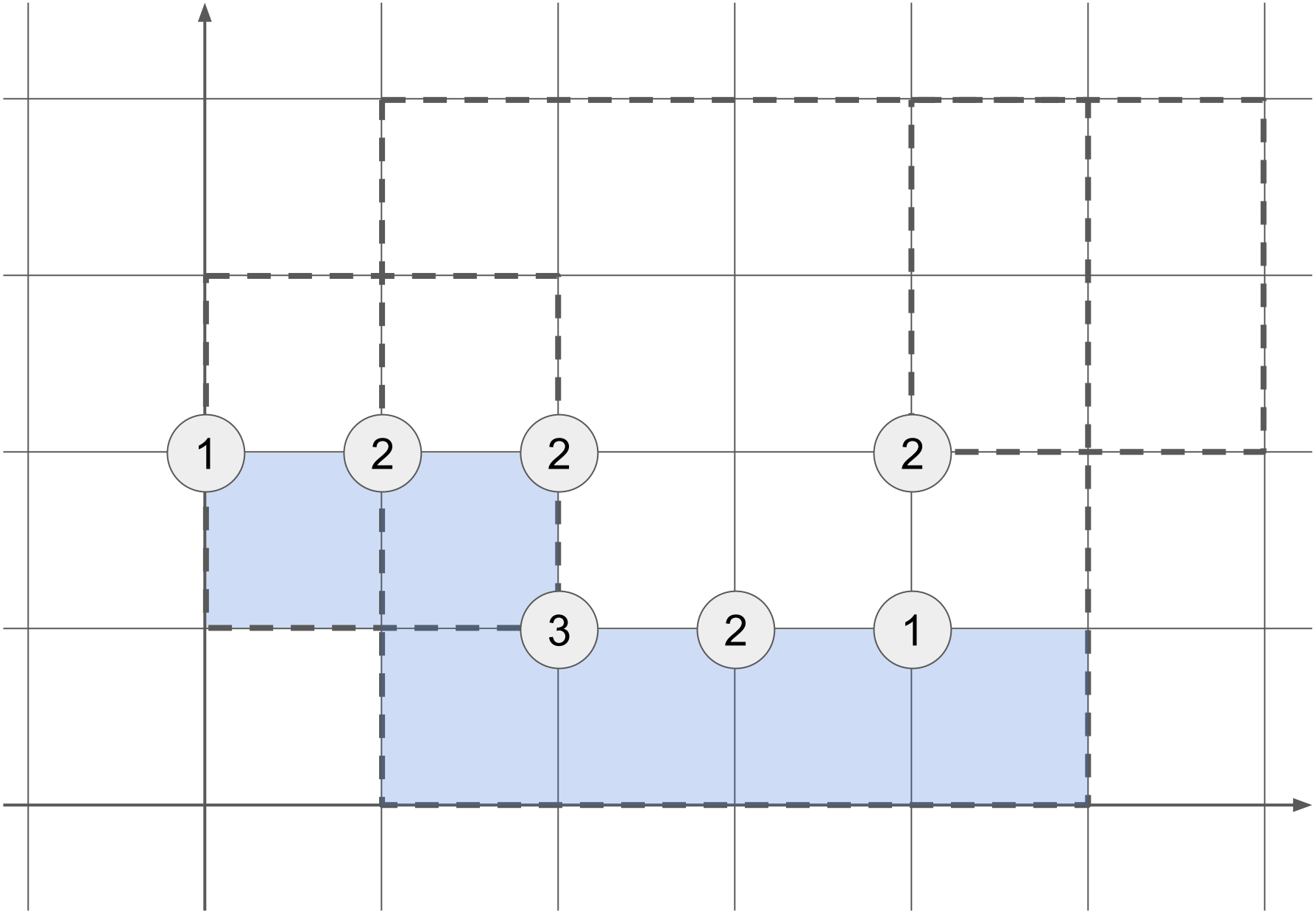

2
2
3
3
2
1
1

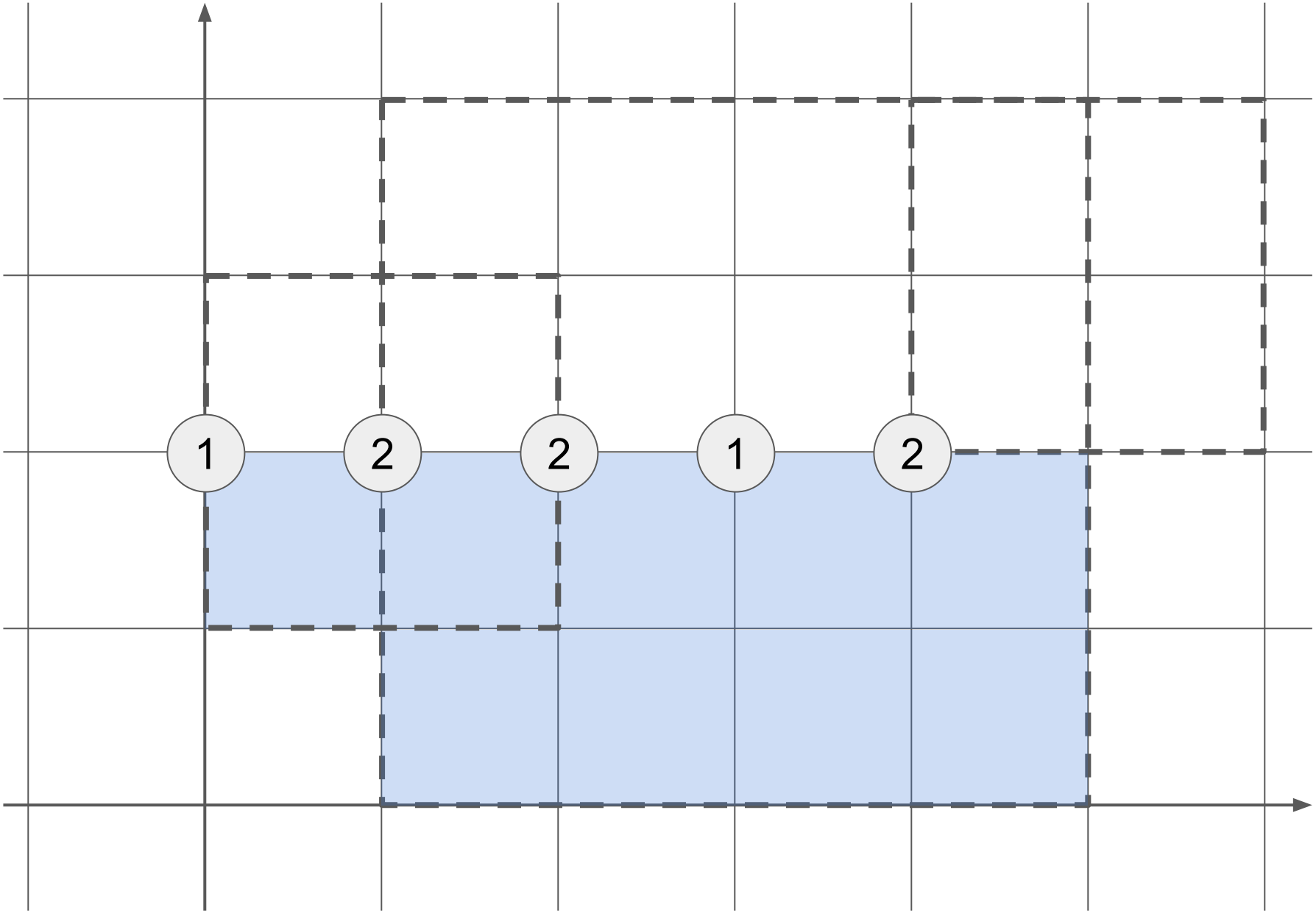

2
2
3
3
2
1

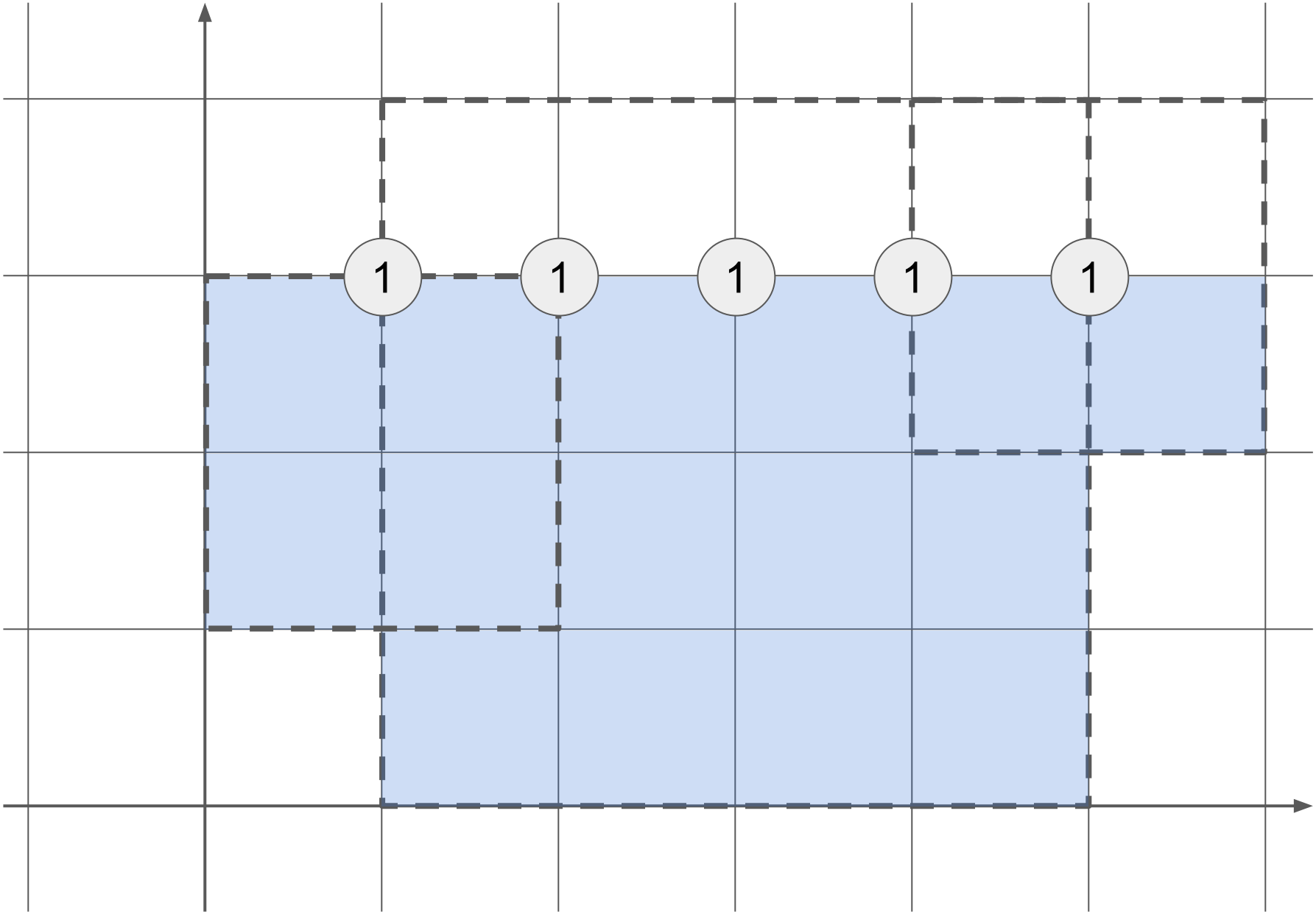

2
3
3
2
1
2

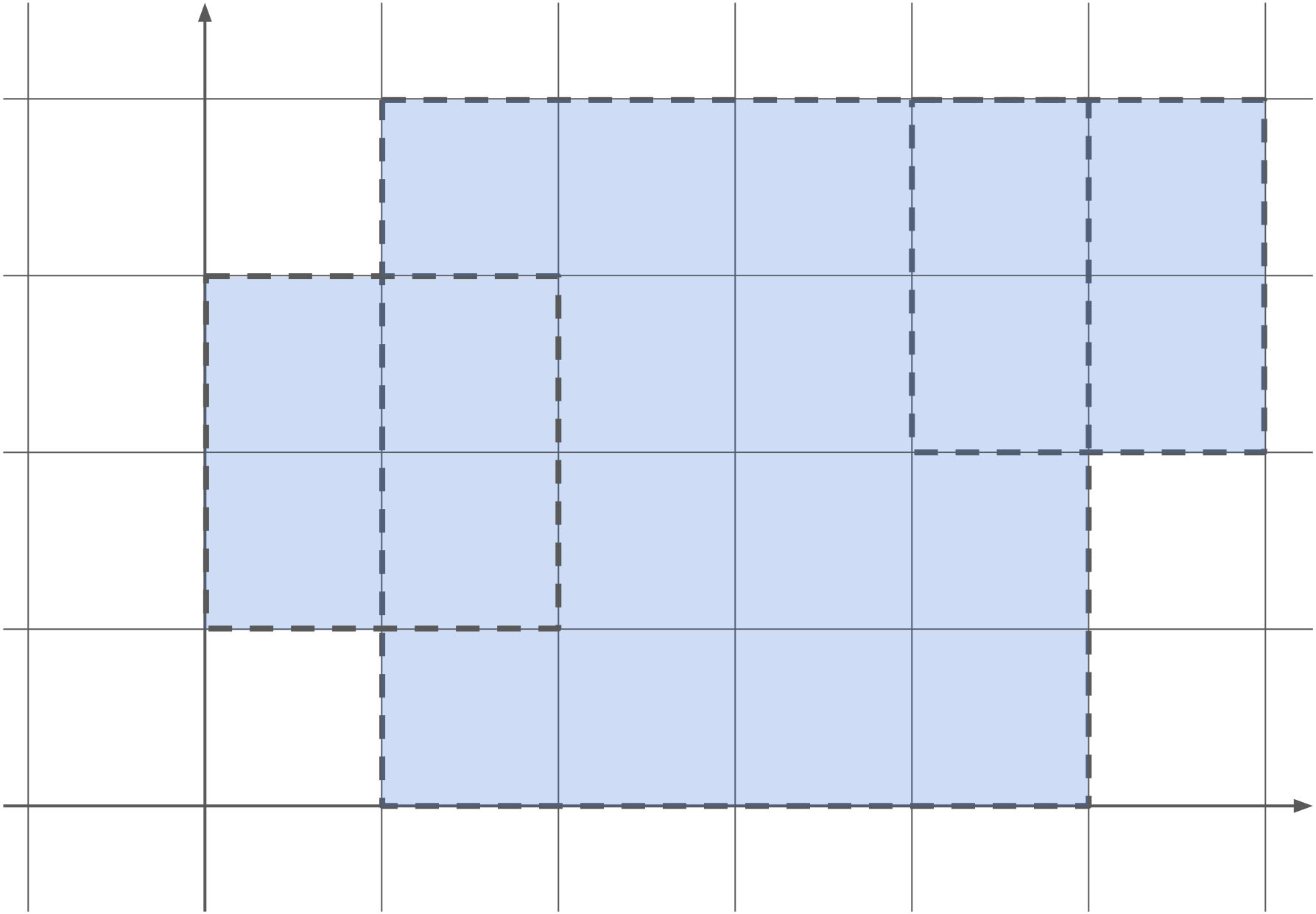

1
1
2
3
3
2
1

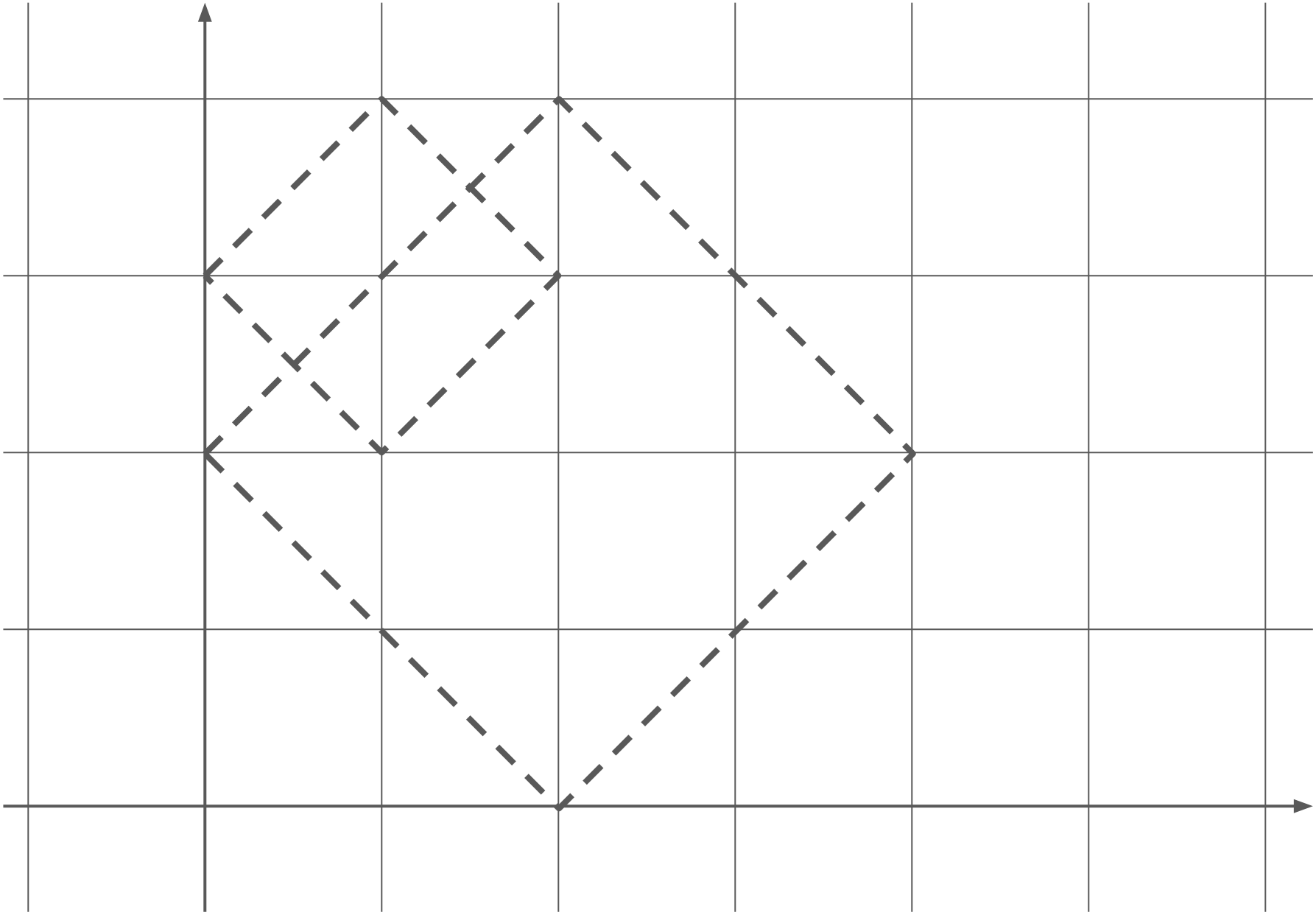

1
1
2
3
3
2
1

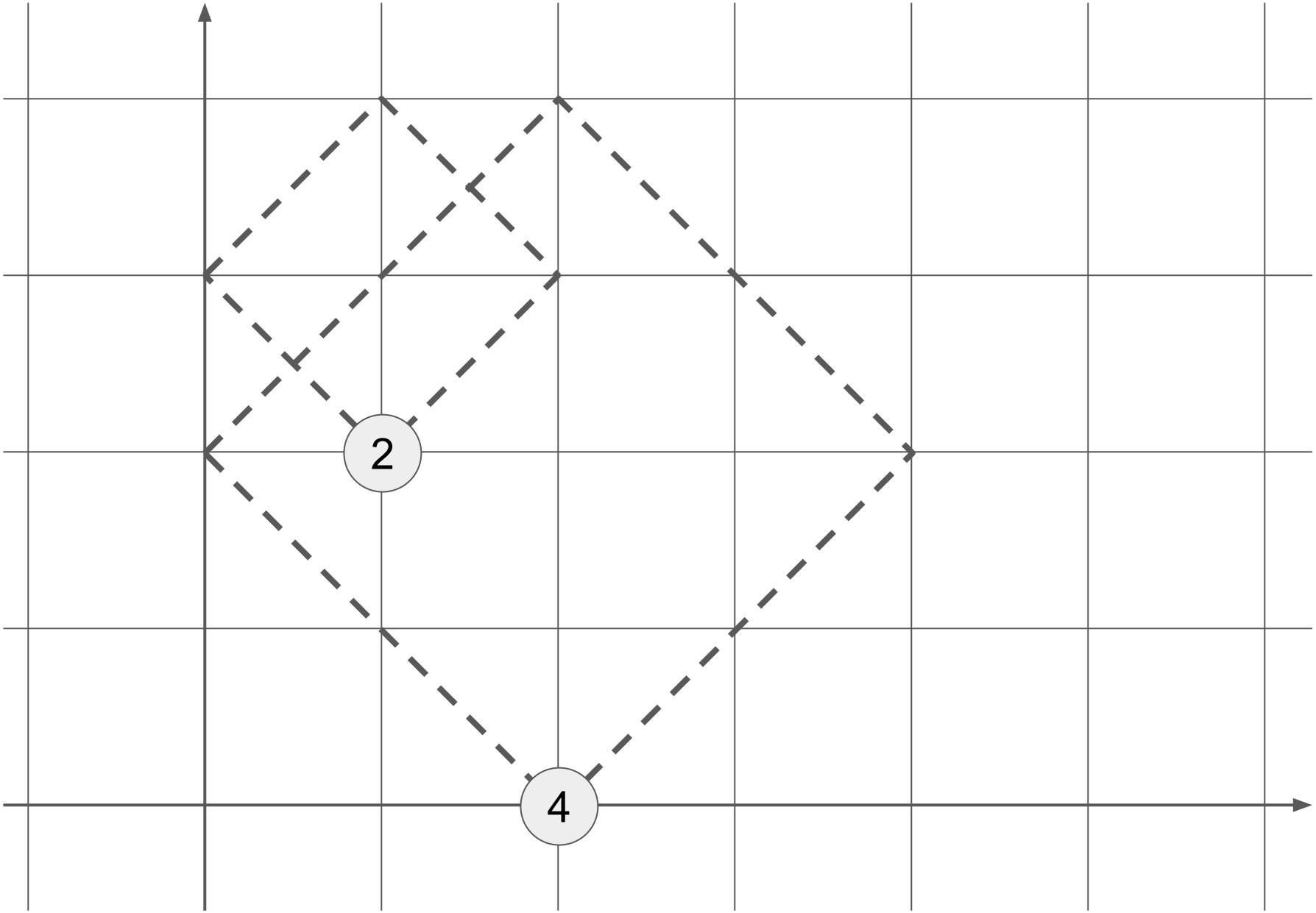

1
2
2
2
3
2
1

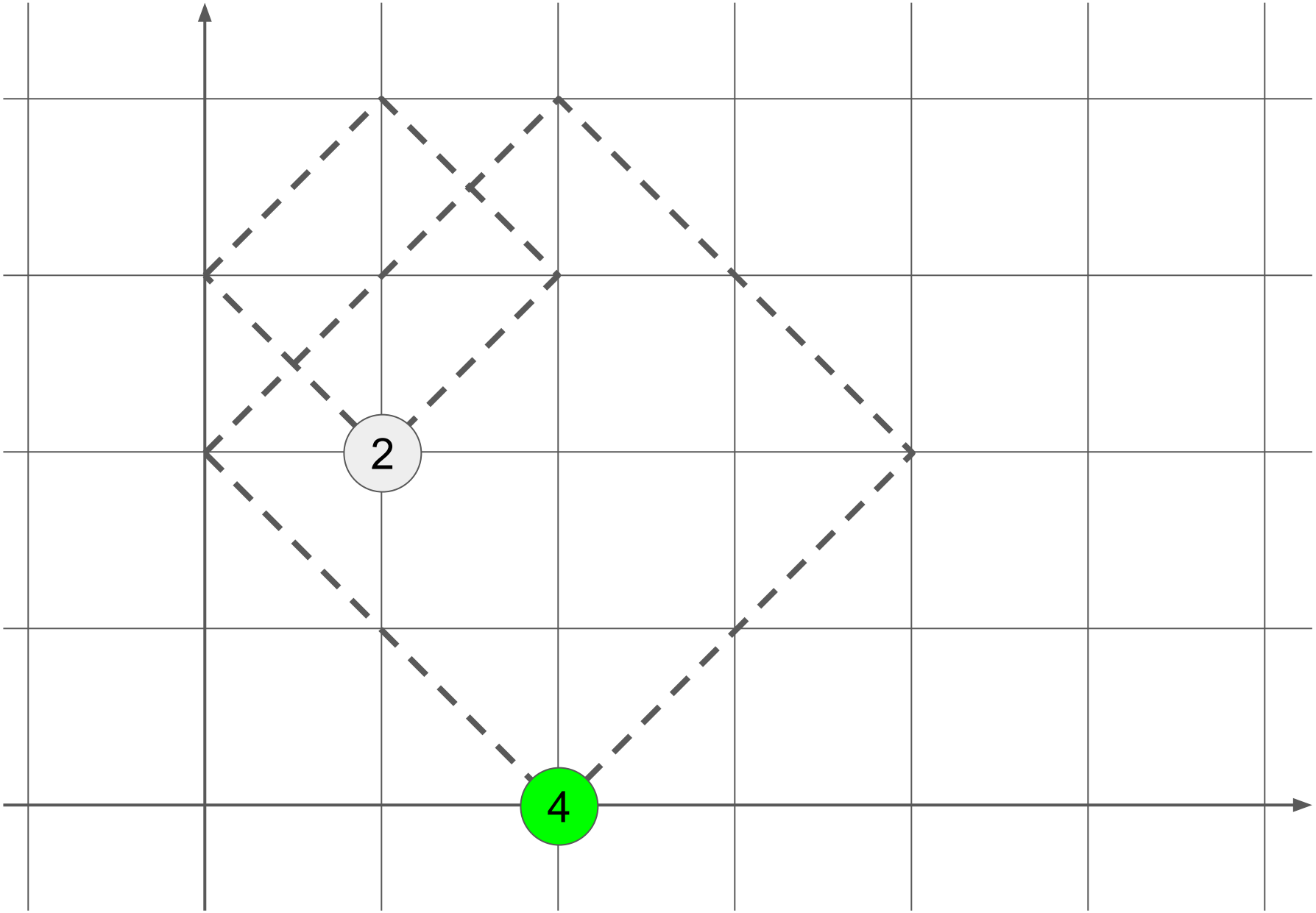

1
2
2
1
2

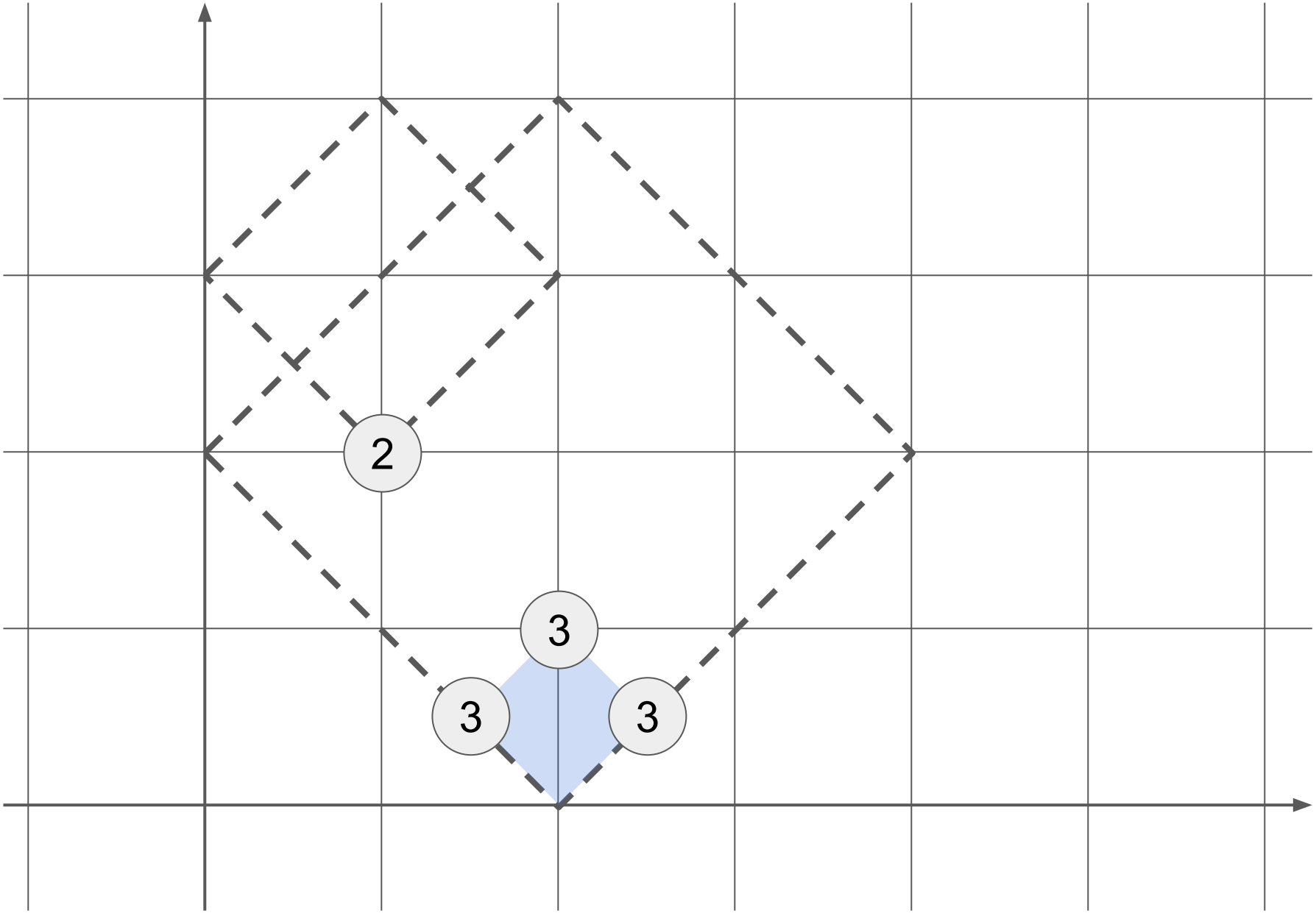

1
1
1
1
1

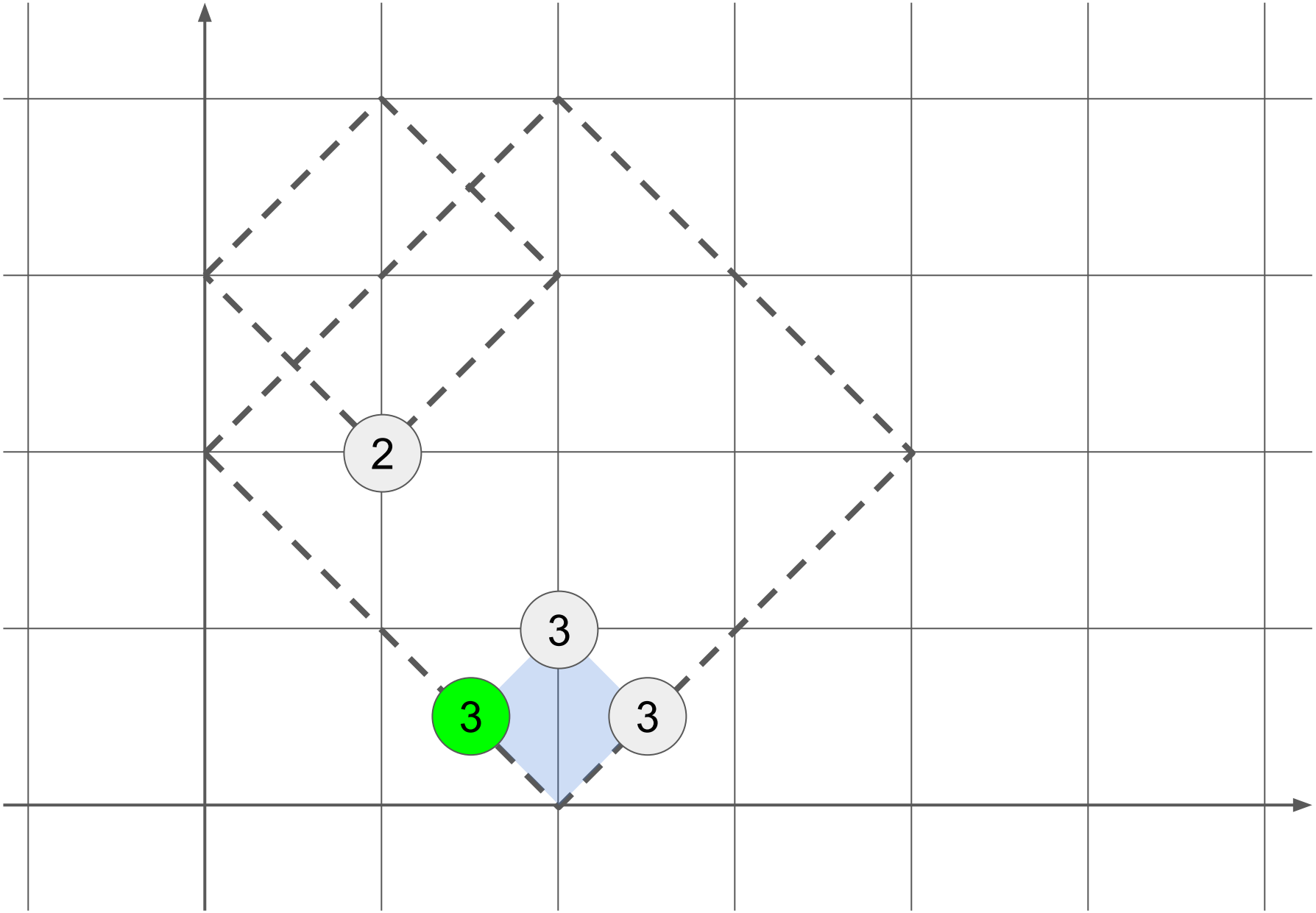


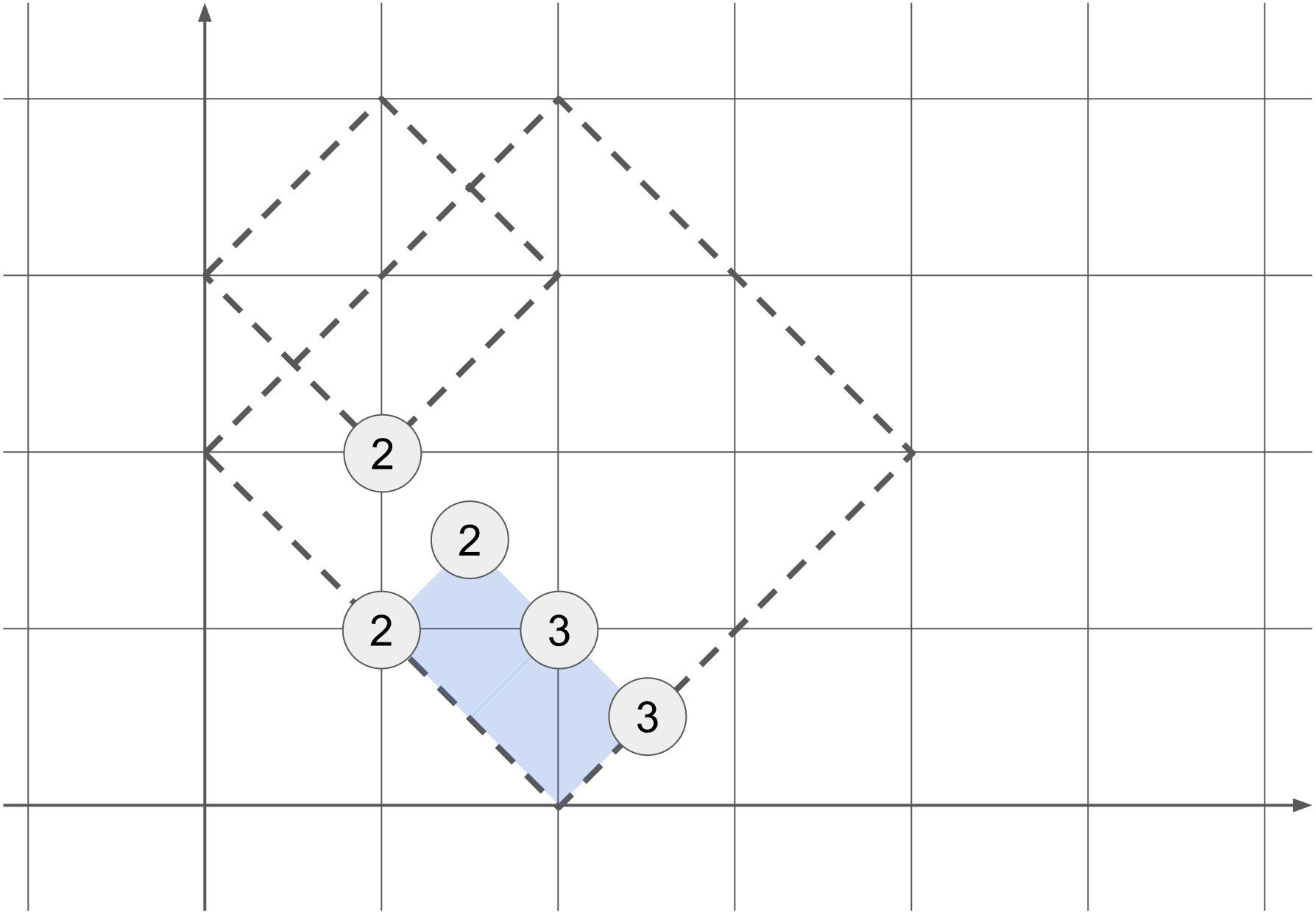


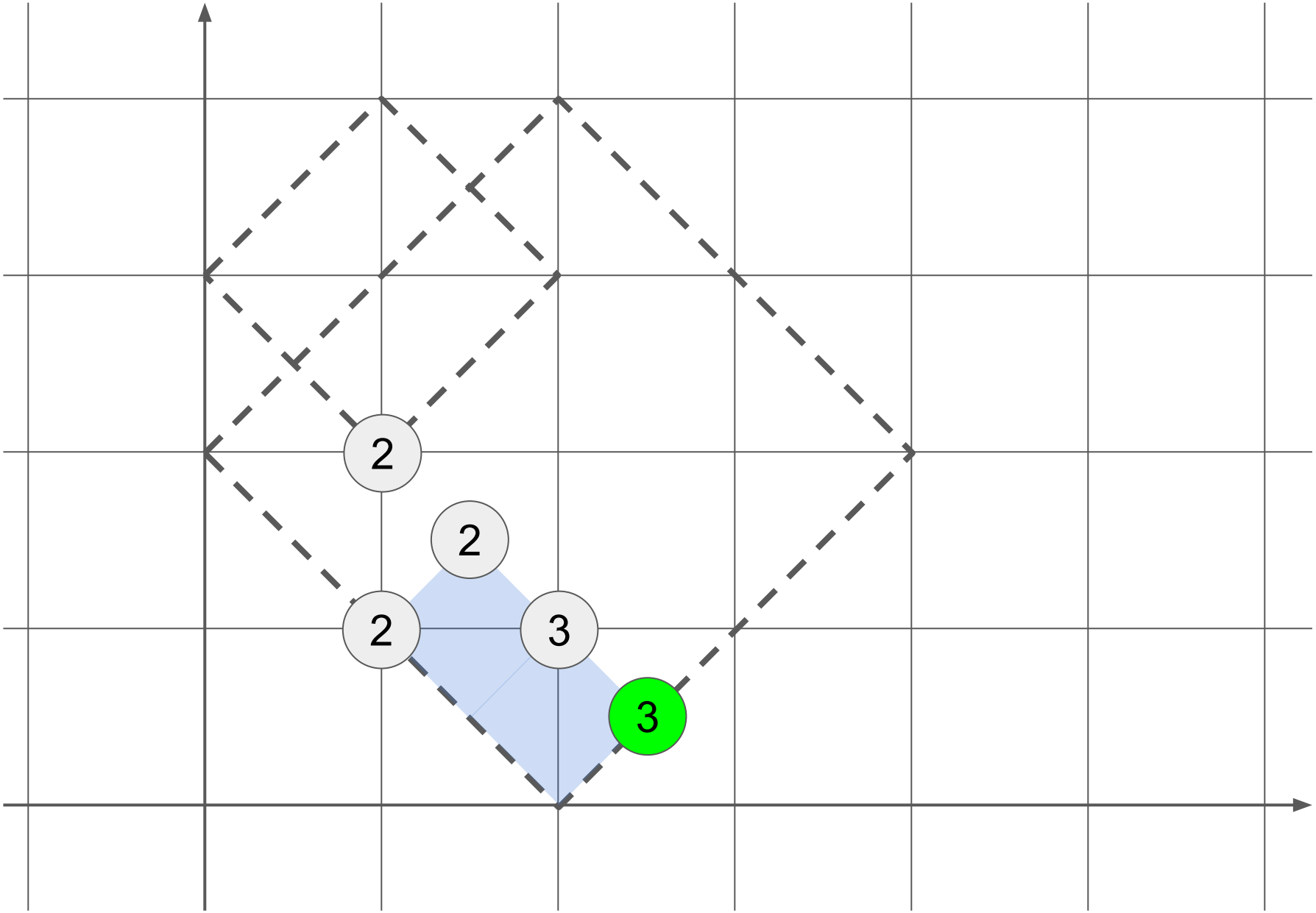

2
4

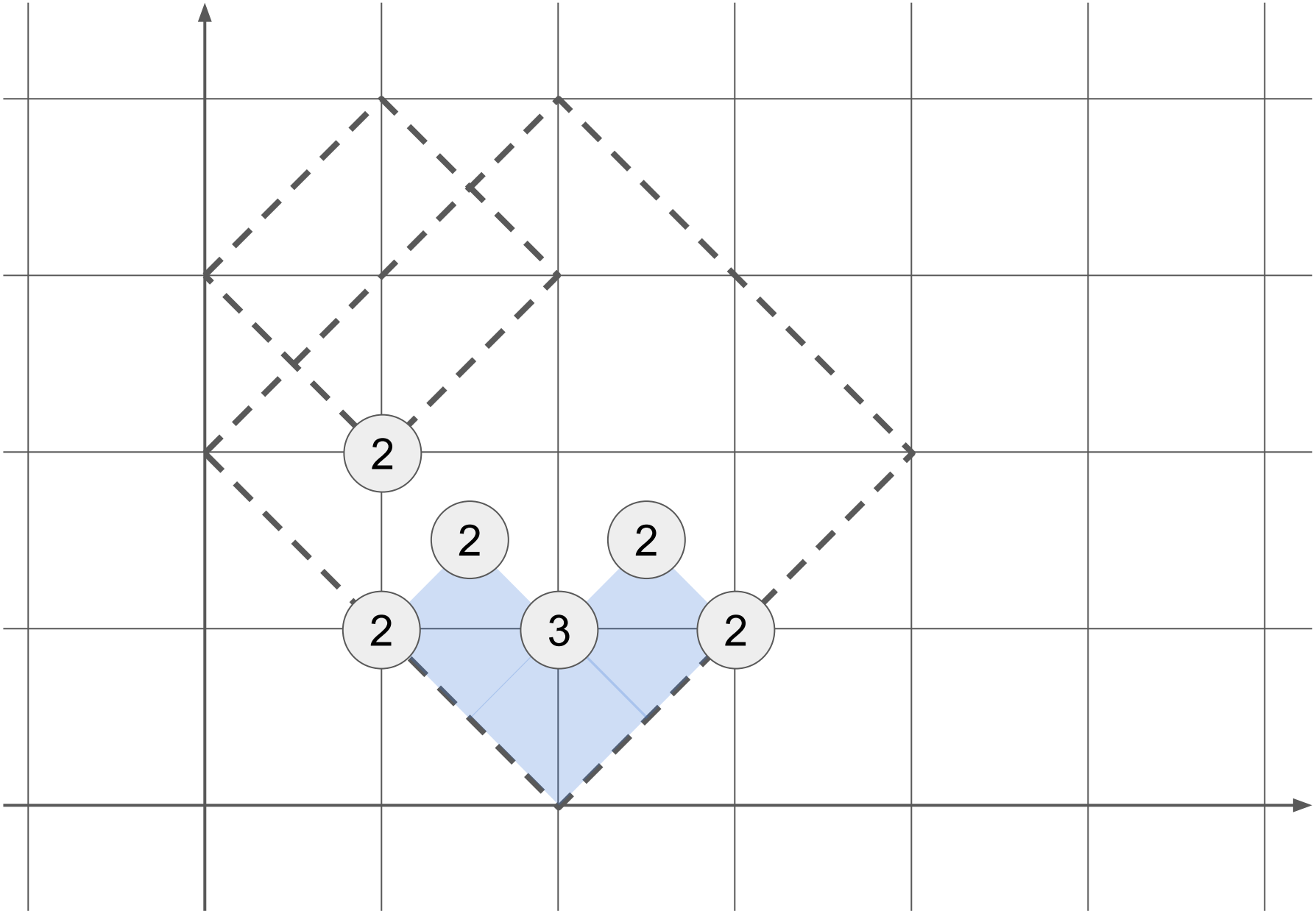

2
4

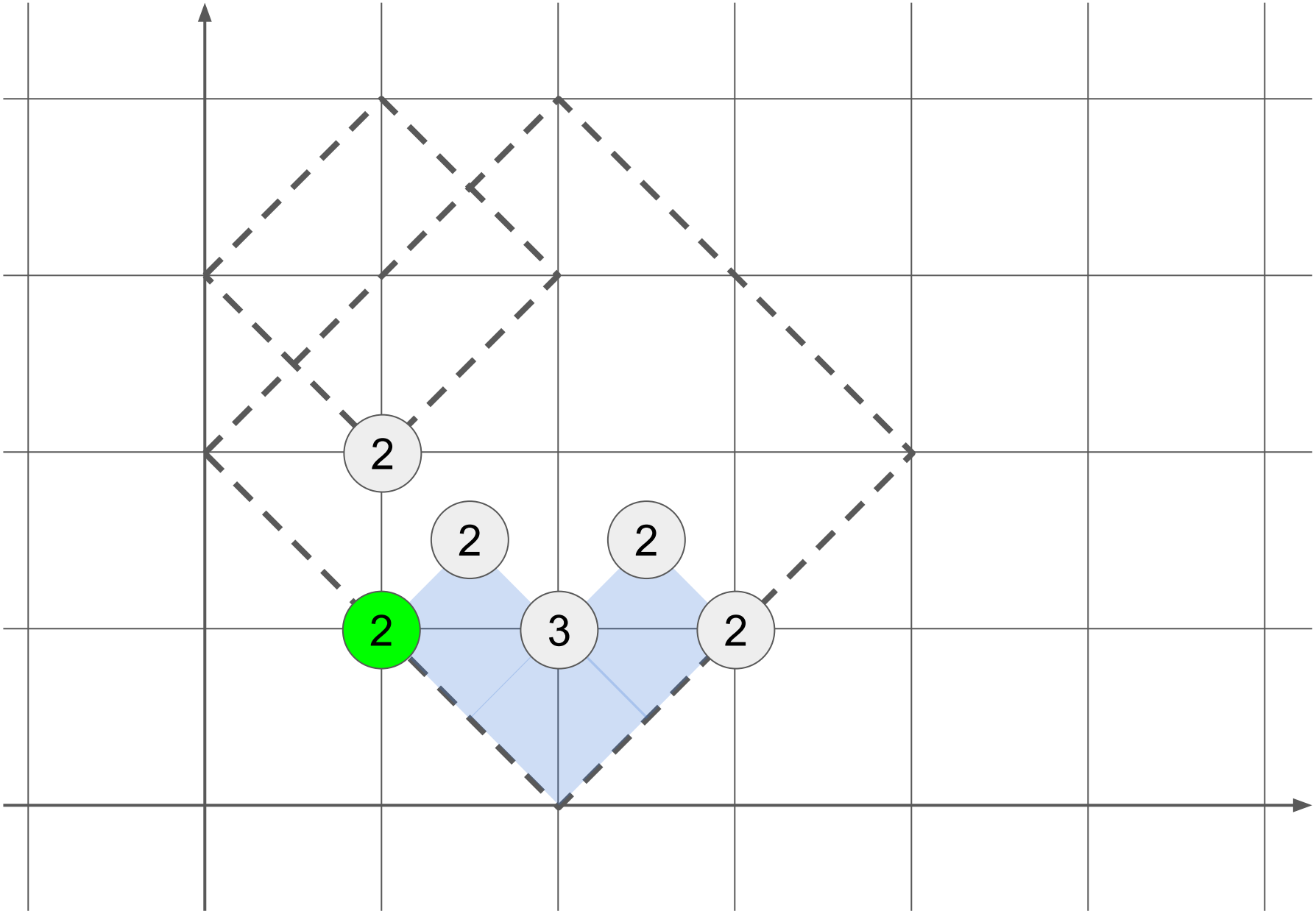

2
3
3
3

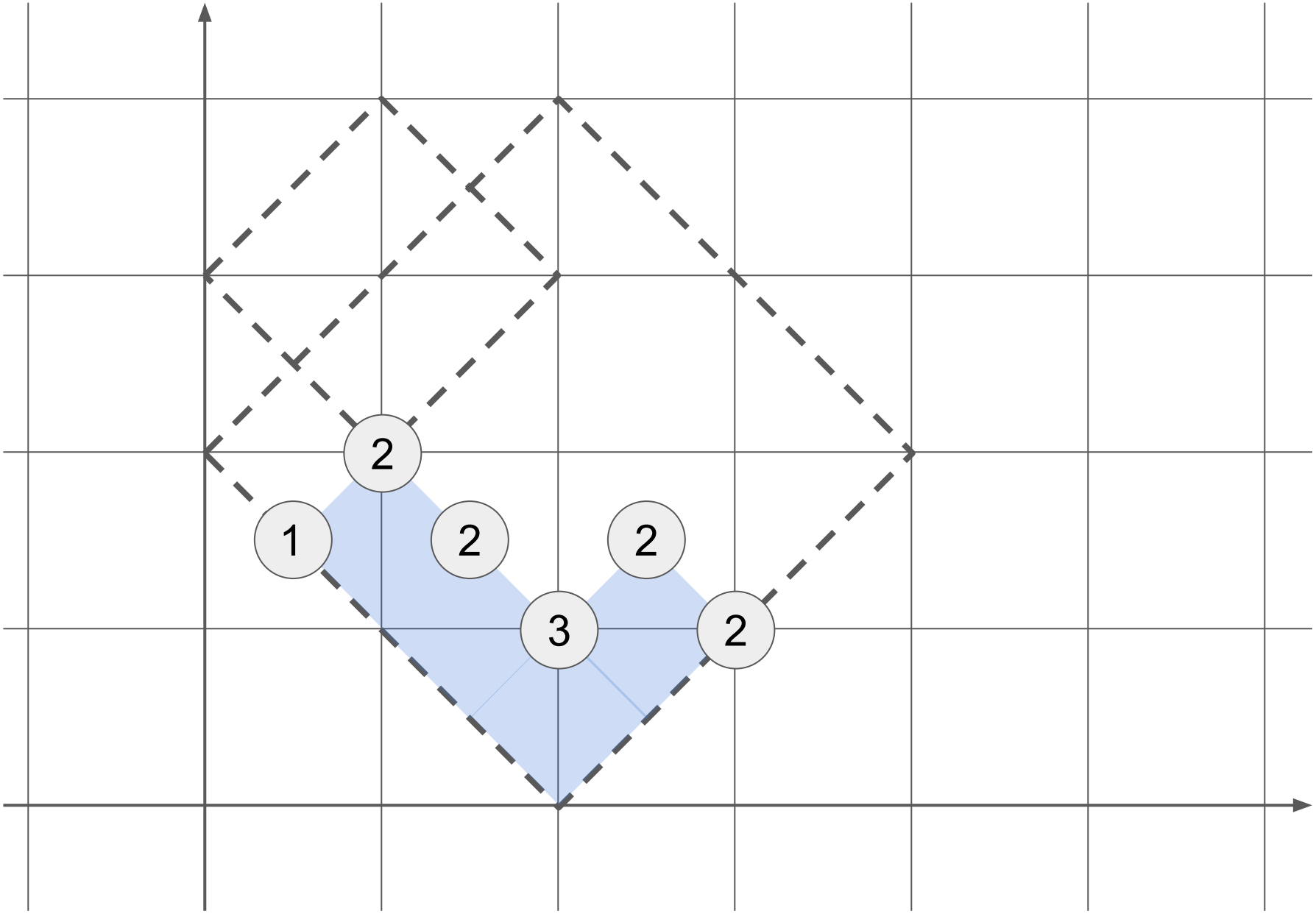

2
3
3
3

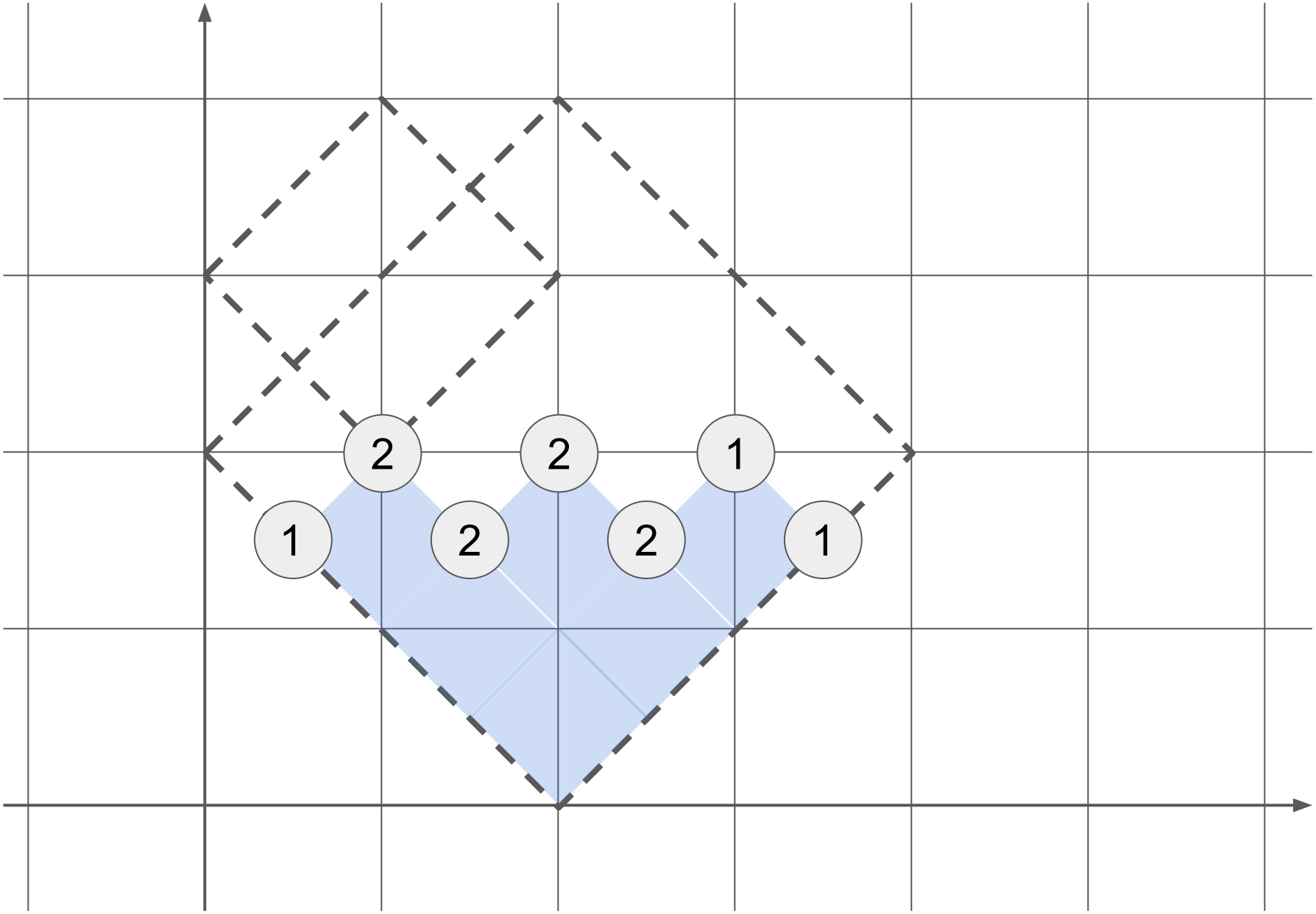

2
2
2
3
3

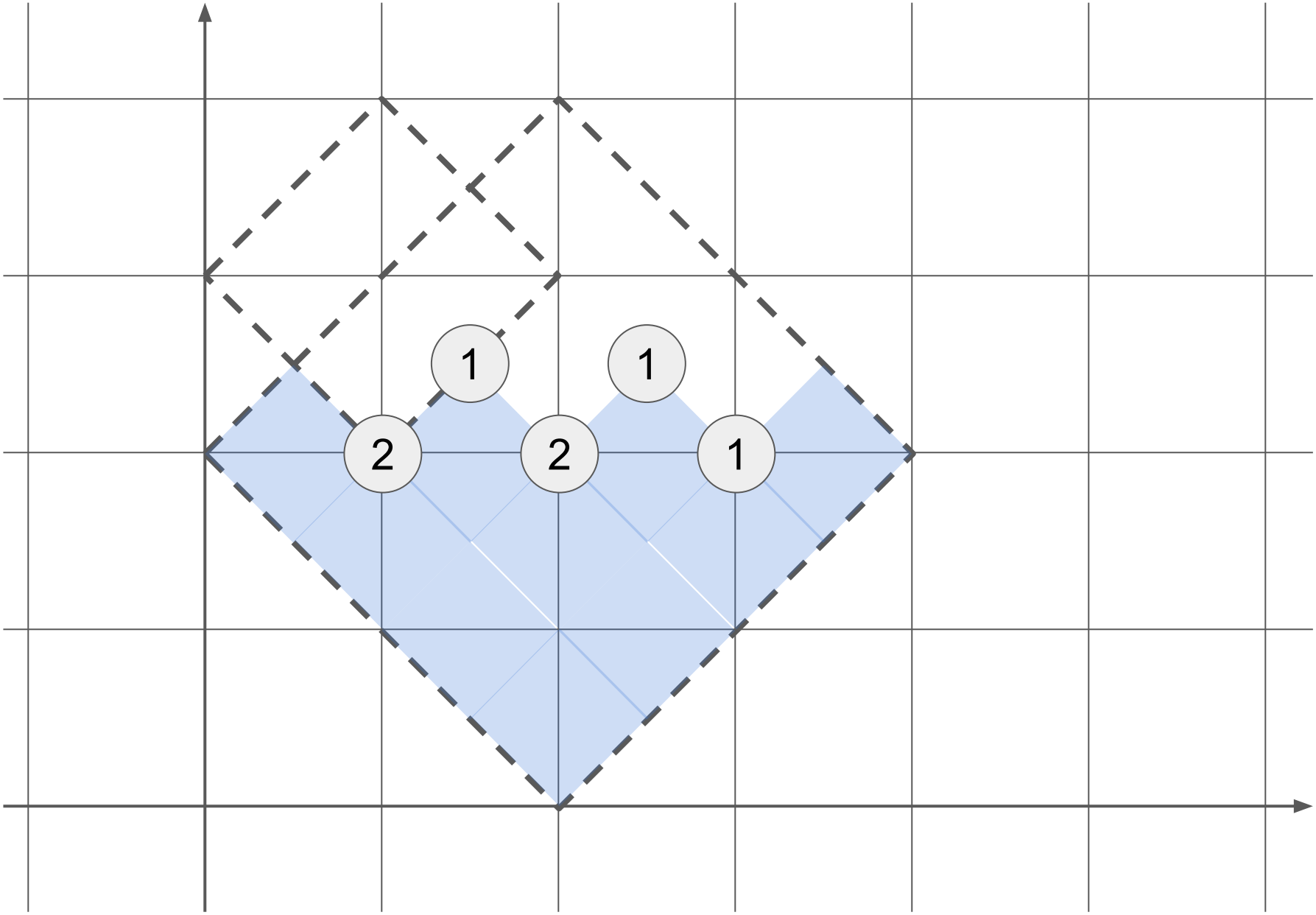

2
2
2
3
3

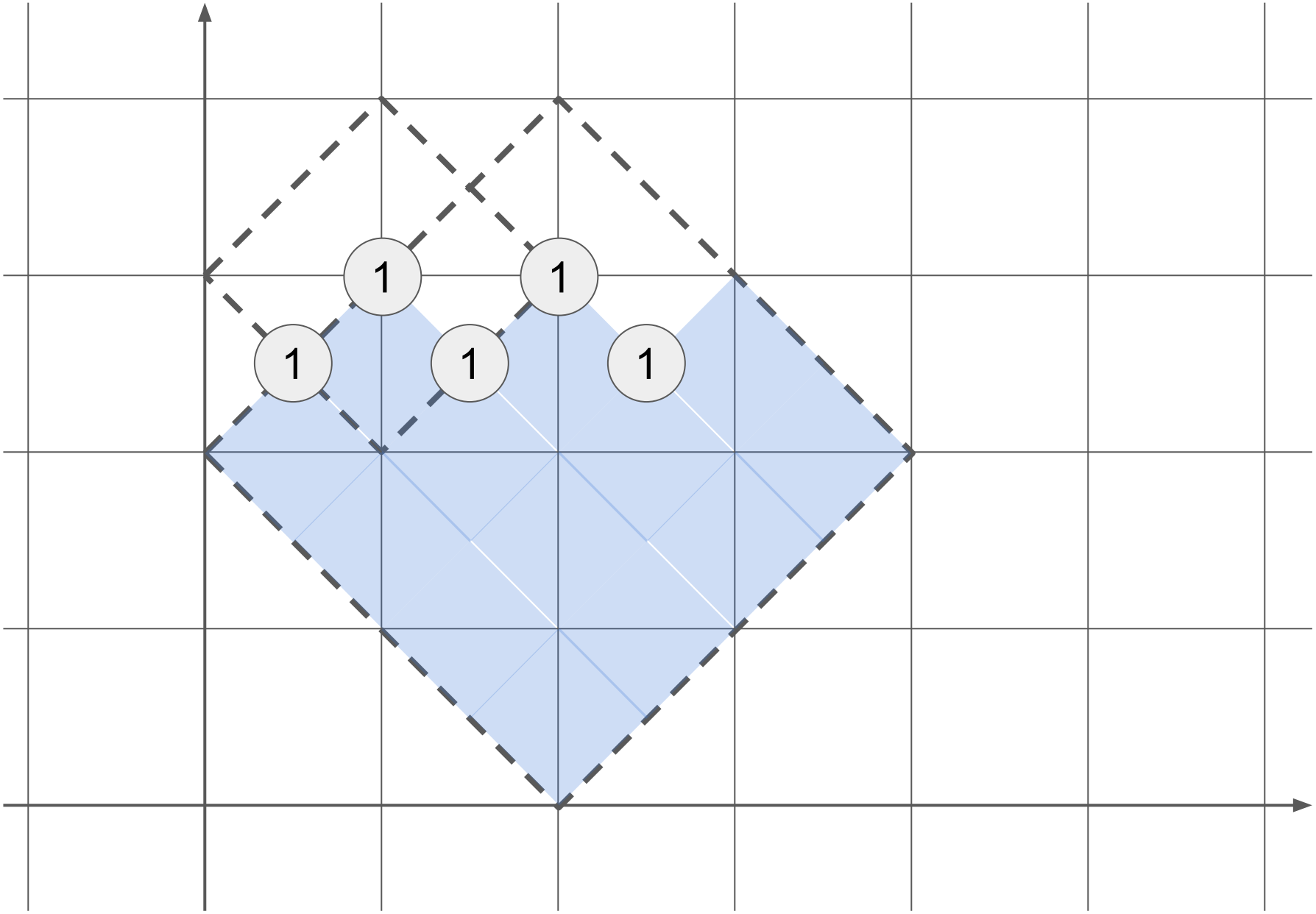

2
2
2
2
3
2

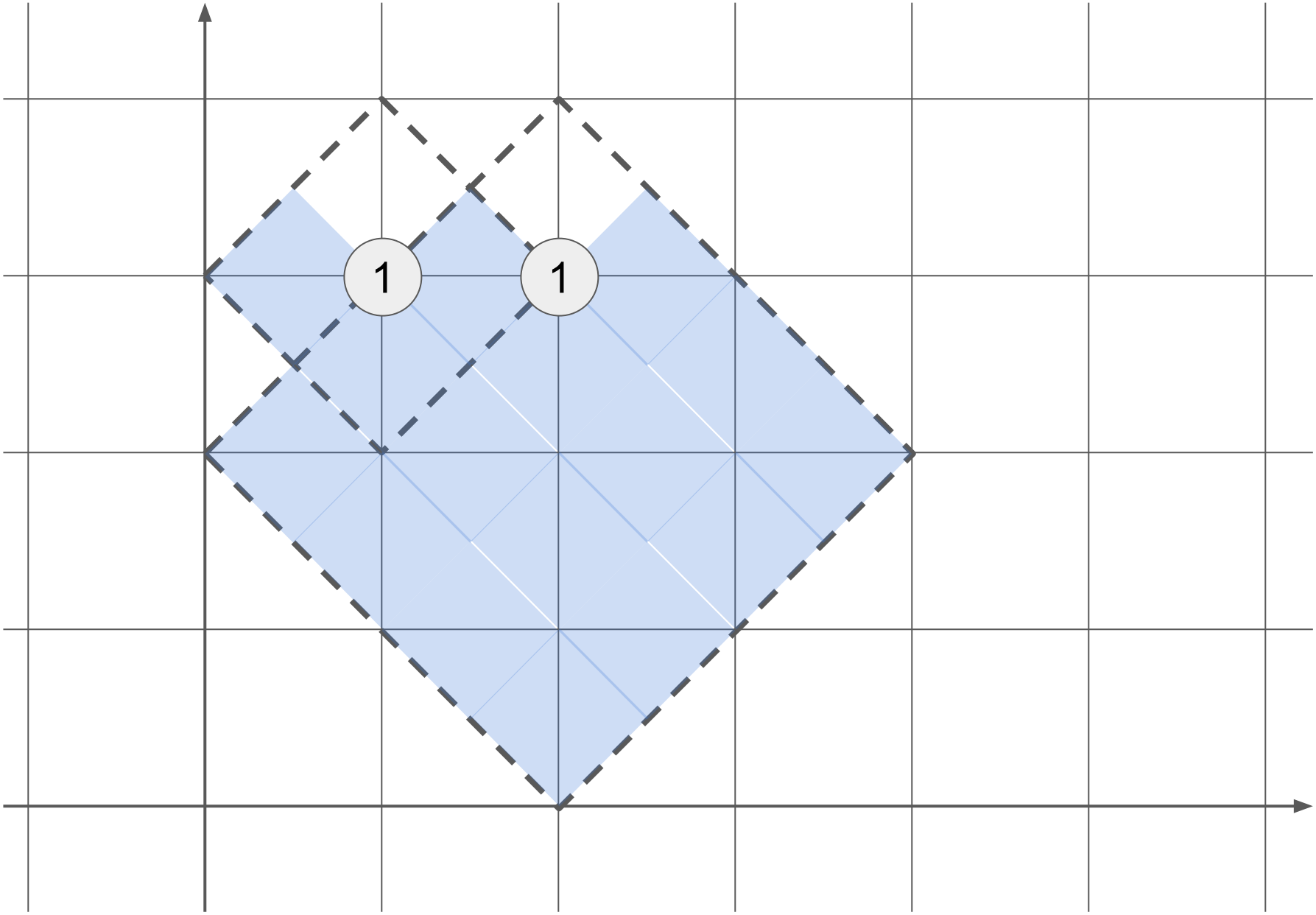

2
2
2
2
3
2

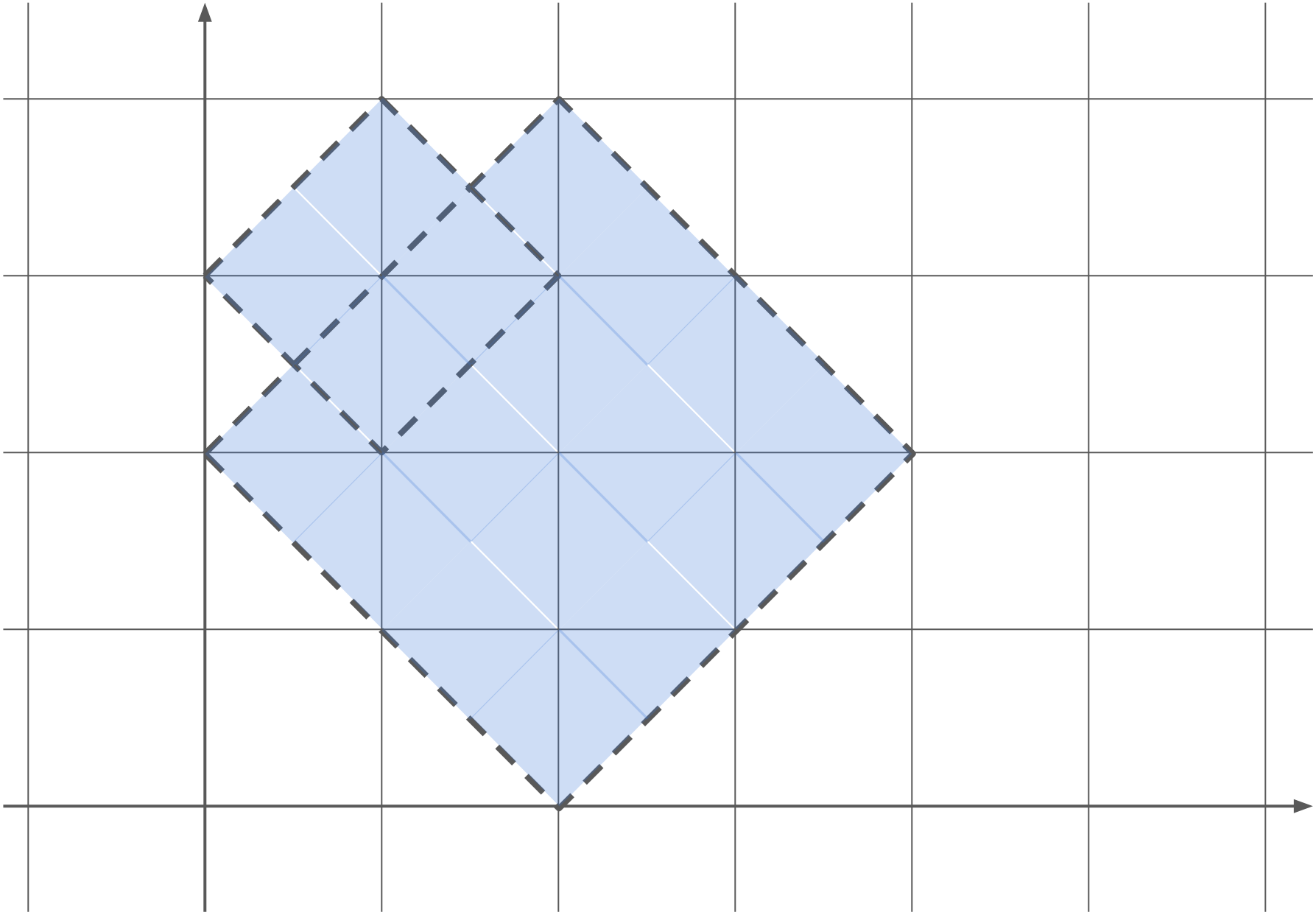

2
1
2
2
3
2

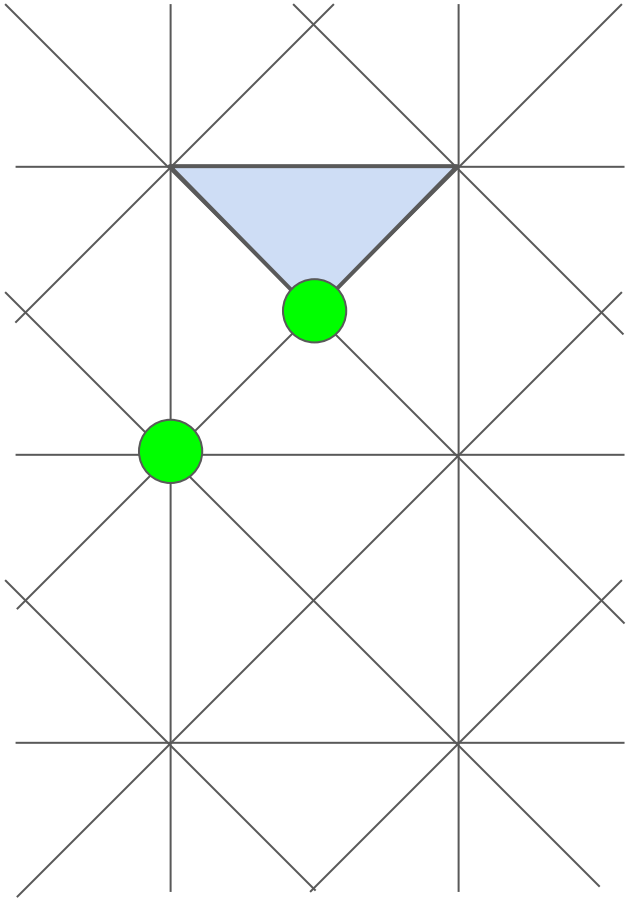

2
2
1
1
2
2
1


1
1
2
2
1

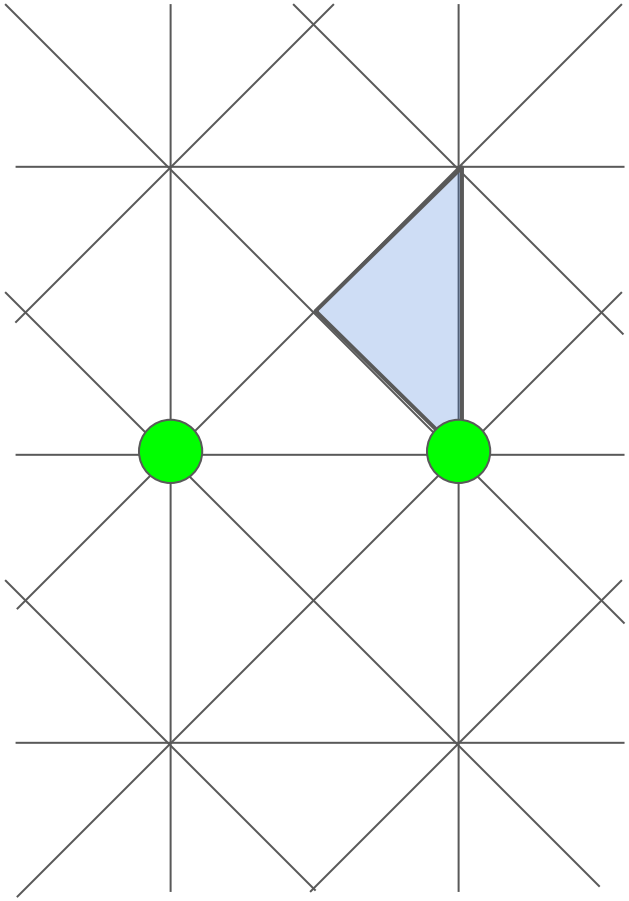

1
1
1
1
1


1
1

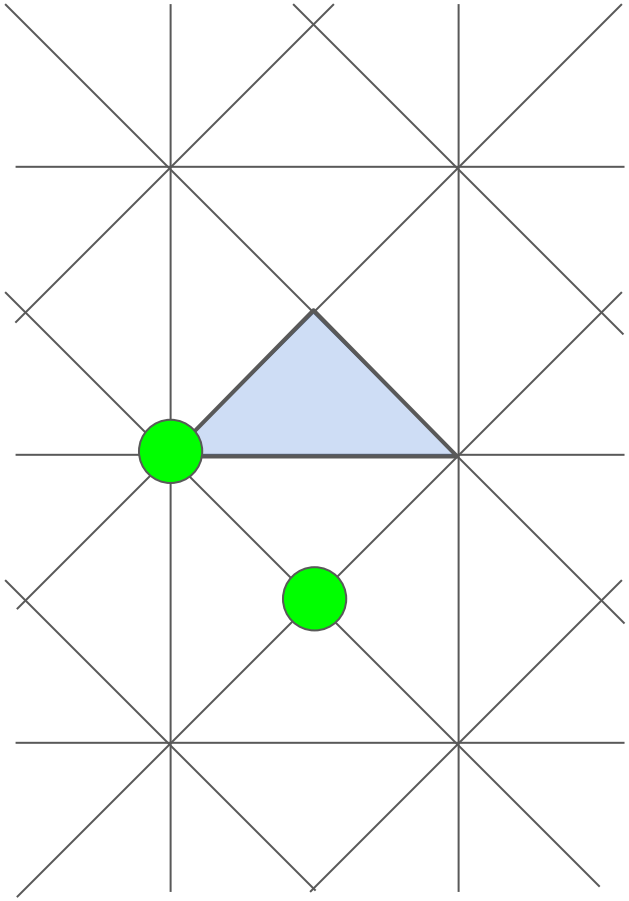




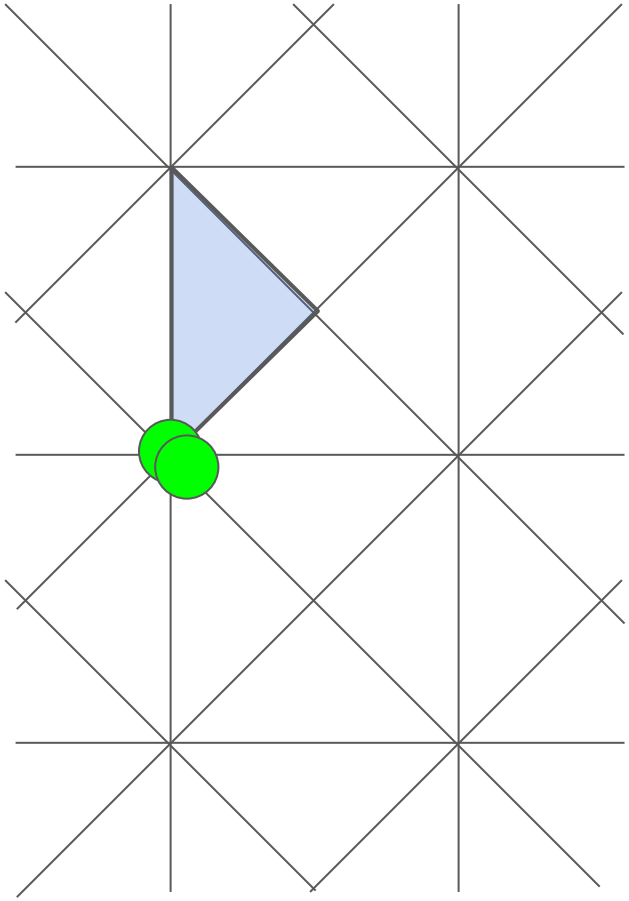
For each triangle we can deduce whether it was painted or not.
We need to check whether we ever had a type A bucket of paint in the lower left corner of the unit square or a type B bucket in the right position (depending on the triangle type).



For each triangle we can deduce whether it was painted or not.
We need to check whether we ever had a type A bucket of paint in the lower left corner of the unit square or a type B bucket in the right position (depending on the triangle type).



For each triangle we can deduce whether it was painted or not.
We need to check whether we ever had a type A bucket of paint in the lower left corner of the unit square or a type B bucket in the right position (depending on the triangle type).



For each triangle we can deduce whether it was painted or not.
We need to check whether we ever had a type A bucket of paint in the lower left corner of the unit square or a type B bucket in the right position (depending on the triangle type).

Time complexity: $O(n + H \cdot W)$
Memory complexity: $O(H \cdot W)$

Problem G
Gambling Guide

Submits: 41
Accepted: at least 16

First solved by: UW1
University of Warsaw
(Dębowski, Radecki, Sommer)
01:30:08

Author: Gustav Matula

Problem:
You're located at a node in an undirected graph.

In each step a neighboring node is chosen at random, and you can either move there or stay where you are.

Find the expected number of steps to get from node 1 to node N, if you used an optimal strategy.

Assume we knew f(x) - the expected number of steps to get from node x to node N.

The optimal strategy to use at each node x is then an obvious one: when offered to move to a neighbour y, move if $f(y) < f(x)$, and stay otherwise.

But we don't know f(x), except for $f(N) = 0$.

Let S be a set of nodes for which we know the value of f(x). Starting from $S = \{N\}$, we'll keep adding nodes one by one in the order of increasing values f(x).

To find the next node to add, we consider nodes outside of S, but neighbouring some node in S. Compute the f'(x) for each such node following the strategy as if that node is the next to add (i.e. move to nodes in S, or stay otherwise).

$$f'(x) = 1 + \sum_{\text{neighbour } y \in S} \frac{f(y)}{degree(x)} + \sum_{\text{neighbour } y \notin S} \frac{f'(x)}{degree(x)}$$

$$f'(x) = \frac{degree(x) + \sum_{\text{neighbour } y \in S} f(y)}{degree(x) - \sum_{\text{neighbour } y \notin S} 1}$$

The node with minimal f'(x) is the next to add.
We set f(x) = f'(x) and add x to S.

We end up with an algorithm very similar to Dijkstra's single source shortest path algorithm, and we can implement it efficiently using the same techniques.

Complexity: $O((N + M) \log N)$ using the classic implementation with a binary heap (or STL set).

Problem D
Donut Drone

Submits: 60
Accepted: ?

Author: Luka Kalinovčić

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

The task is to implement two functions:
move(k): Moves a drone k steps and reports the final coordinates.
update(row, col, value): Updates the elevation at provided coordinates.

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

Let’s start with a naive solution:

```
def simple_move(k):
  for i in range(k):
    coords = step(coords)
  return coords
```

Complexity: O(k) - too slow.

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

Observation: The drone will eventually enter a cycle.

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

Observation: The drone will eventually enter a cycle.

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

Observation: The drone will eventually enter a cycle.

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

Observation: The drone will eventually enter a cycle.

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

Observation: The drone will eventually enter a cycle.

1	2	6	1	1
2	4	1	2	2
5	5	5	3	5
3	1	2	5	3

Observation: The drone will eventually enter a cycle.

```
def smarter_move(k):
  first_seen = dict()
  for i in range(k):
    if coords not in first_seen:
      first_seen[coords] = i
    else:
      cycle_length = i - first_seen[coords]
      steps_left = k - i
      return simple_move(steps_left % cycle_length)
    coords = step(coords)
  return coords
```

Complexity $O(R \cdot C)$ - still too slow in the worst case.

Key idea: Maintain an array jump[row] that stores the cell we would end up if we moved C steps from a cell (row, 1) in the first column.

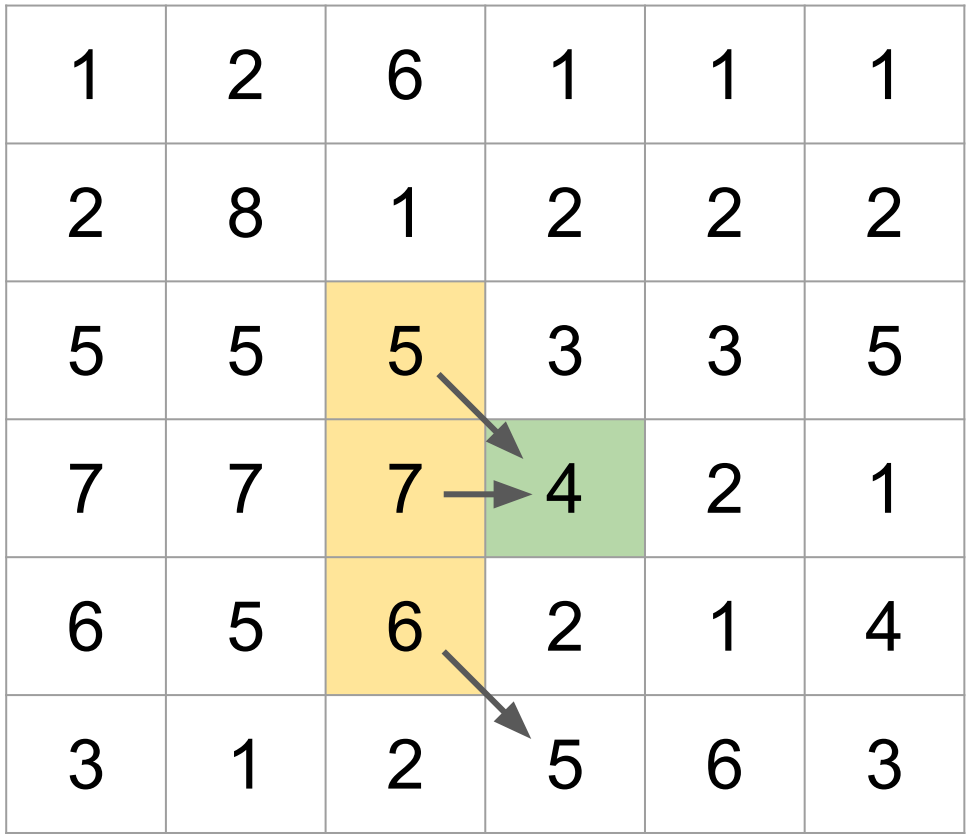
As soon as we reach the first column we can start making jumps of size C that stay in the first column until there are less C steps to make.

Then we proceed to make single steps again to find the final cell.

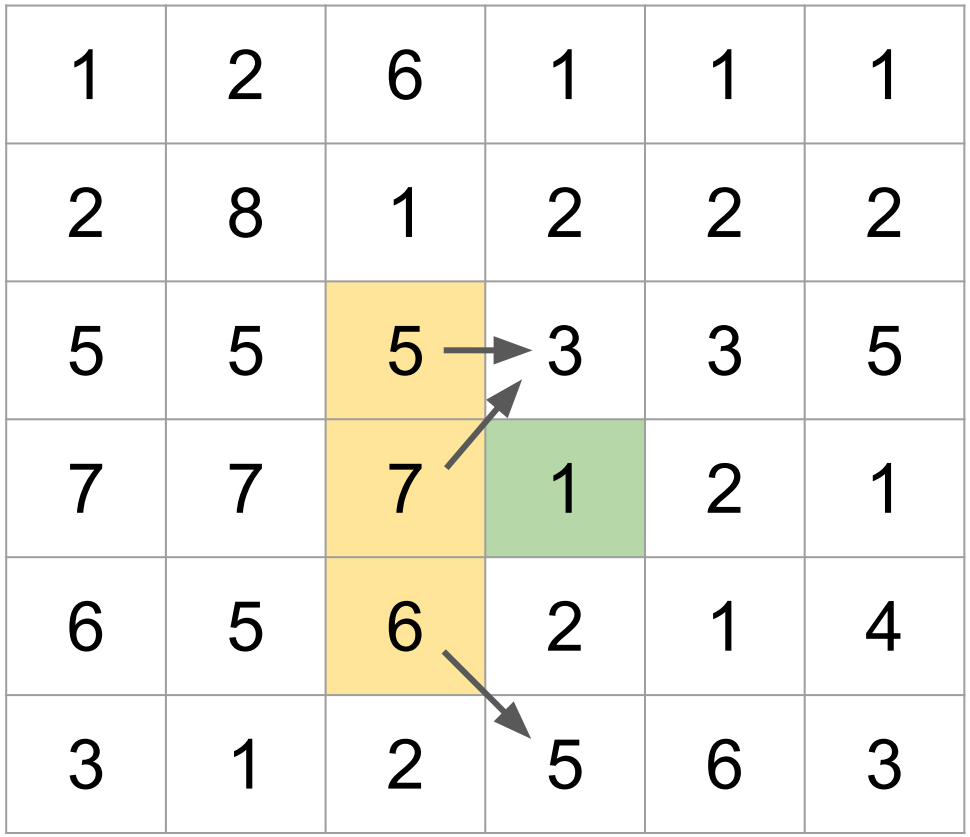
If we also implement the cycle detection among the cells in the first column we end up with a O(R + C) move operation.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	4	2	1
6	5	6	2	1	4
3	1	2	5	6	3

However, the update(row, col, value) becomes tricky, as we may need to update the jump[row] array.



Up to three cells may be directly affected.



Up to three cells may be directly affected.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	1	2	1
6	5	6	2	1	4
3	1	2	5	6	3

For each affected cell, we'll run the update algorithm.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	1	2	1
6	5	6	2	1	4
3	1	2	5	6	3

For each affected cell, we'll run the update algorithm.
1) Repeatedly make steps to find in which cell in the first column we'll end up.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	1	2	1
6	5	6	2	1	4
3	1	2	5	6	3

For each affected cell, we'll run the update algorithm.
1) Repeatedly make steps to find in which cell in the first column we'll end up.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	1	2	1
6	5	6	2	1	4
3	1	2	5	6	3

For each affected cell, we'll run the update algorithm.
1) Repeatedly make steps to find in which cell in the first column we'll end up.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	1	2	1
6	5	6	2	1	4
3	1	2	5	6	3

For each affected cell, we'll run the update algorithm.
1) Repeatedly make steps to find in which cell in the first column we'll end up.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	1	2	1
6	5	6	2	1	4
3	1	2	5	6	3

For each affected cell, we'll run the update algorithm.
1) Repeatedly make steps to find in which cell in the first column we'll end up.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	1	2	1
6	5	6	2	1	4
3	1	2	5	6	3

For each affected cell, we'll run the update algorithm.
2) Starting from the affected cell, backtrack to the first column, maintaining an interval of affected rows.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	1	2	1
6	5	6	2	1	4
3	1	2	5	6	3

For each affected cell, we'll run the update algorithm.
2) Starting from the affected cell, backtrack to the first column, maintaining an interval of affected rows.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	1	2	1
6	5	6	2	1	4
3	1	2	5	6	3

For each affected cell, we'll run the update algorithm.
2) Starting from the affected cell, backtrack to the first column, maintaining an interval of affected rows.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	1	2	1
6	5	6	2	1	4
3	1	2	5	6	3

For each affected cell, we'll run the update algorithm.
3) If we reach the first column, we have an interval of rows to update jump[row] for.

1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	1	2	1
6	5	6	2	1	4
3	1	2	5	6	3

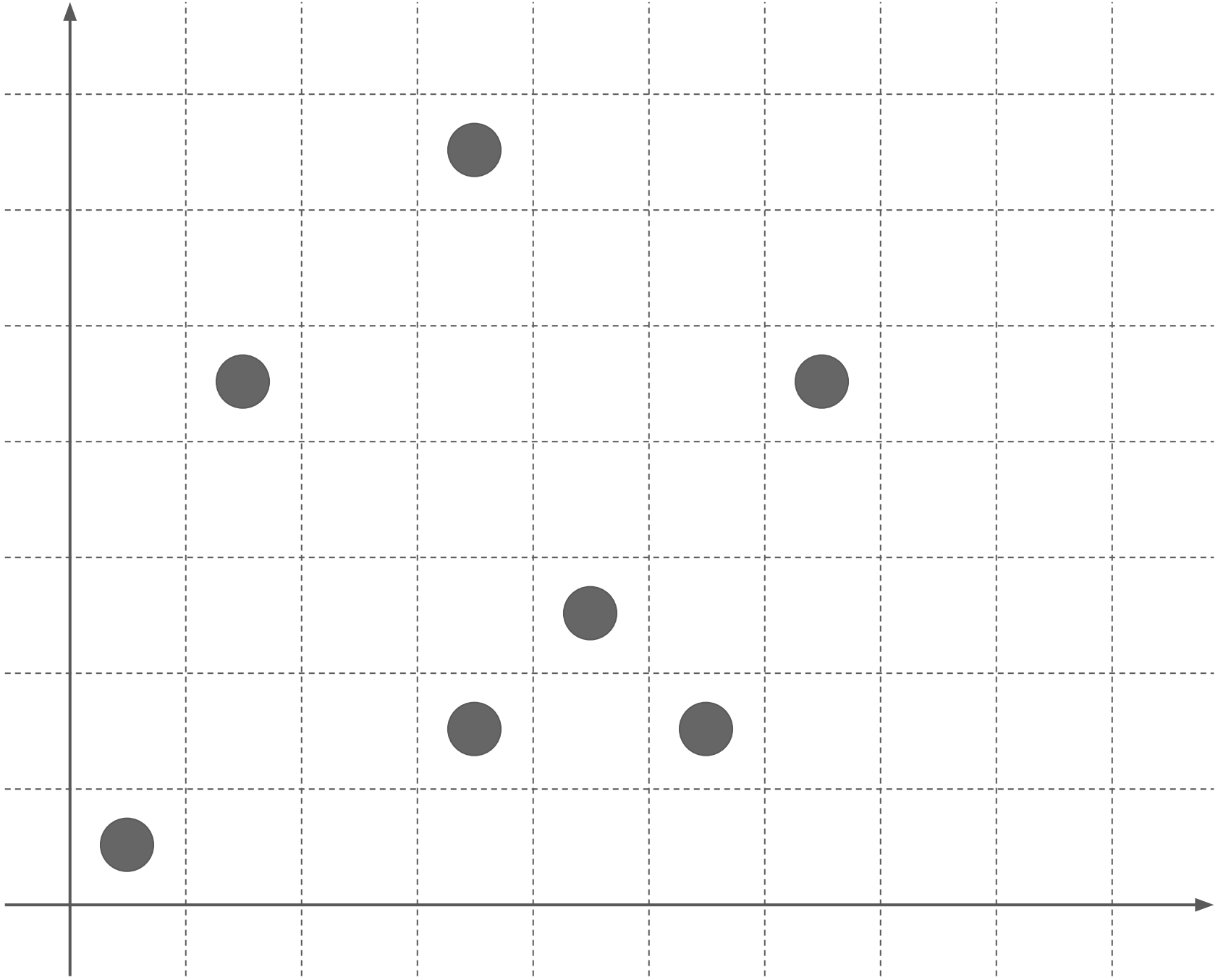
Interval bounds may only move by ±1 between neighbouring columns, so we can maintain the affected interval in O(1) per column as we backtrack. Overall update complexity is O(C).

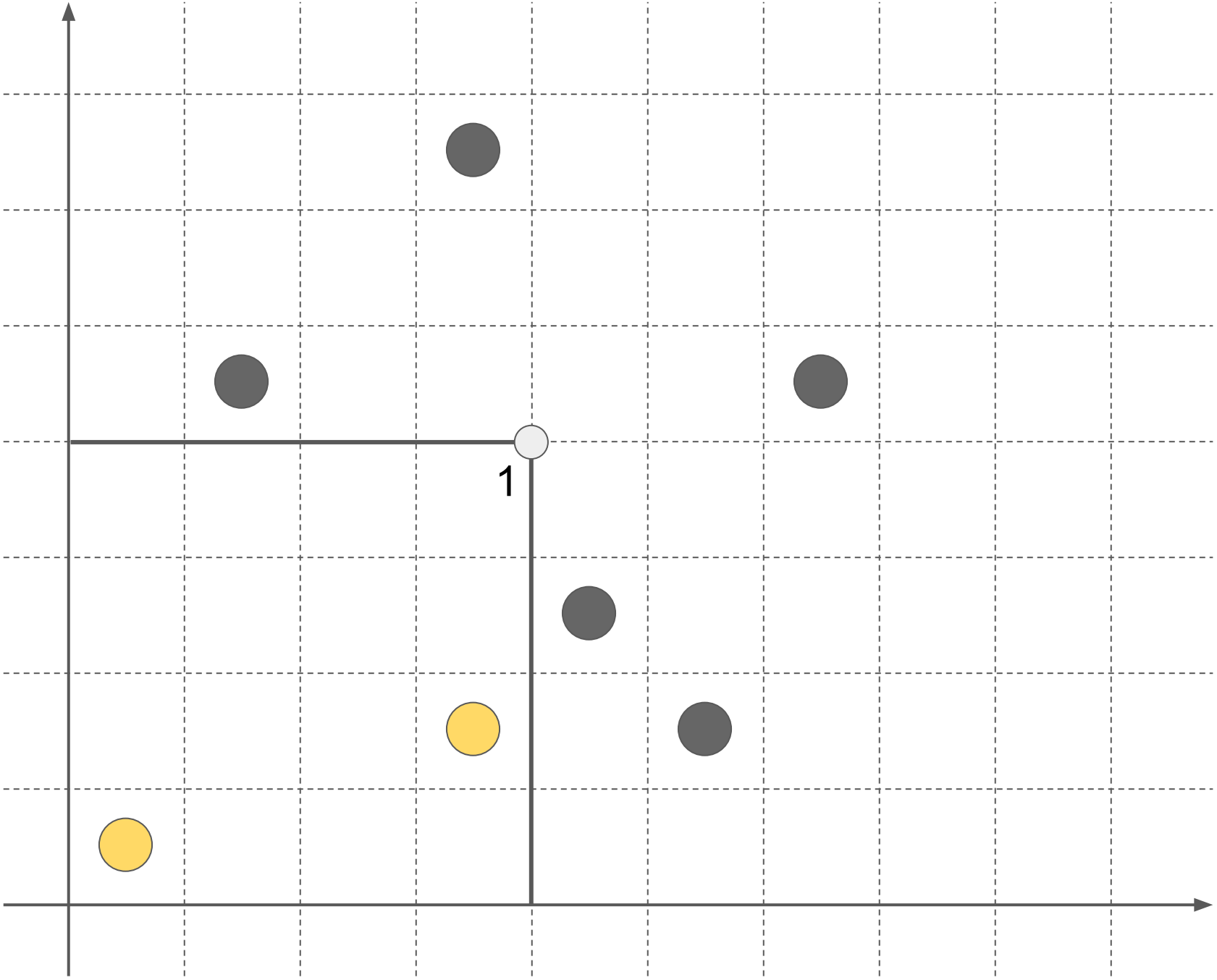
Problem B
Buffalo Barricades

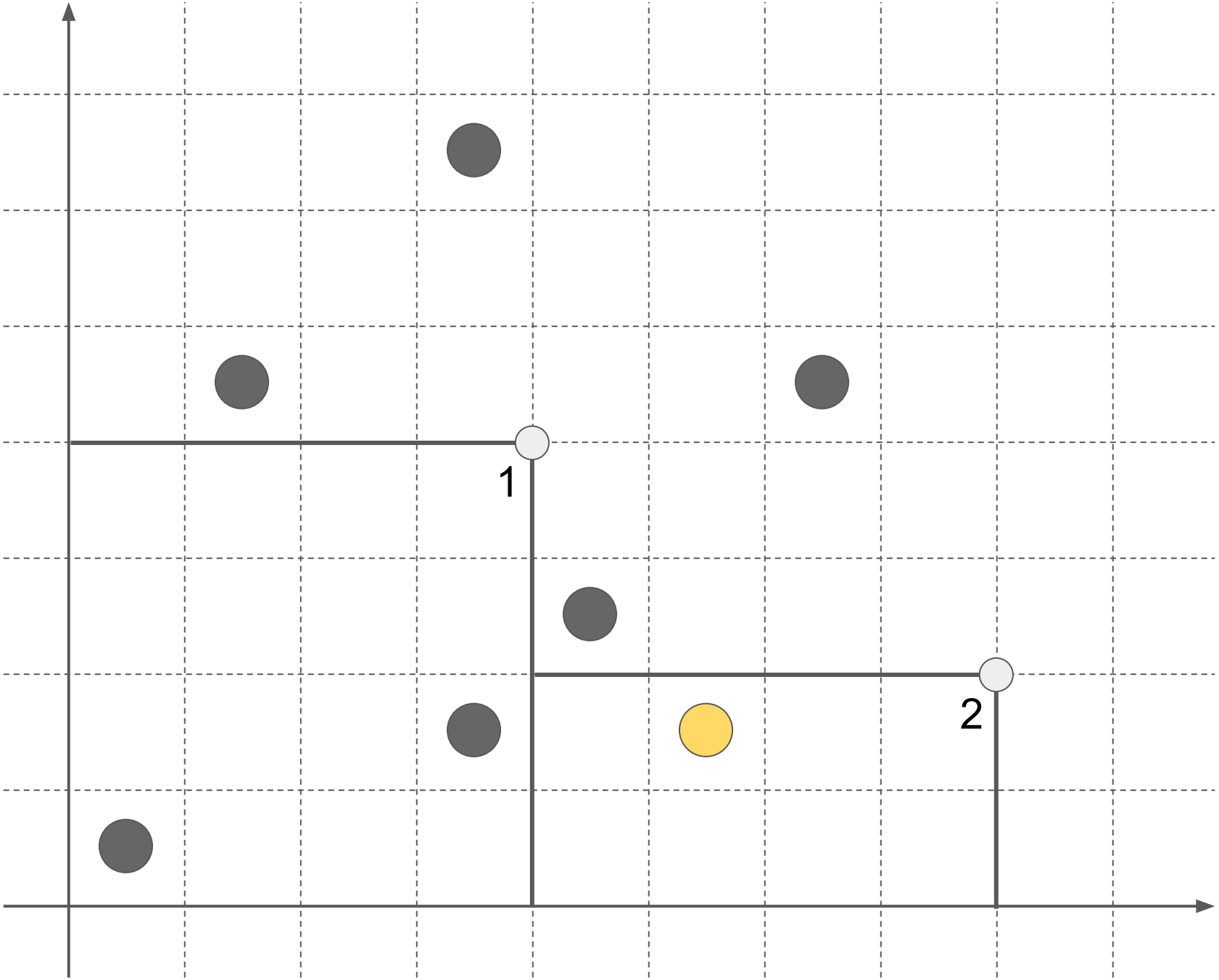
Submits: 17
Accepted: at least 1

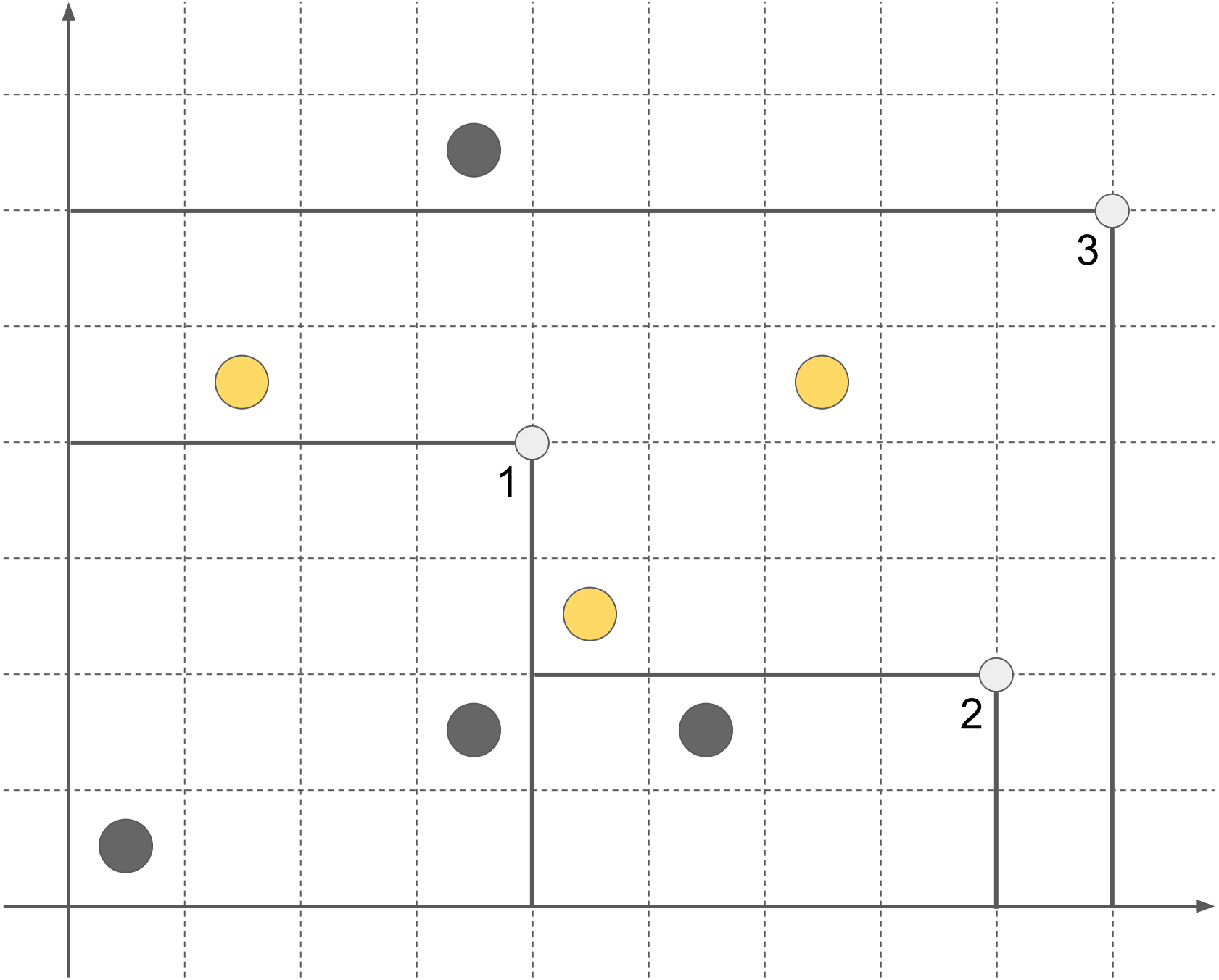
First solved by: UW1
University of Warsaw
(Dębowski, Radecki, Sommer)
02:40:43

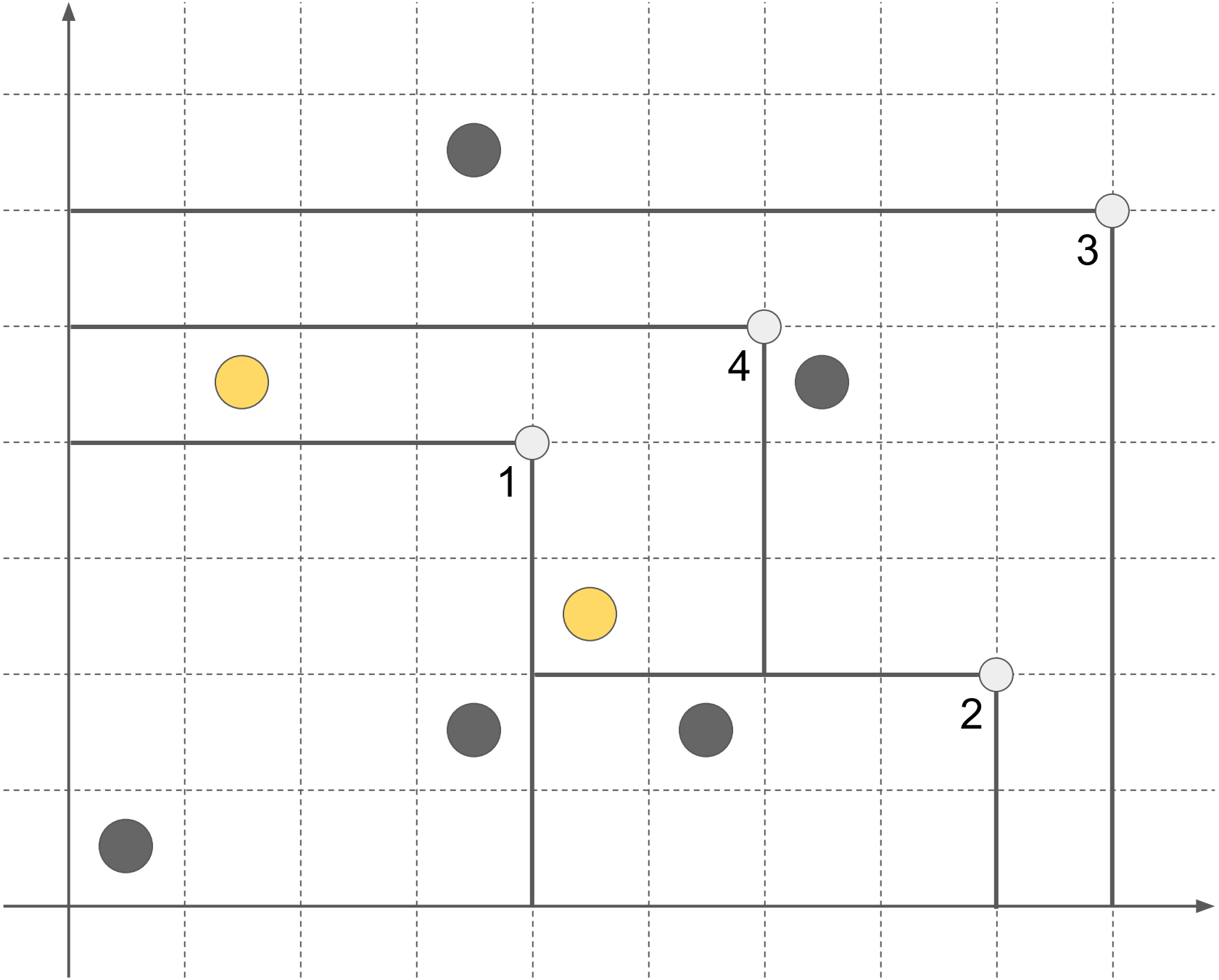
Author: Luka Kalinovčić




1


1
2


3
1
2


3
4
1
2

High level algorithm:

1) Identify the regions at the end, when all fences are up.

2) Count the buffalos in each region.

3) Work backwards, removing fences and merging the two regions that become one (using the standard union-find algorithm). Prior to the fence removal we simply record the current number of buffalos in the region to output later.

We'll do 1) and 2) together in a single pass of a sweep-line algorithm. In addition to that, we'll also compute the ids of regions that need to be merged in step 3) at each fence removal.

Sweep-line algorithm overview:

We process fence posts and buffalos in order of decreasing y coordinate.

At each step we maintain a set of "active" vertical fences that have not yet hit another horizontal fence.

a) When we encounter a buffalo, we find the closest active fence to the right, that's the fence of a region containing the buffalo at the end.

Sweep-line algorithm overview:

We process fence posts and buffalos in order of decreasing y coordinate.

At each step we maintain a set of "active" vertical fences that have not yet hit another horizontal fence.

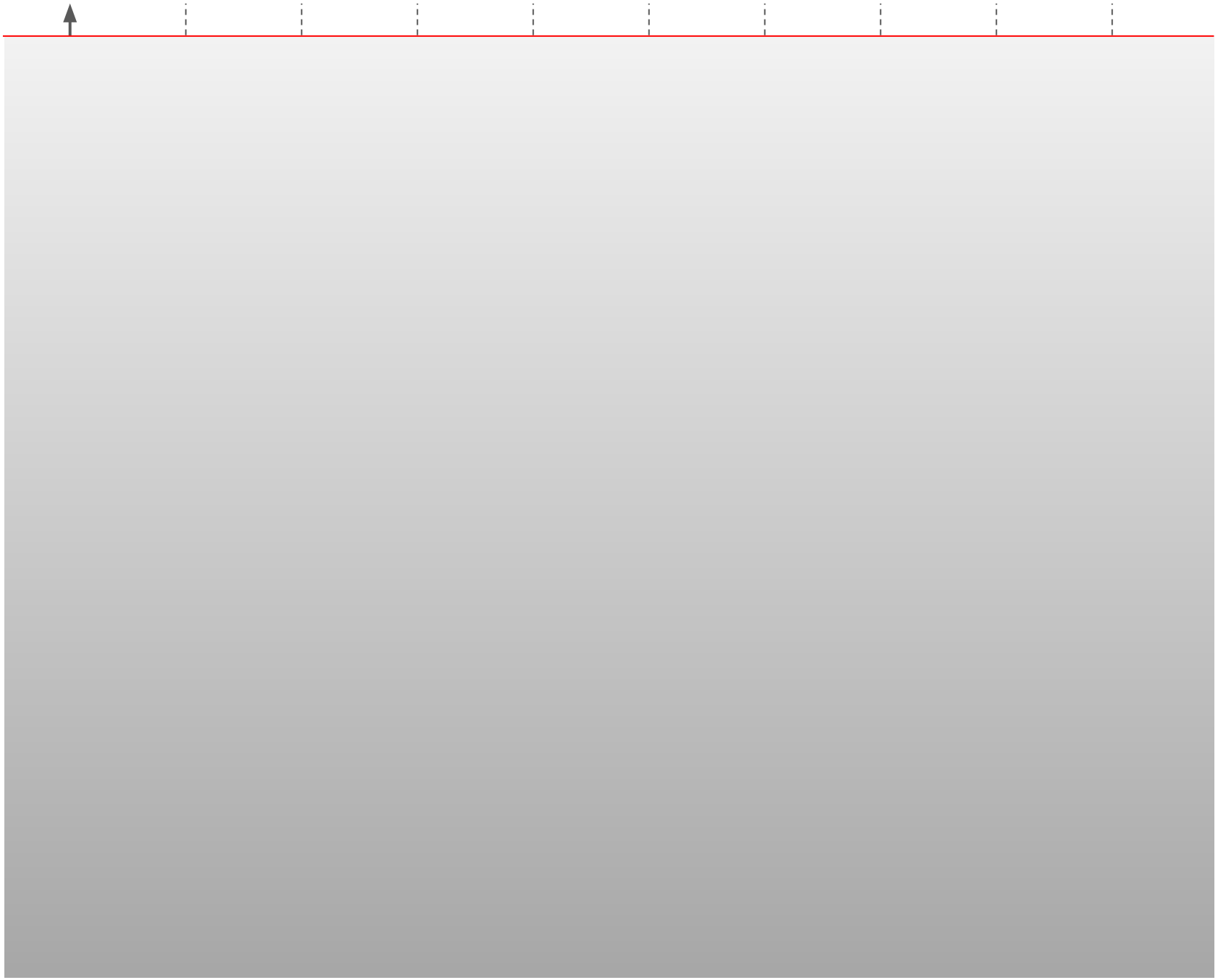
b) When we encounter a fence, we find the neighboring region that it will get merged with when the fence is removed the same way: it's the first active fence to the right.

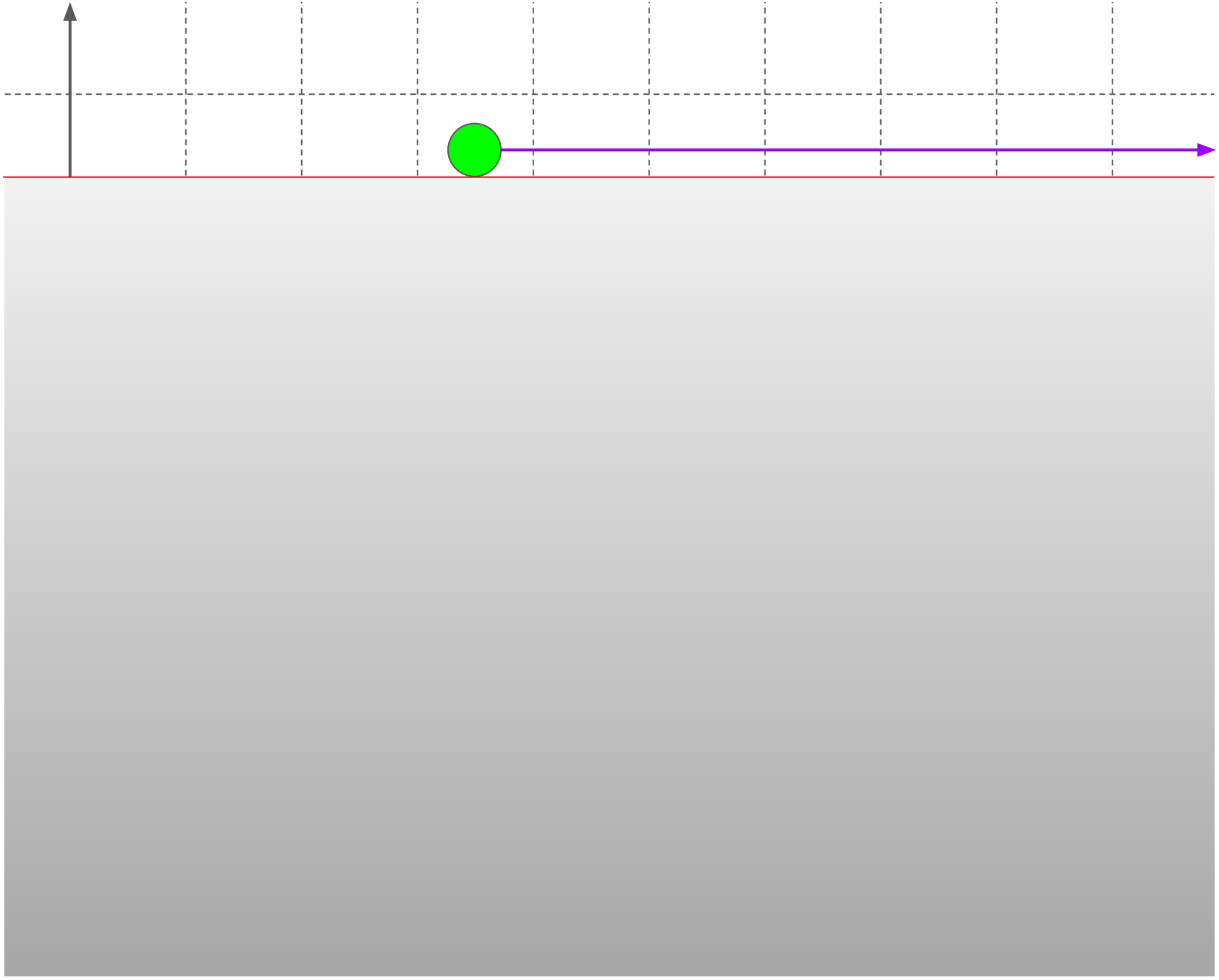
Sweep-line algorithm overview:

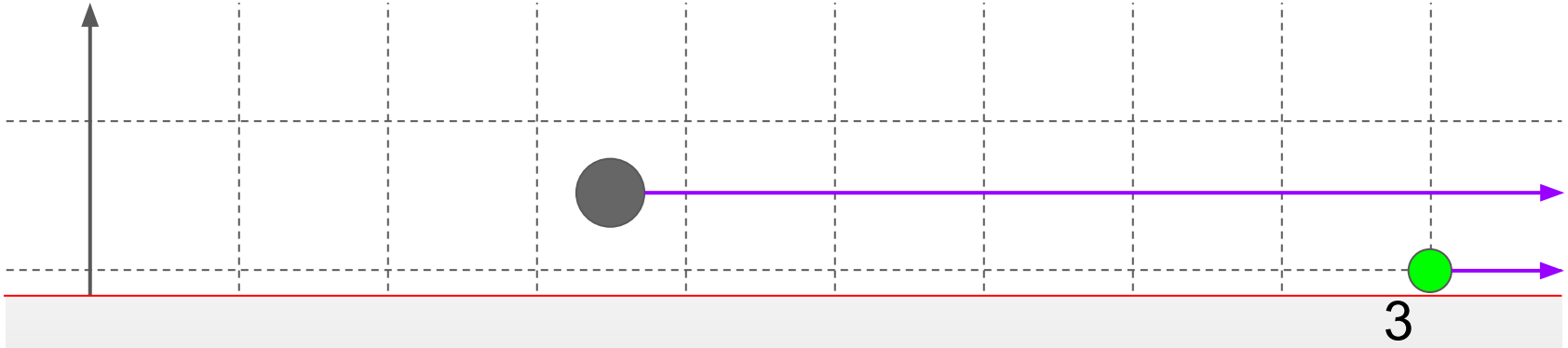
We process fence posts and buffalos in order of decreasing y coordinate.

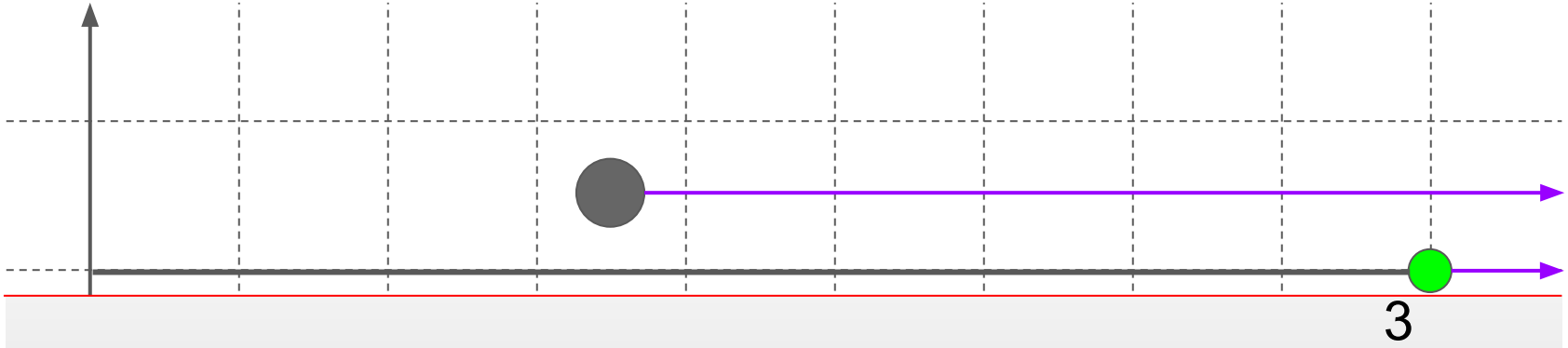
At each step we maintain a set of "active" vertical fences that have not yet hit another horizontal fence.

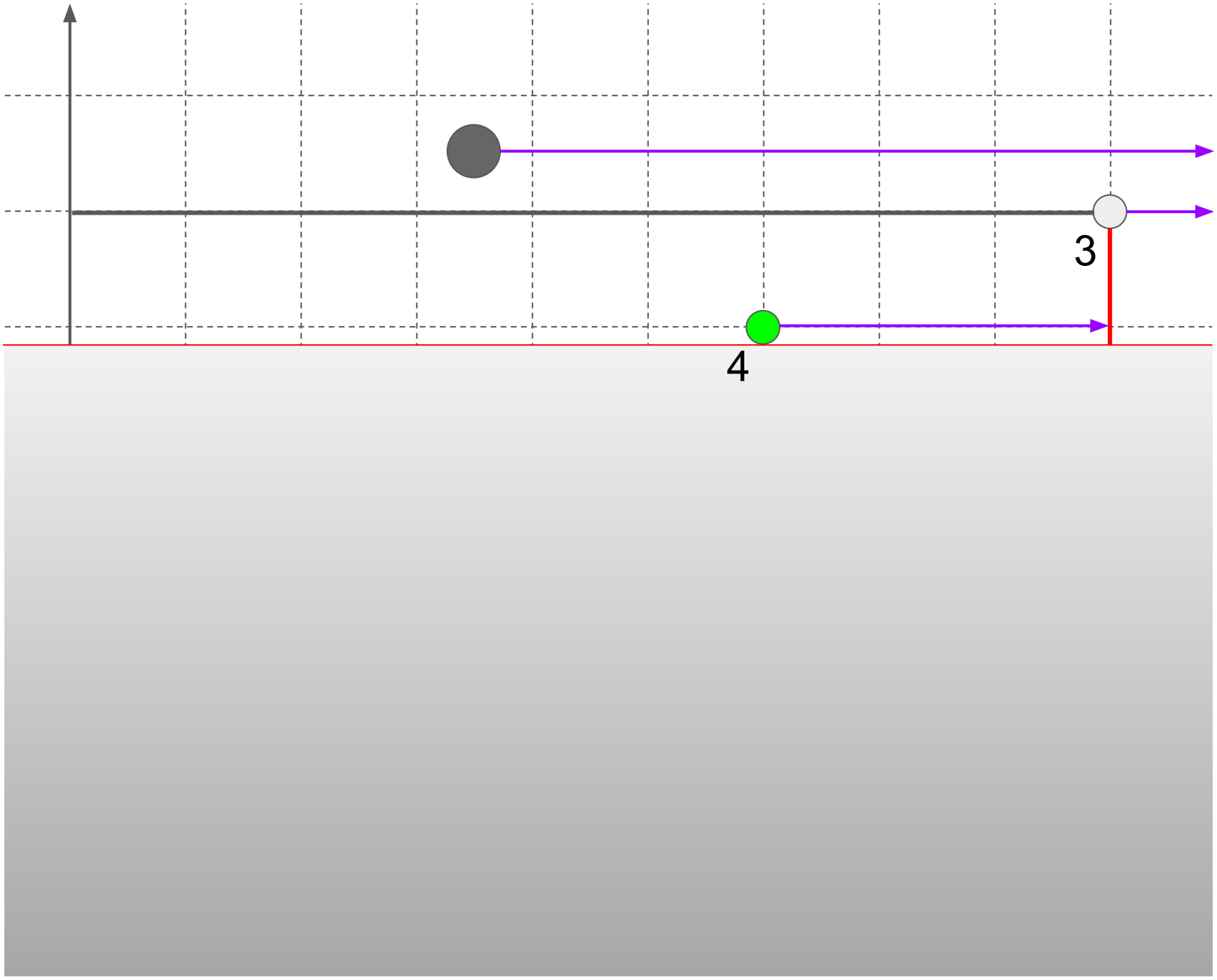
c) We also erect the horizontal fence starting from the fence post going to the left. Our fence will hit the first active fence to the left that has a smaller index (i.e. was erected prior to this fence). Other vertical fences we encounter along the way will, in turn, hit the horizontal fence we are building, so we remove them from the active set.

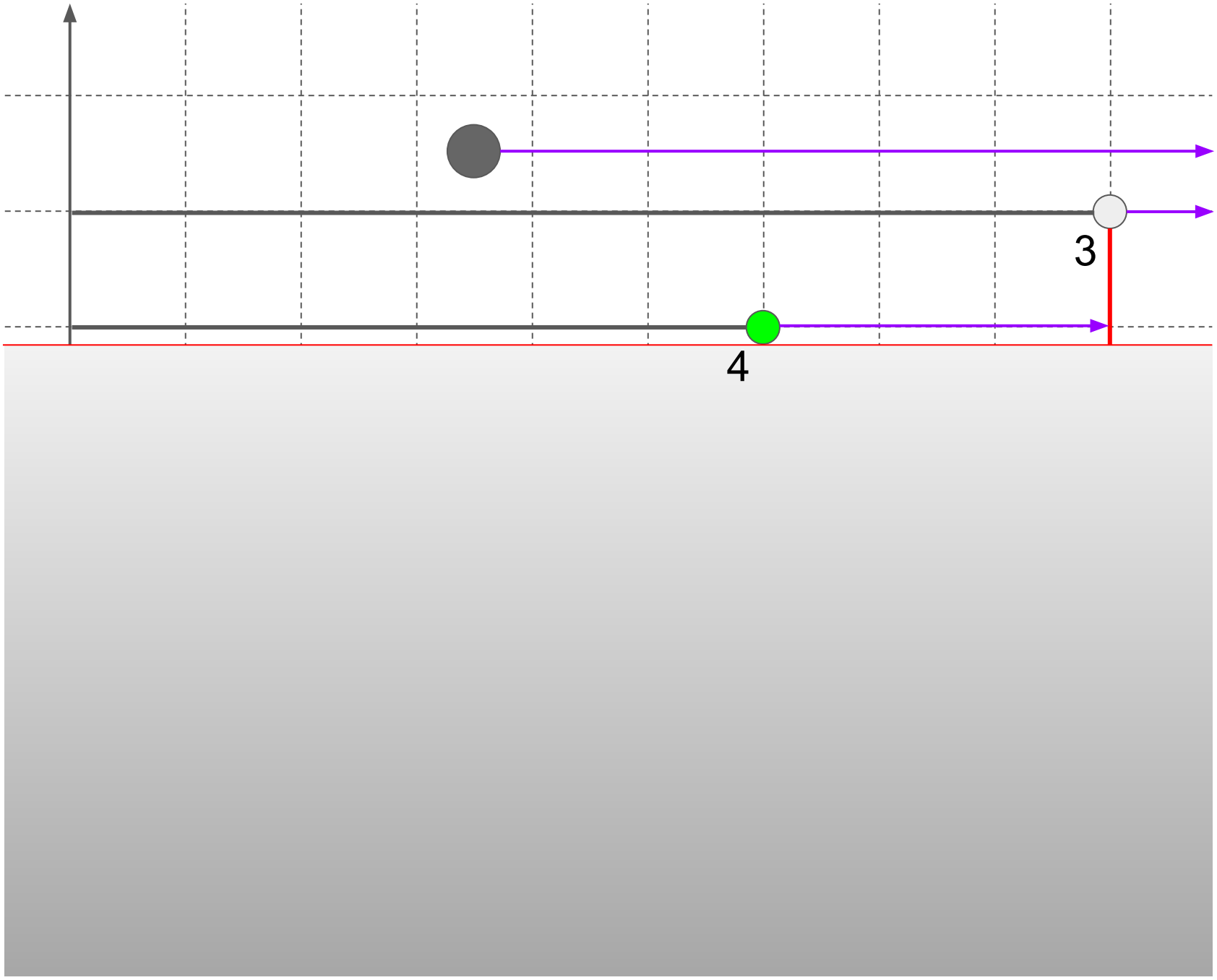


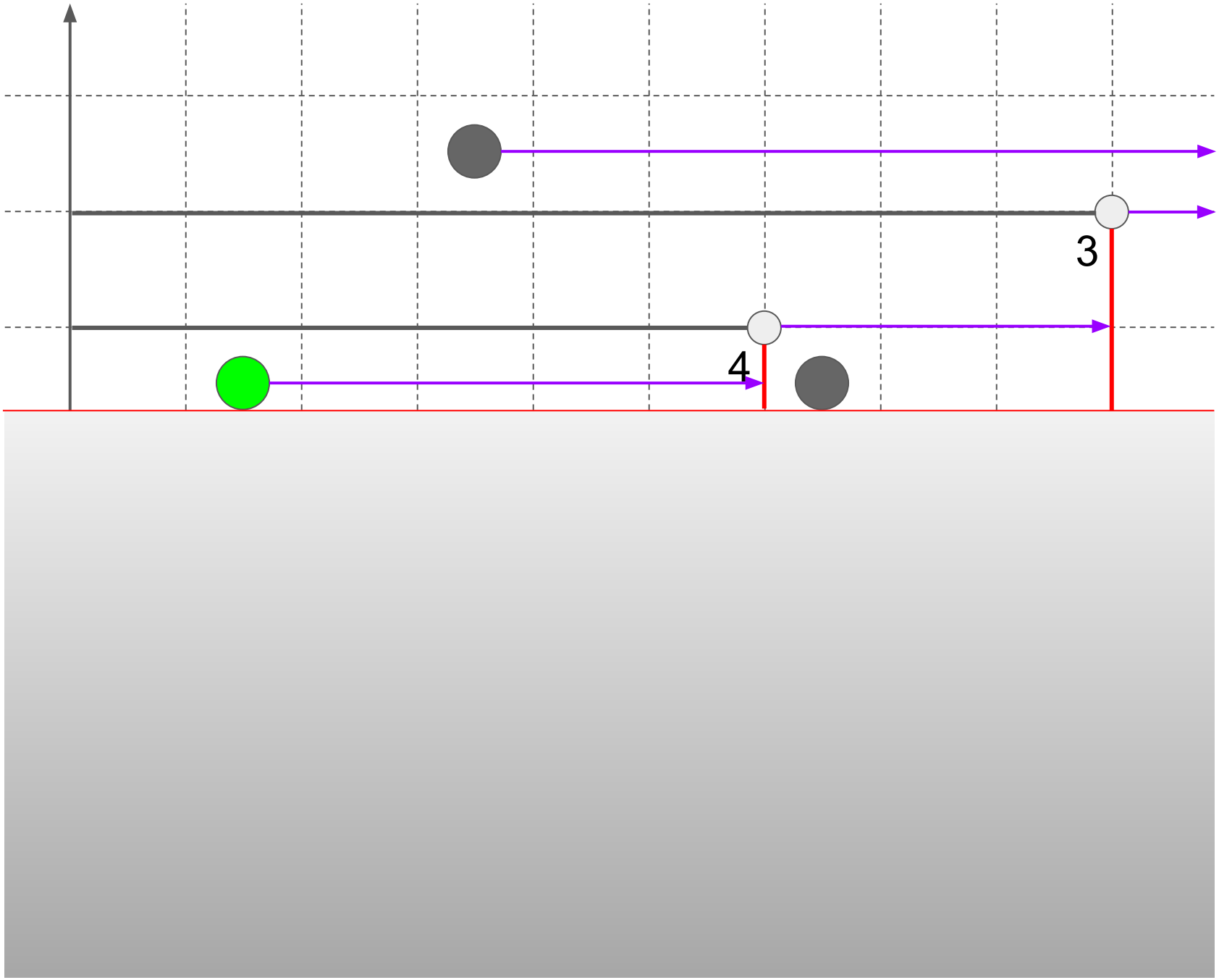


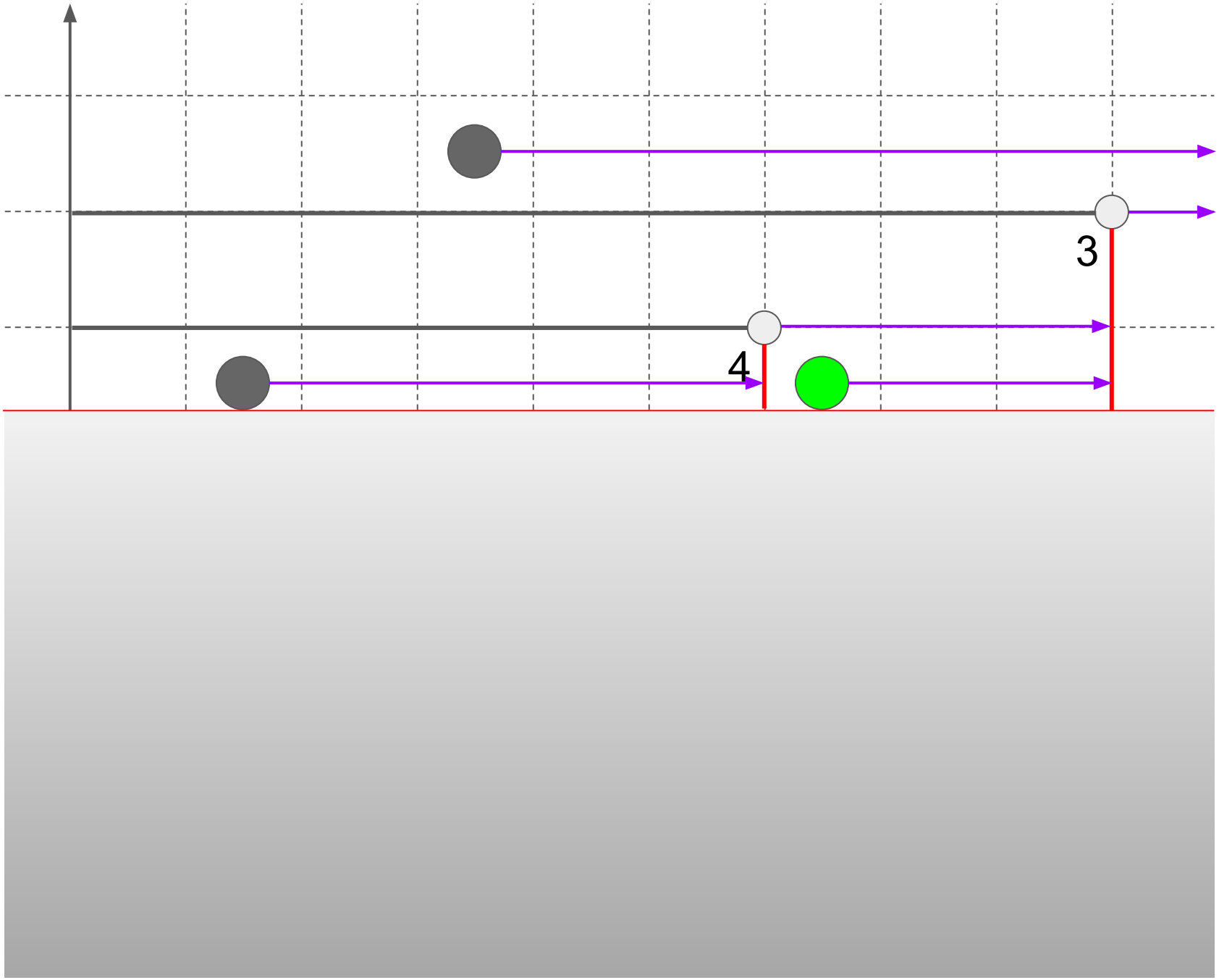

3

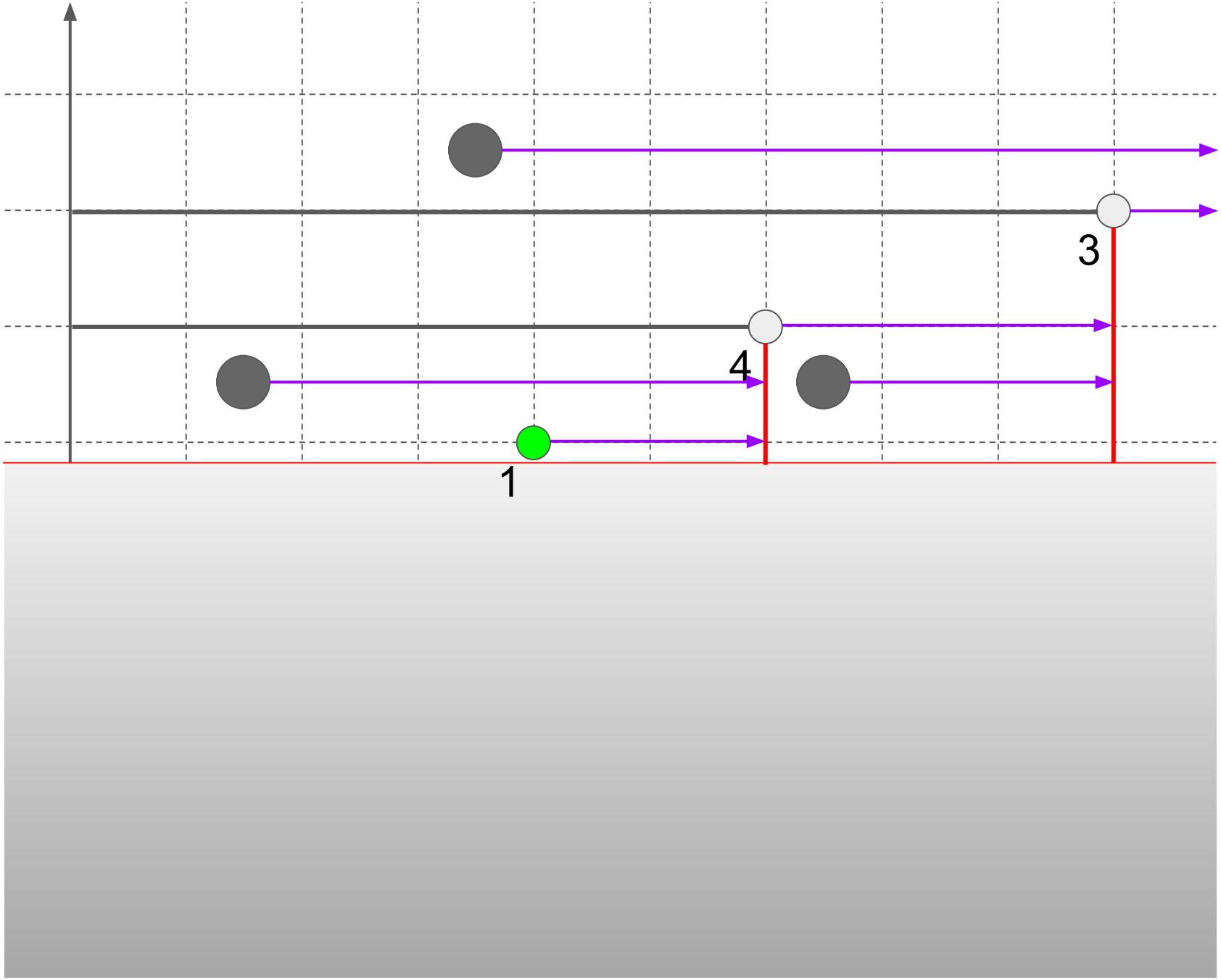

3

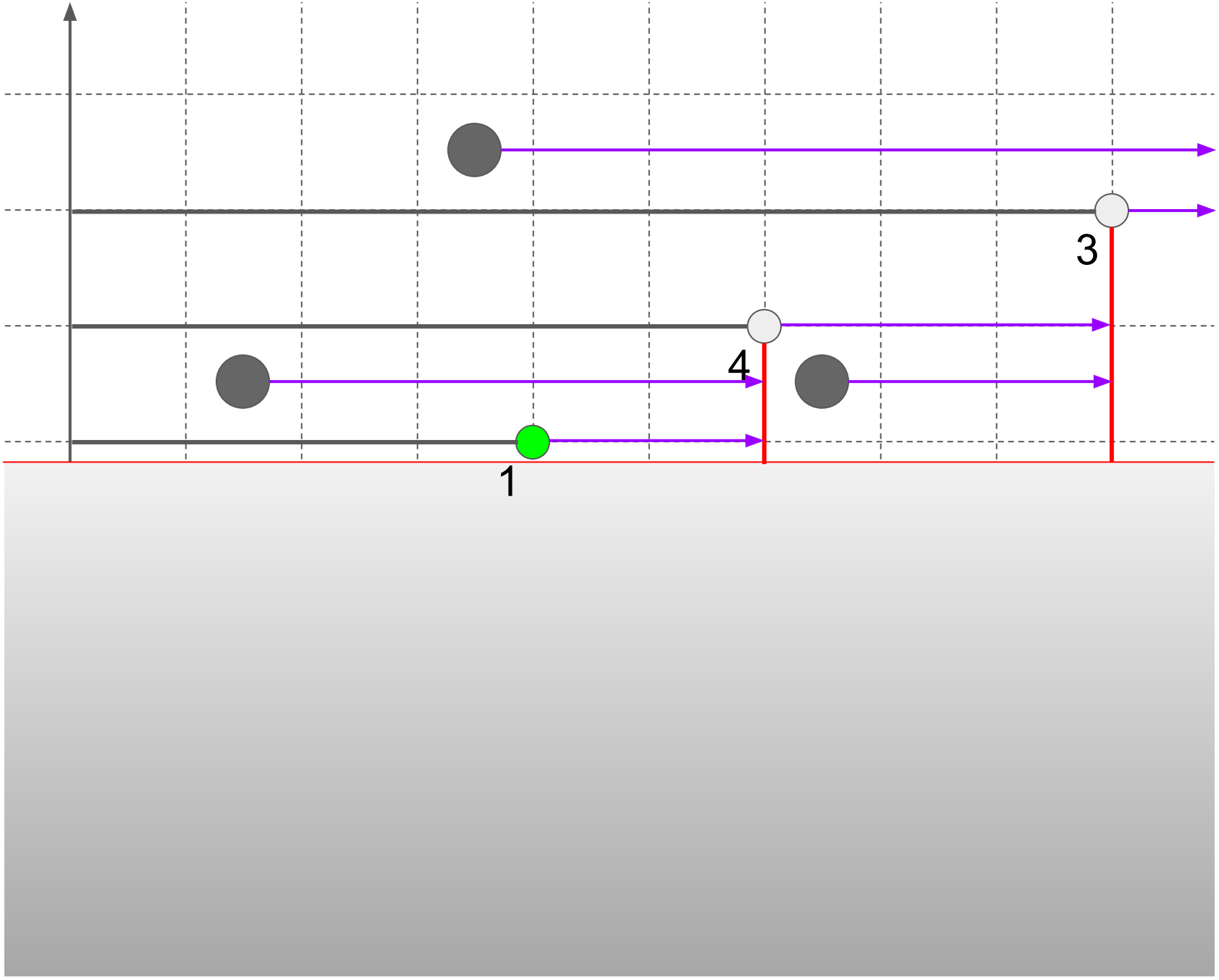

3
4

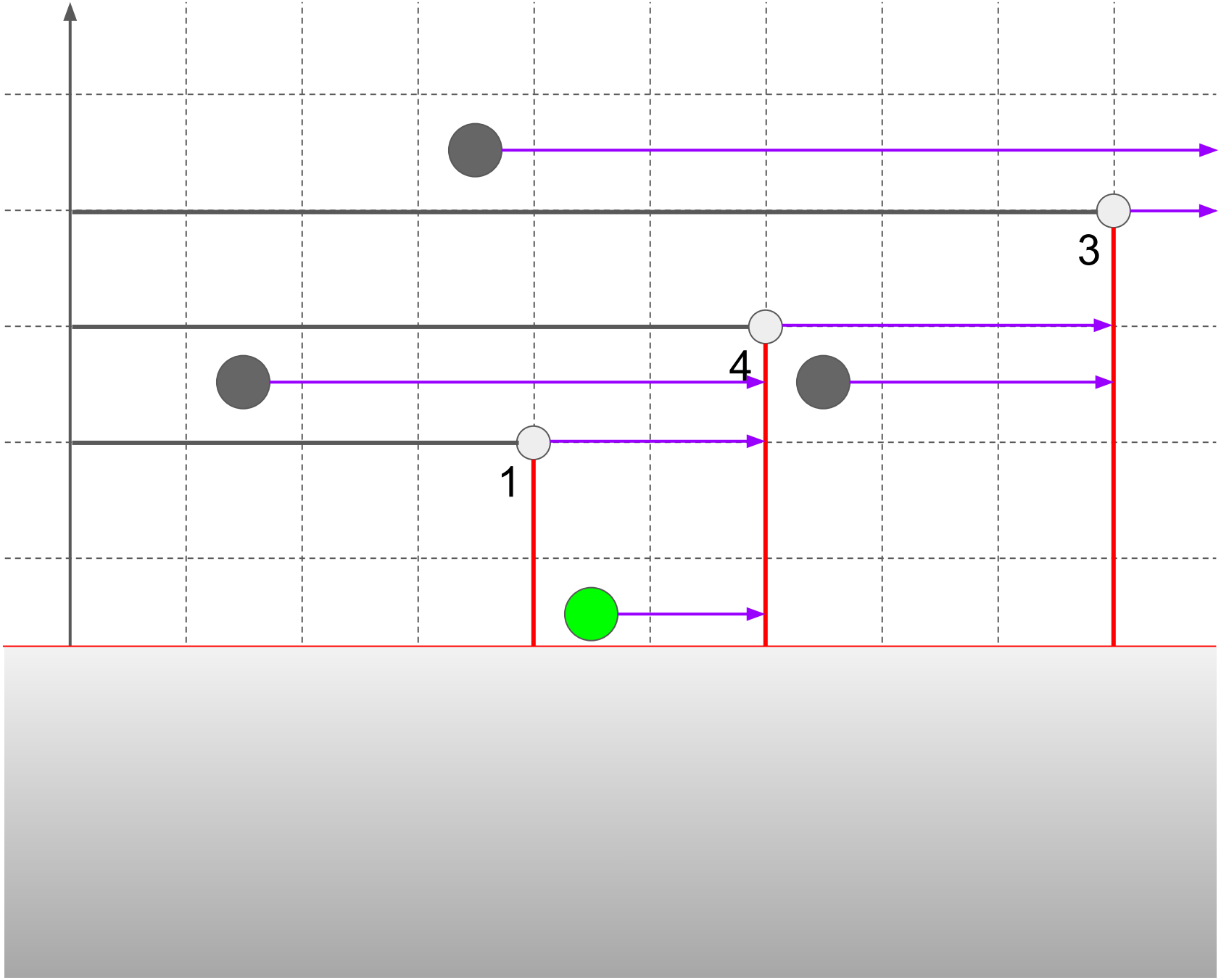

3
4

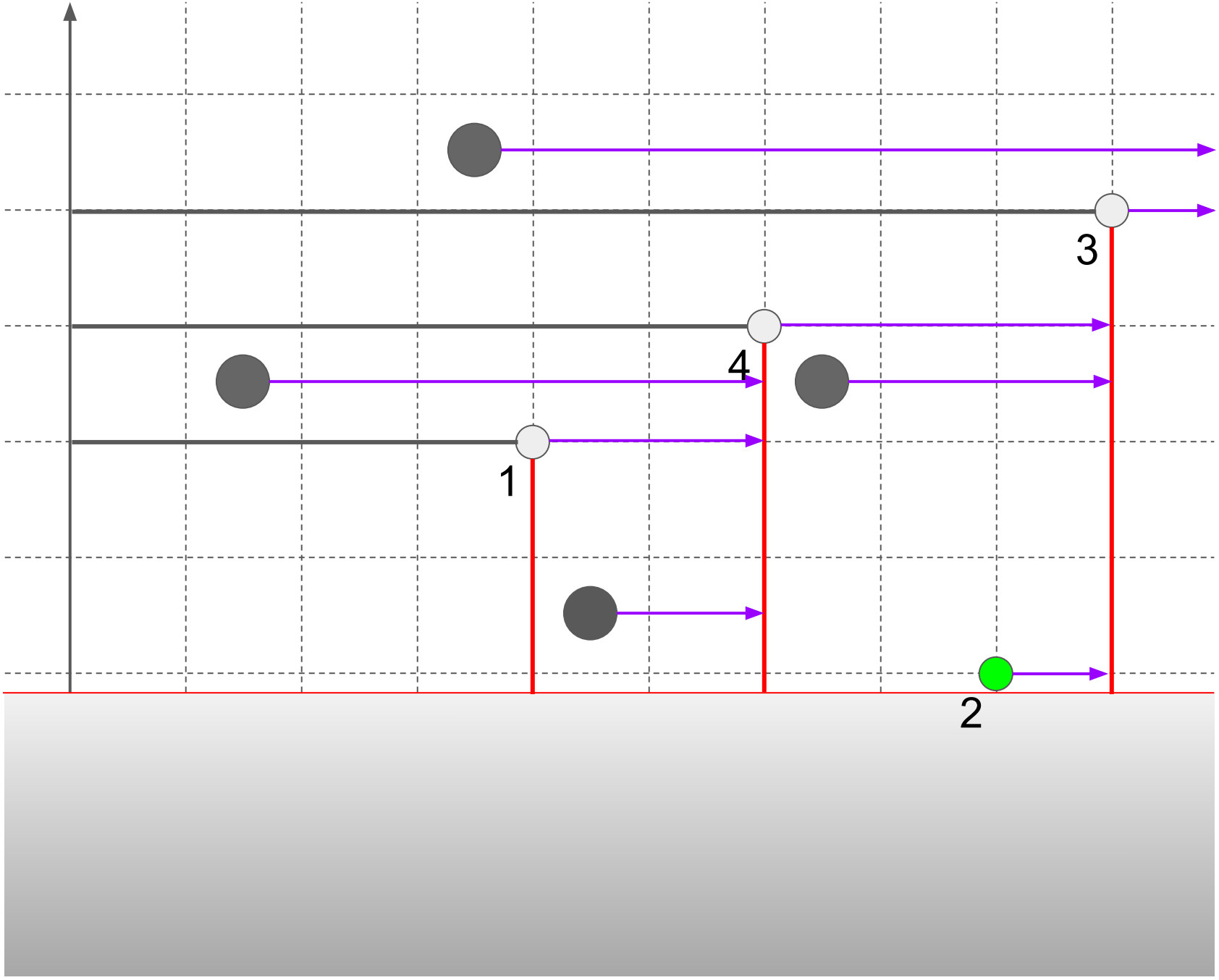

3
4

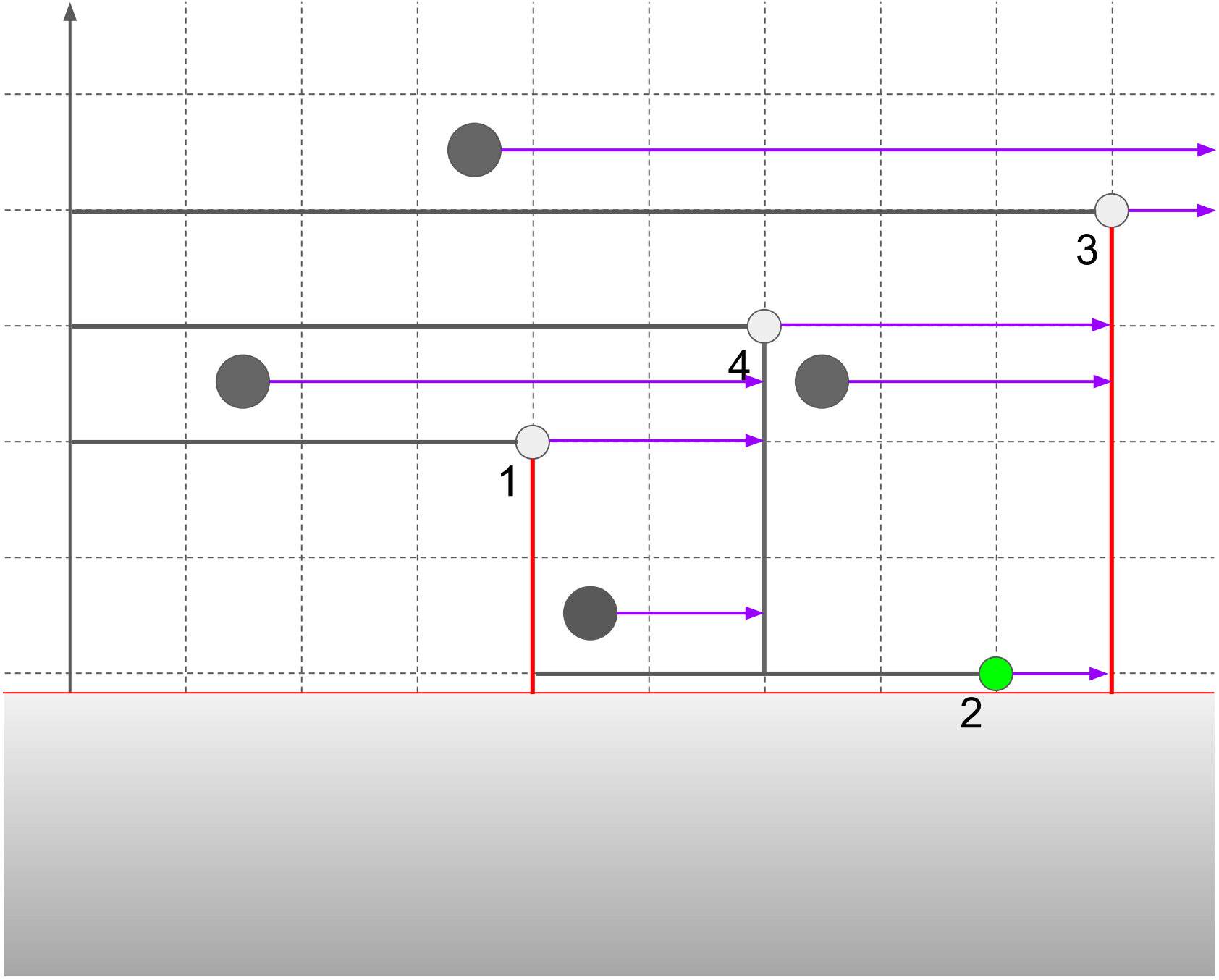

3
4

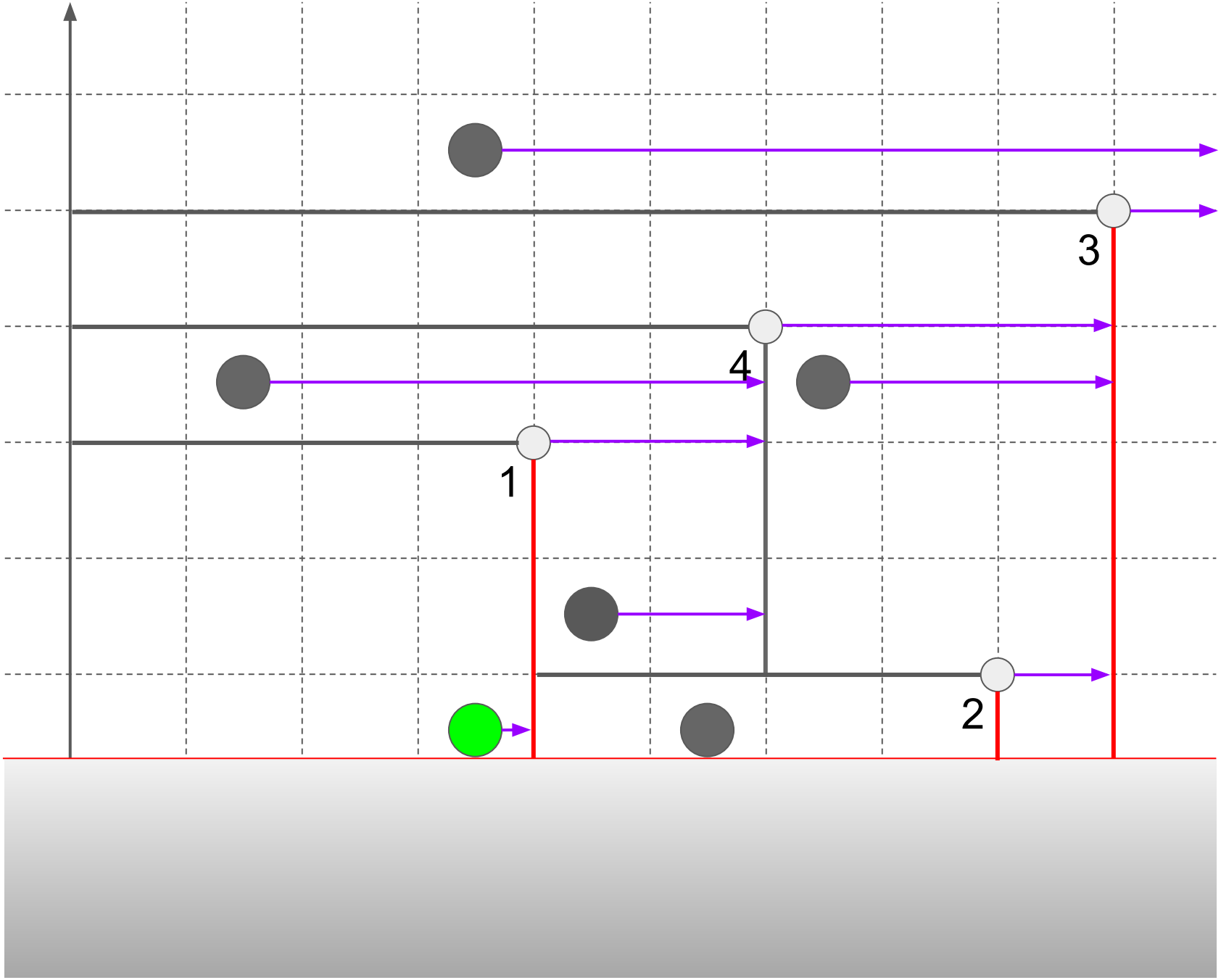

3
4
1

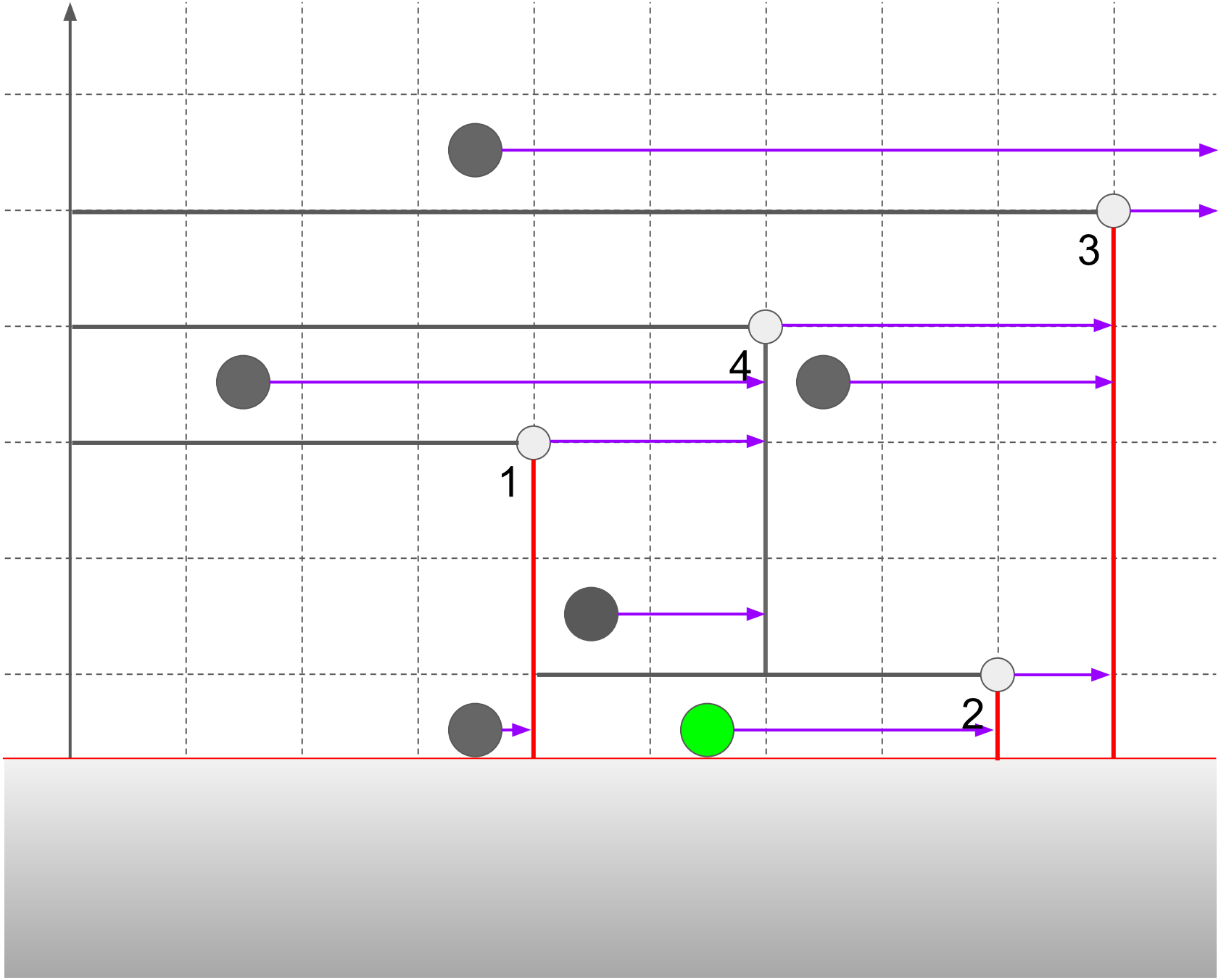

3
4
1

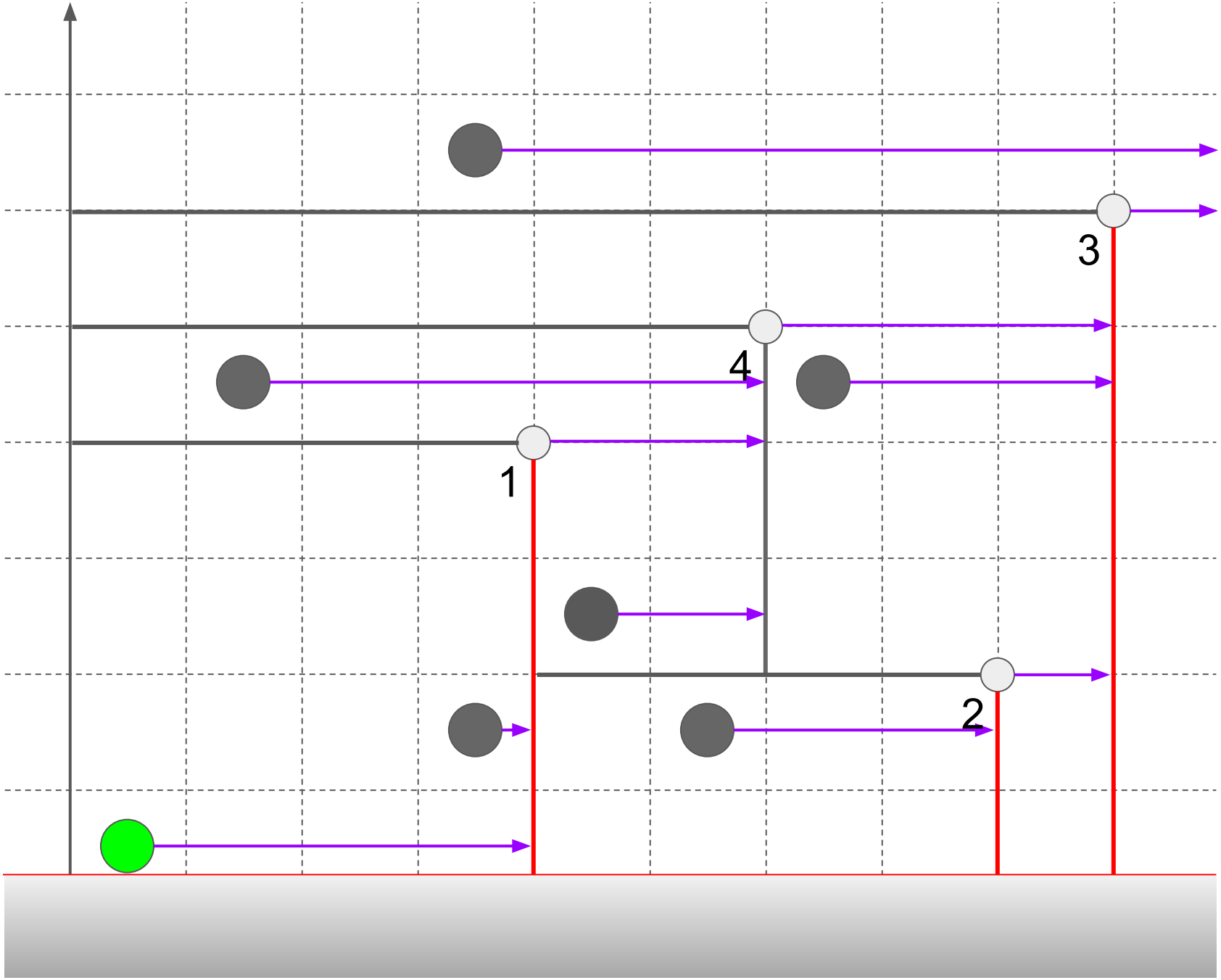

3
4
1

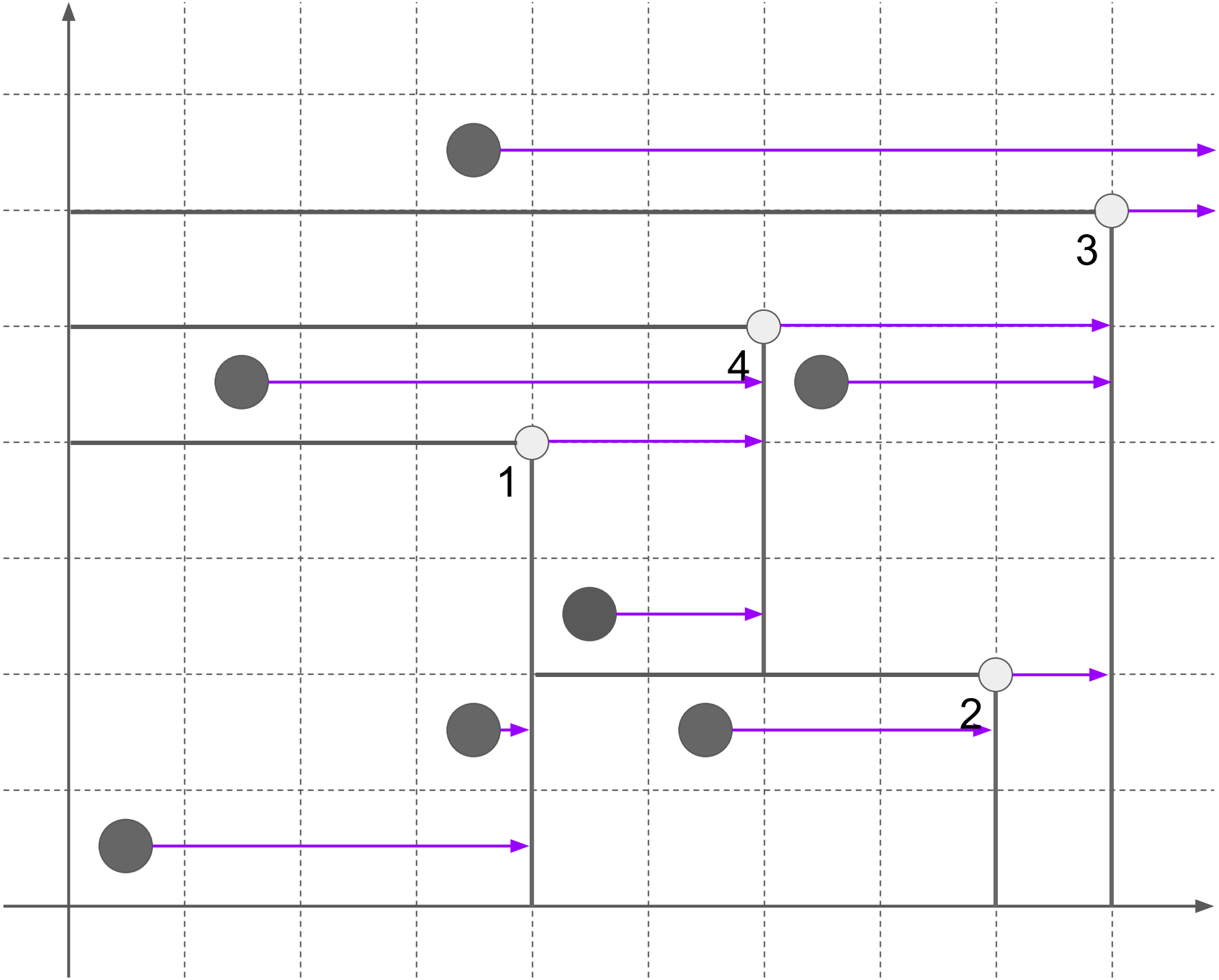

3
4
1
2


3
4
1
2


3
4
1
2


3
4
1
2


3
4
1
2


3
4
1
2

Complexity $O((N + M) \log (N + M))$

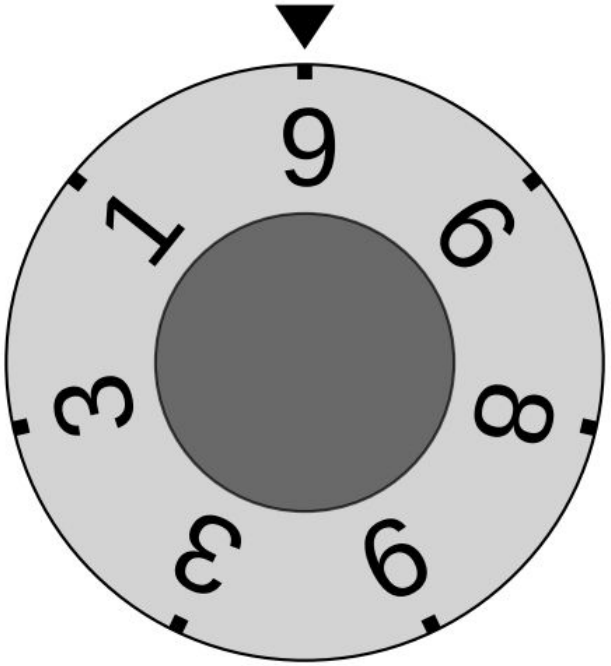
Problem K
Kitchen Knobs

Submits: 52
Accepted: at least 1

First solved by: UW1
University of Warsaw
(Dębowski, Radecki, Sommer)
01:24:54

Author: Goran Žužić, Luka Kalinovčić

Weird kitchen knobs with 7 non-zero digits. The power of a kitchen element is the number you get from reading the digits clockwise starting from the top position.



Power: 9689331

We have a sequence of N kitchen elements, and can rotate any consecutive subsequence of kitchen knobs by an arbitrary degree in a single step.

Find the smallest number of steps to get maximum power on each element.

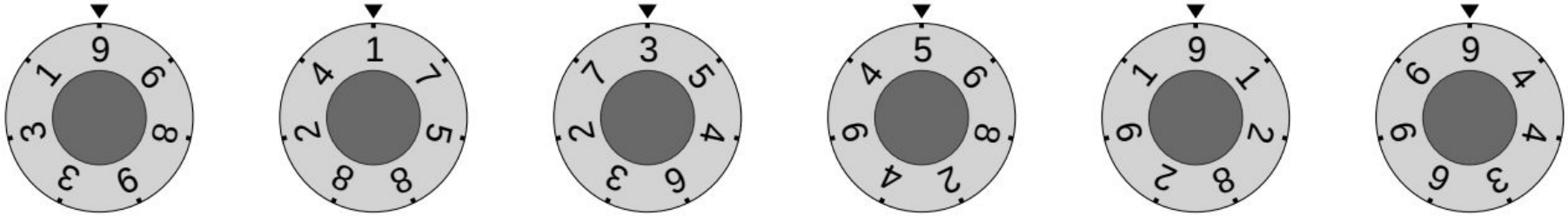
Because we have exactly 7 digits on each knob, every element either has:

a) all digits the same, in which case it's always at maximal power, or

b) exactly one position in which the maximal power is achieved.

We can pretend as if knobs of type a) didn't exist, and simplify the problem statement:

Given a sequence A with elements from [0, 6], find the smallest number of operations to make every element equal to 0. In a single operation we can add k to each number in an arbitrary subsequence of A (modulo 7).



0 3 6 5 5 5

+ 4

0 3 3 2 2 2

+ 4

0 0 0 2 2 2

+ 5

0 0 0 0 0 0

Let define another sequence B: B[i] = A[i] − A[i − 1]

A:		1	5	6	2	2	0	5	2	3
B:	1	4	1	3	0	5	5	4	1	4

+ 2

A:		1	5	**1**	**4**	**4**	**2**	**0**	**4**	3
B:	1	4	**3**	3	0	5	5	4	**6**	4

Observe what happens to sequence B as we apply the operation to sequence A.

Once again we can simplify the problem:

Given a set B with elements from [0, 6], find the smallest number of operations to make every element equal to 0. In a single operation we can add k to any number in the set and subtract k from any other number in the set (modulo 7).

1	4 -2	1	3	0	5	5 +2	4	1	4
1	2 -2	1	3	0	5 +2	0	4	1	4
1	0	1	3 -3	0	0	0	4 +3	1	4
1 +1	0	1 -1	0	0	0	0	0	1	4
2 +1	0	0	0	0	0	0	0	1 -1	4
3 +4	0	0	0	0	0	0	0	0	4 -4
0	0	0	0	0	0	0	0	0	0

Observation: Given any set of N numbers that add up to 0 (modulo 7), we can make all numbers zero in $N - 1$ operations.

In each operation take any two non-zero numbers from the set, and make one of them zero. If there are only two numbers left, it is guaranteed they will both become zero after the last operation.

Simplifying the problem even further:

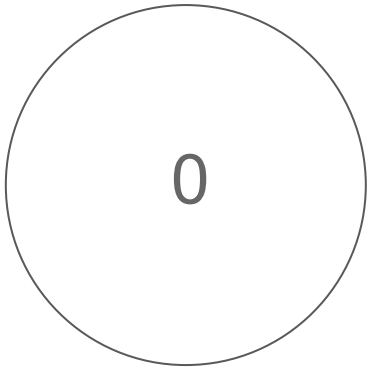
Given a set B with elements from [0, 6], group them into as many groups as possible such that the sum of each group is 0 (modulo 7).

B: 1 4 1 3 0 5 5 4 1 4

Simplifying the problem even further:

Given a set B with elements from [0, 6], group them into as many groups as possible such that the sum of each group is 0 (modulo 7).

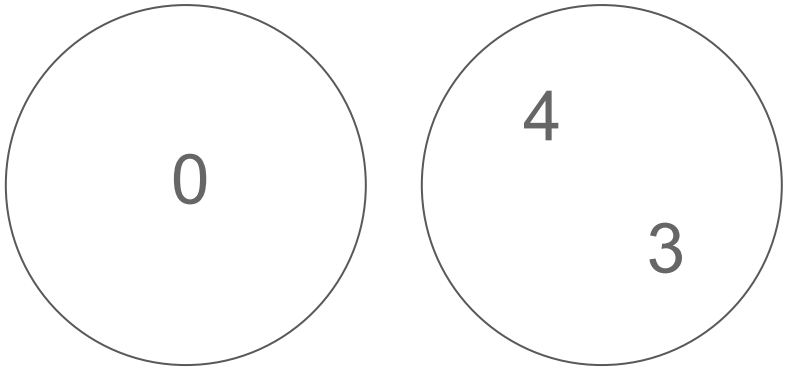
B: 1 4 1 3 5 5 4 1 4



Simplifying the problem even further:

Given a set B with elements from [0, 6], group them into as many groups as possible such that the sum of each group is 0 (modulo 7).

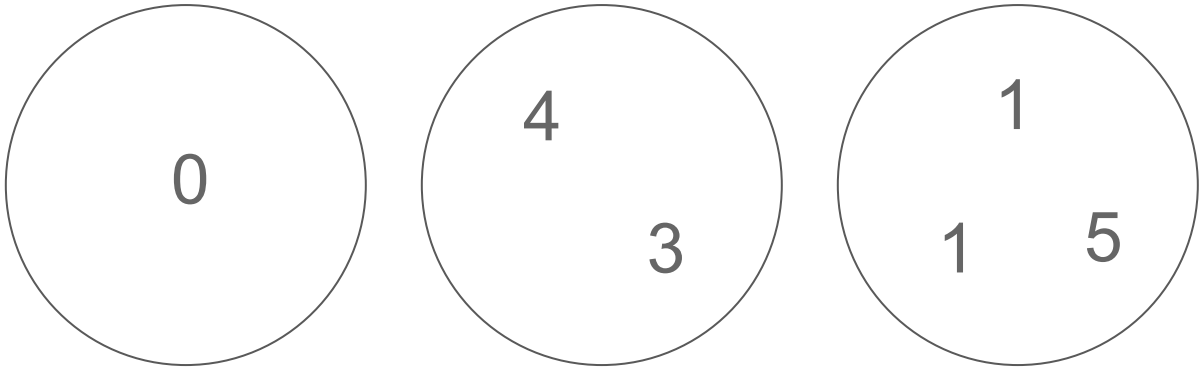
B: 1 1 5 5 4 1 4



Simplifying the problem even further:

Given a set B with elements from [0, 6], group them into as many groups as possible such that the sum of each group is 0 (modulo 7).

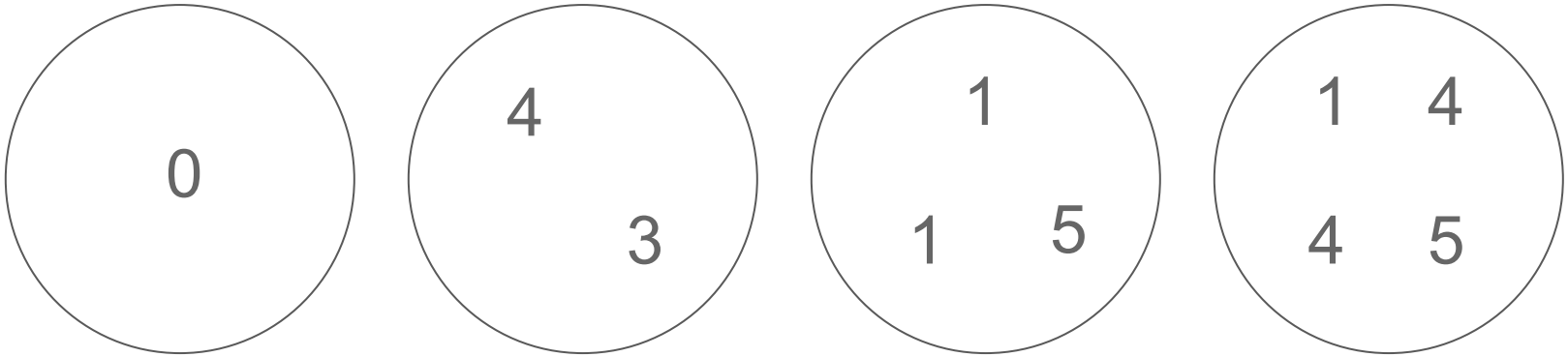
B: 1 5 4 4



Simplifying the problem even further:

Given a set B with elements from [0, 6], group them into as many groups as possible such that the sum of each group is 0 (modulo 7).

The solution is then N − number of groups = 10 - 4 = 6



To find the optimal grouping of numbers we start greedy:

1) As long as we have a zero in the set, make a group with a single zero in it.

2) As long as there is a pair of numbers that add up to 7 (1 and 6, 2 and 5, 3 and 4), make a group with these two numbers in it.

At this point the numbers in our set come from a set of at most three distinct integers: no zeros, either ones or sixes, either twos or fives, either threes or fours.

There exists a greedy O(N) strategy we could follow, but it's rather hard to find. Instead we may use a O(N^3) dynamic programming to complete the assignment.

Problem I
Intrinsic Interval

Submits: 42
Accepted: at least 1

First solved by: Jagiellonian 1
Jagiellonian University in Krakow
(Hlembotskyi, Stokowacki, Zieliński)
02:10:47

Author: Gustav Matula

An interval of the permutation is a consecutive subsequence consisting of consecutive numbers.

An interval of the permutation is a consecutive subsequence consisting of consecutive numbers.

An interval of the permutation is a consecutive subsequence consisting of consecutive numbers.

An interval of the permutation is a consecutive subsequence consisting of consecutive numbers.

2 3 1 6 4 7 5 8

An interval of the permutation is a consecutive subsequence consisting of consecutive numbers.

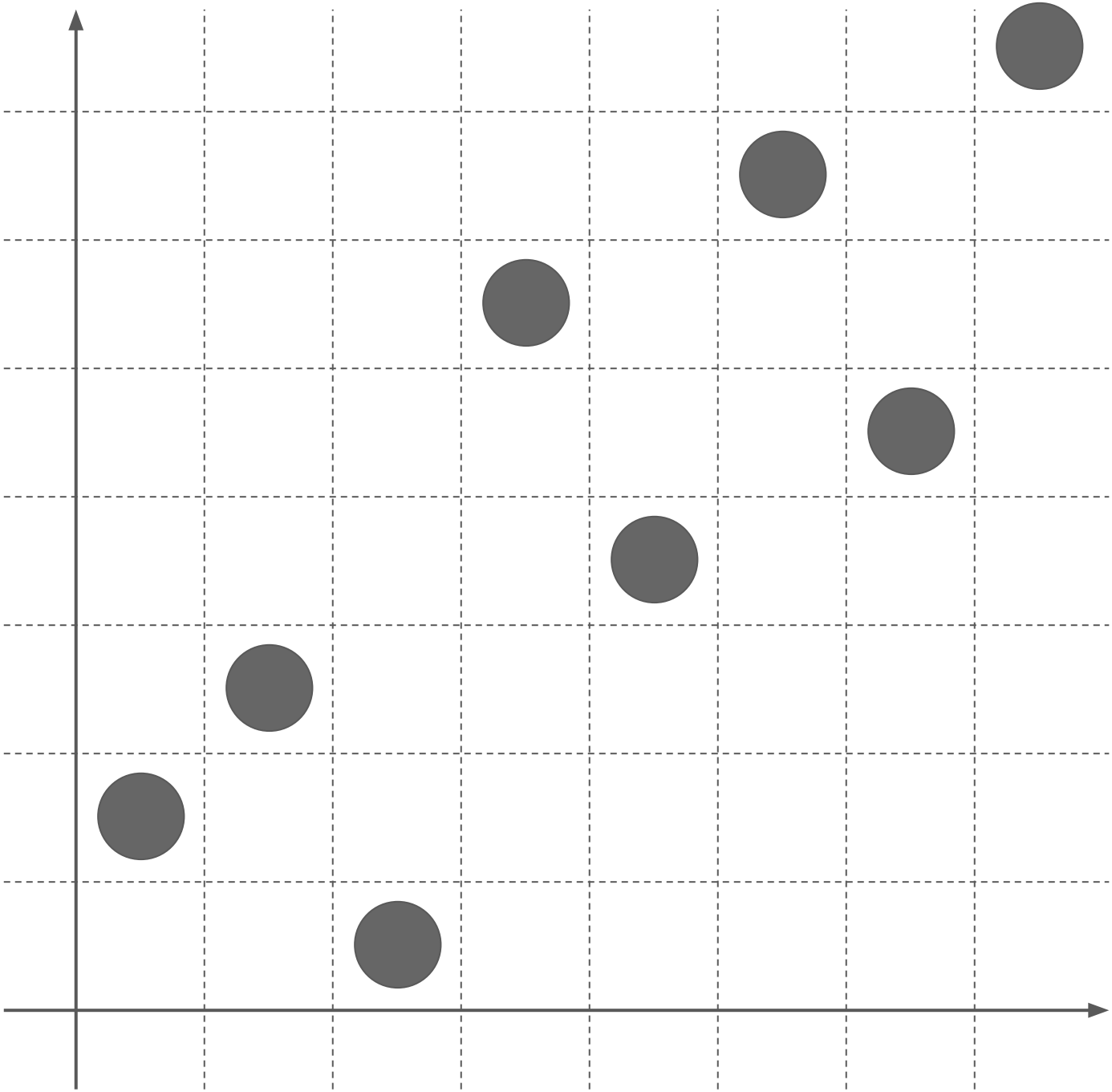
An interval of the permutation is a consecutive subsequence consisting of consecutive numbers.

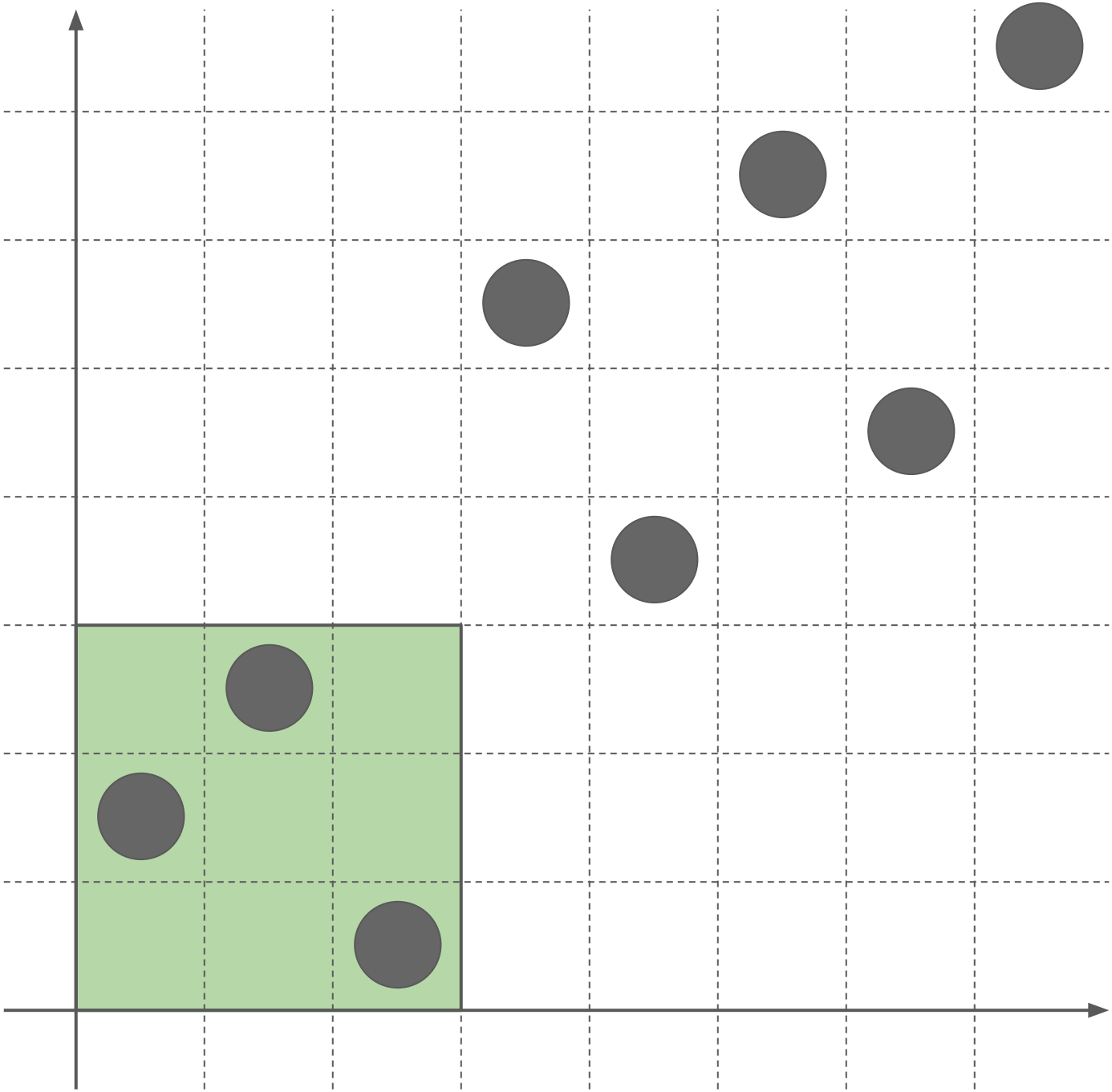
For a given subsequence we need to find the shortest enclosing interval.

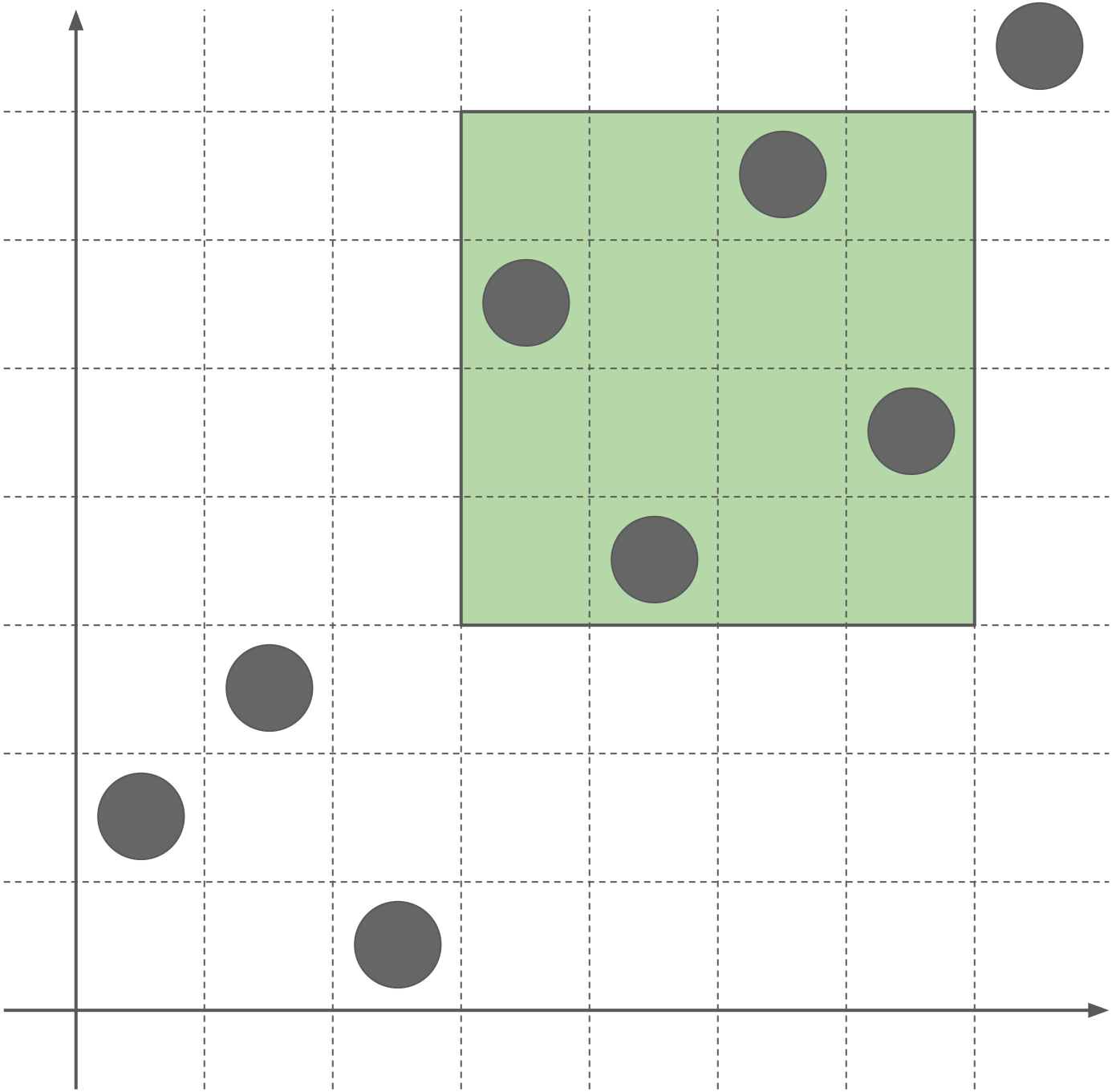
An interval of the permutation is a consecutive subsequence consisting of consecutive numbers.

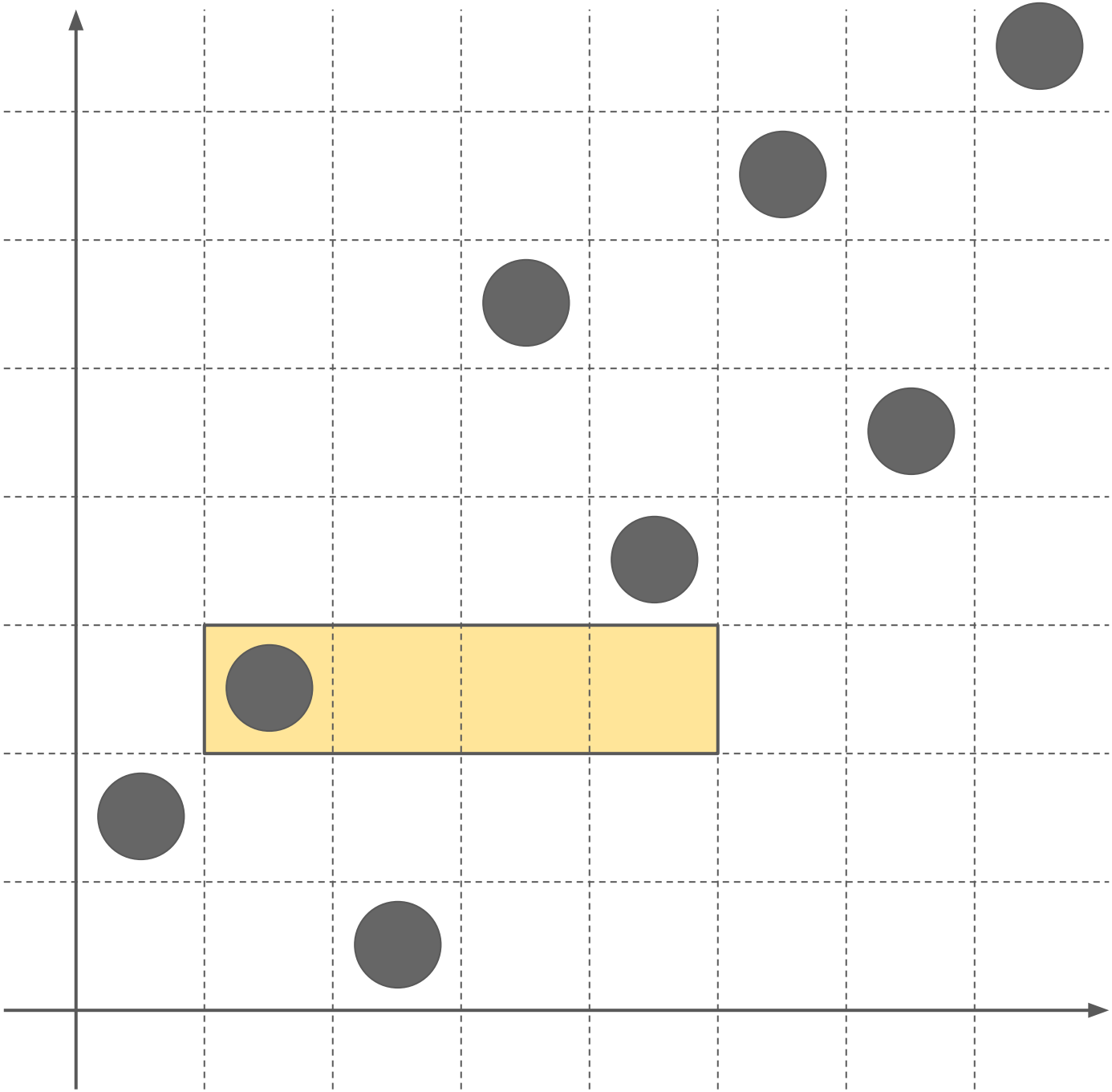
For a given subsequence we need to find the shortest enclosing interval.

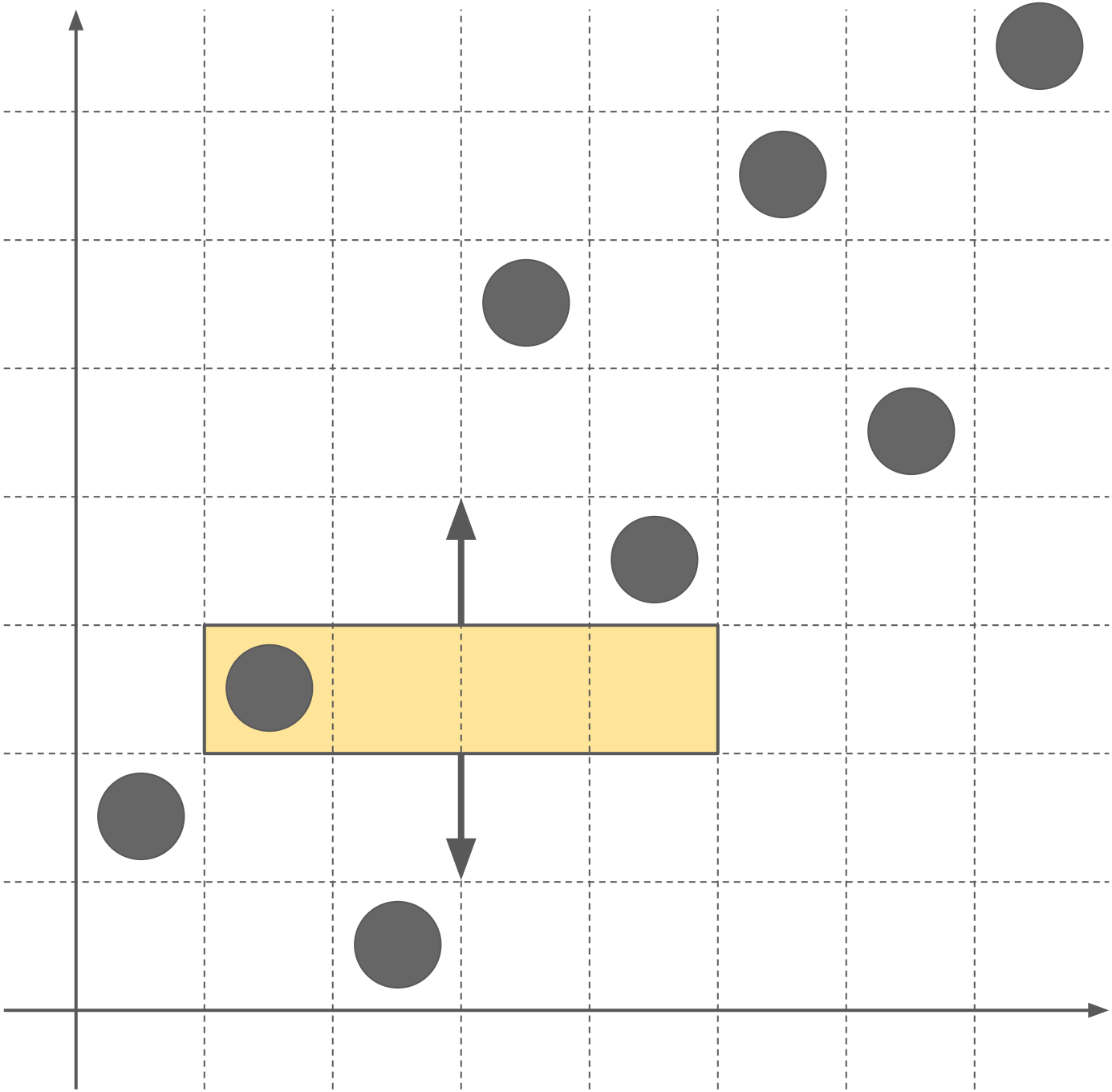
To see how we could expand the subsequence into the shortest enclosing interval, let's visualize the permutation in two dimensions.

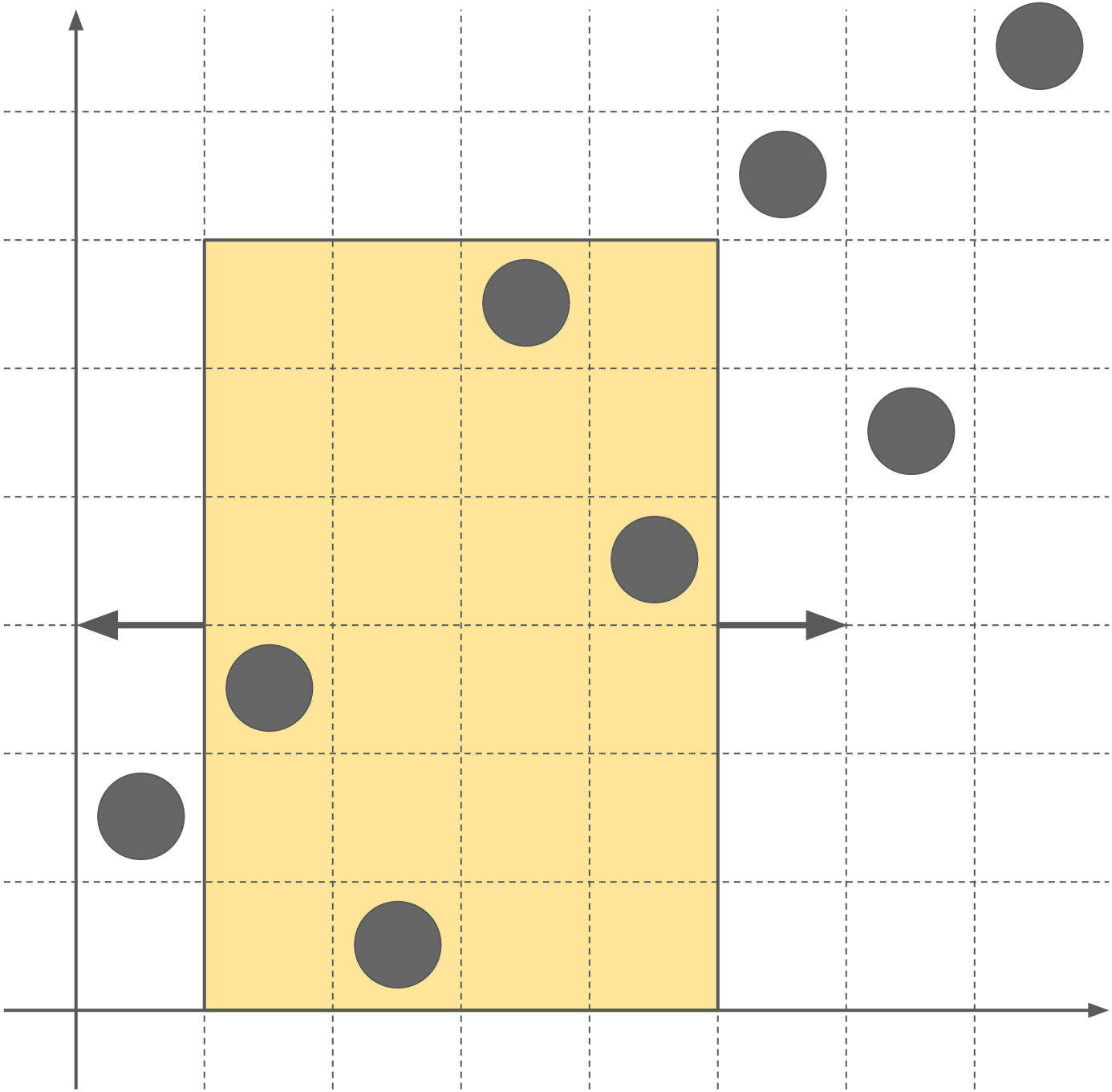


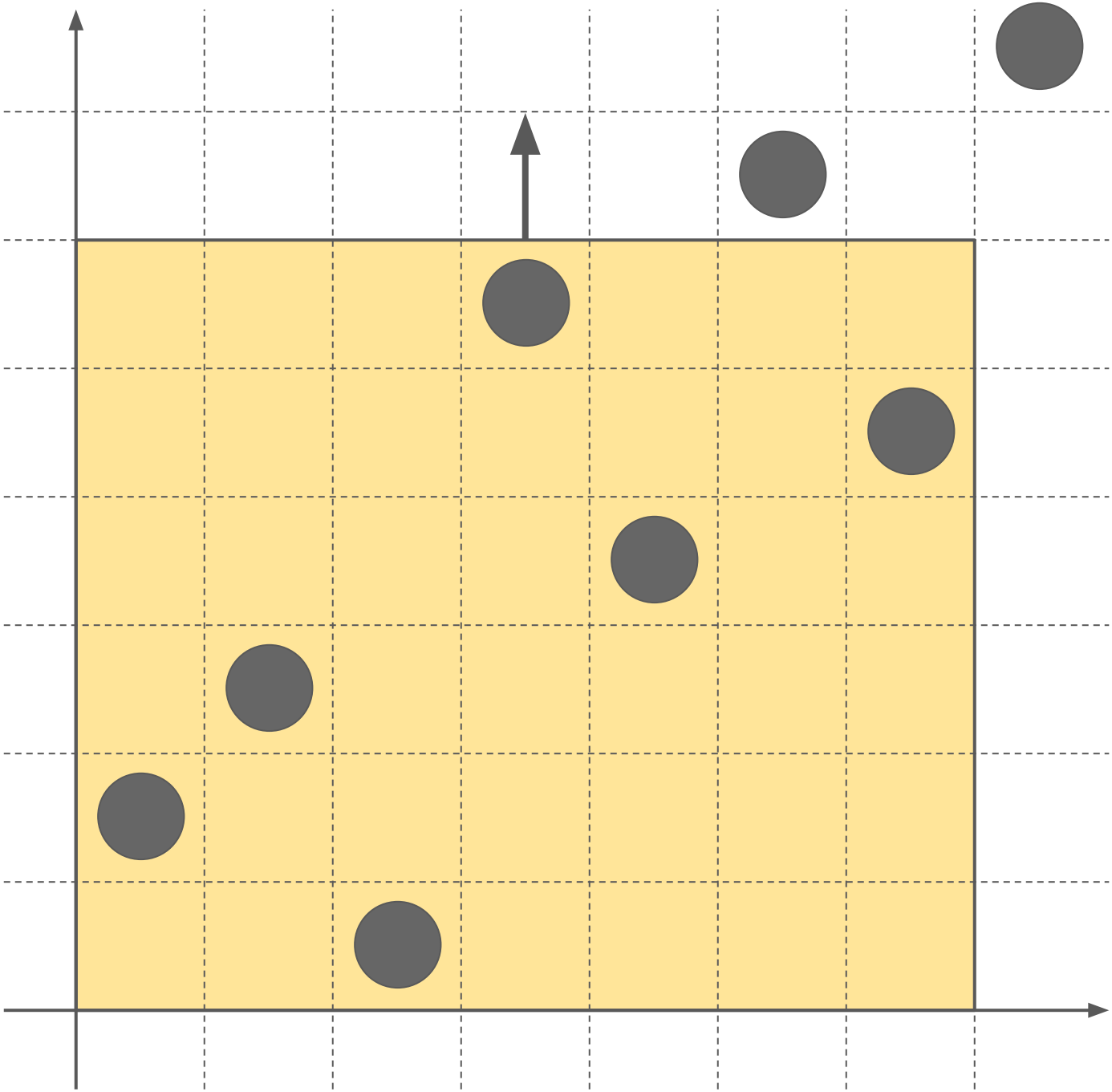


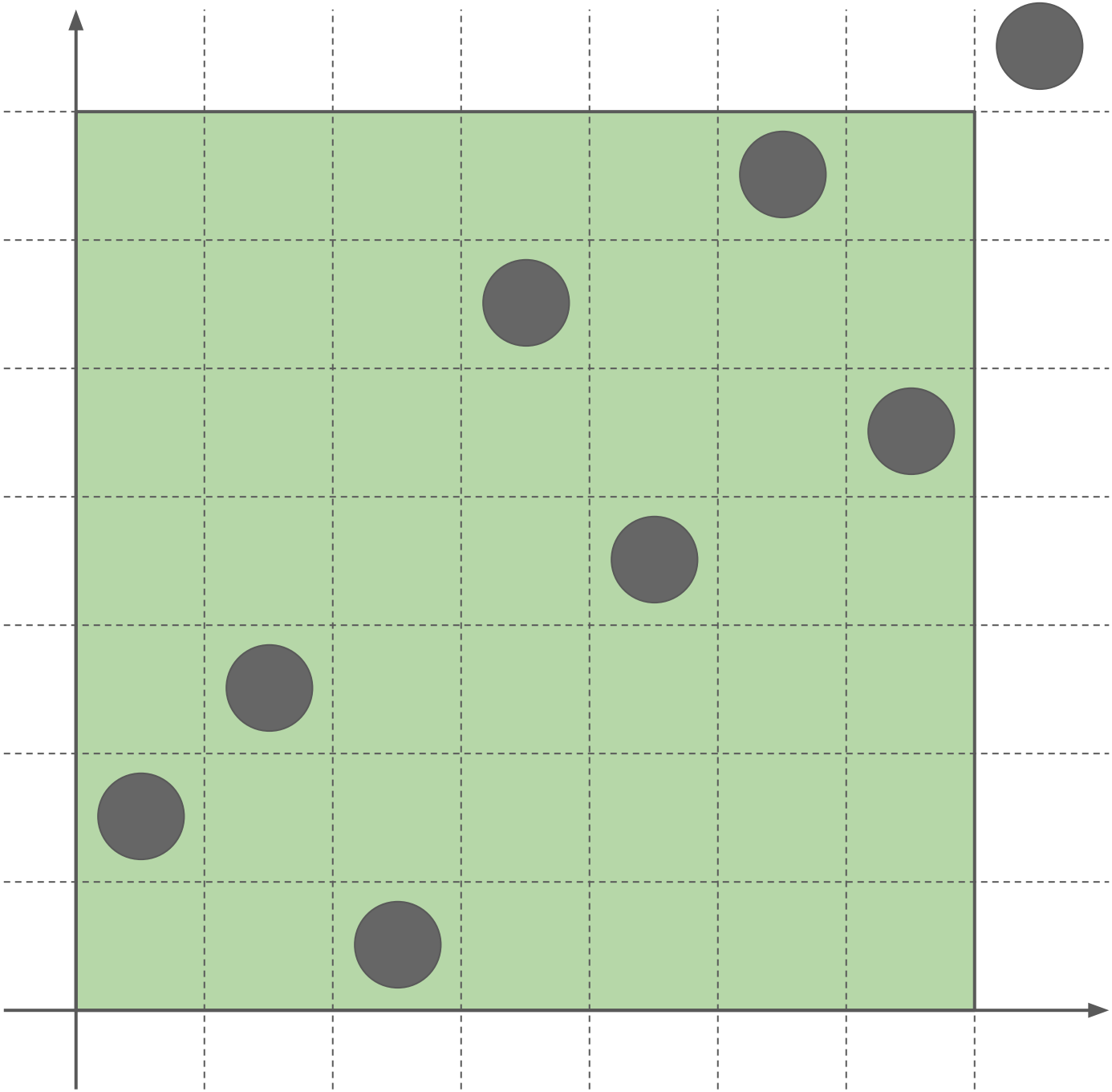












With careful implementation of the algorithm, it is possible to expand a subsequence [a, b] to an enclosing interval [x, y] in O(|y - x| - |b - a|).

However, that's too slow for this problem.

Instead, we'll develop divide and conquer algorithm to answer all queries at once.

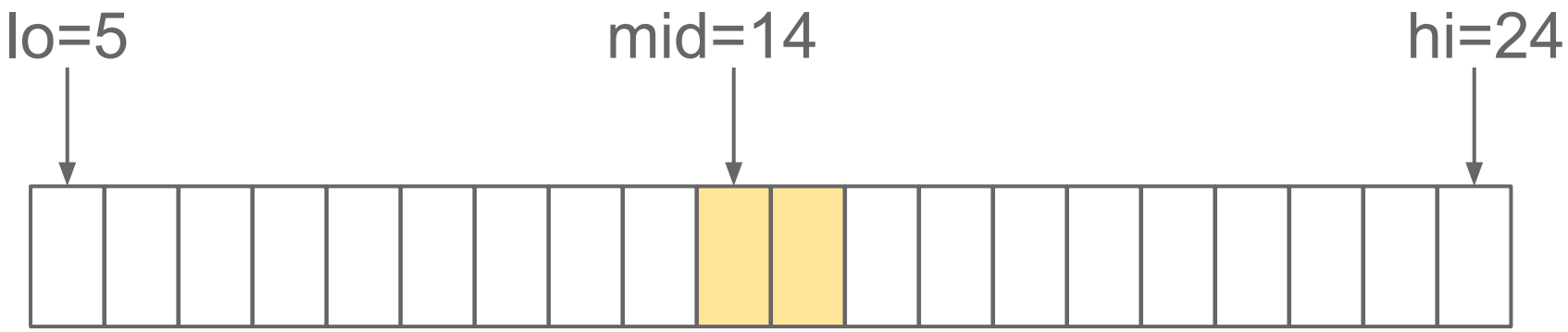
We initialize the result for each query with interval [1, n] and then we'll try to improve it.

Improve(queries, lo, hi) will try to improve each query in queries by considering intervals completely within [lo, hi] window.

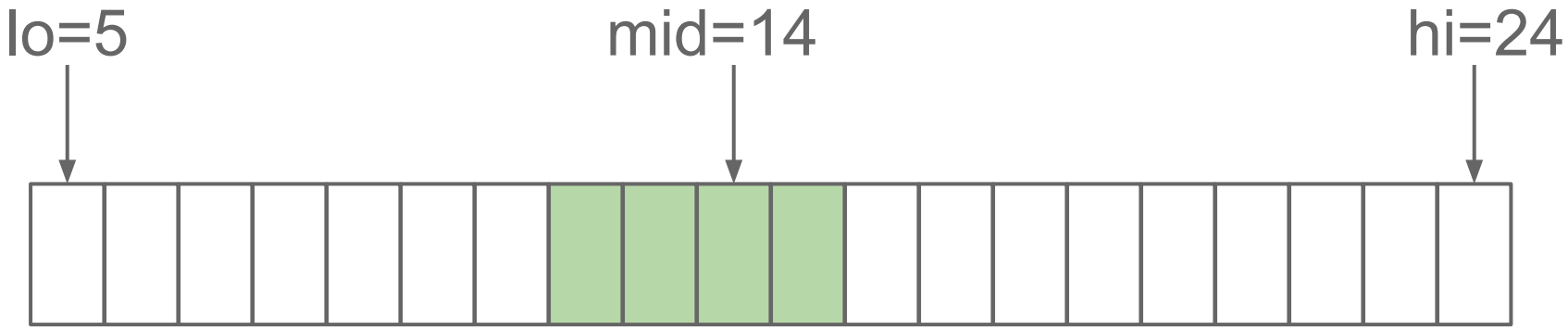
```
Improve(queries, lo, hi):
  if lo == hi: return
  mid = (lo + hi) / 2
  Improve([q in queries where q.b <= mid], lo, mid)
  Improve([q in queries where q.a > mid], mid + 1, hi)

  ImproveViaMid(queries, lo, mid, hi)
```

ImproveViaMid considers all intervals that contain [mid, mid + 1], and are within the [lo, hi] to improve provided queries.

A query participates in O(log(N)) ImproveViaMid calls.

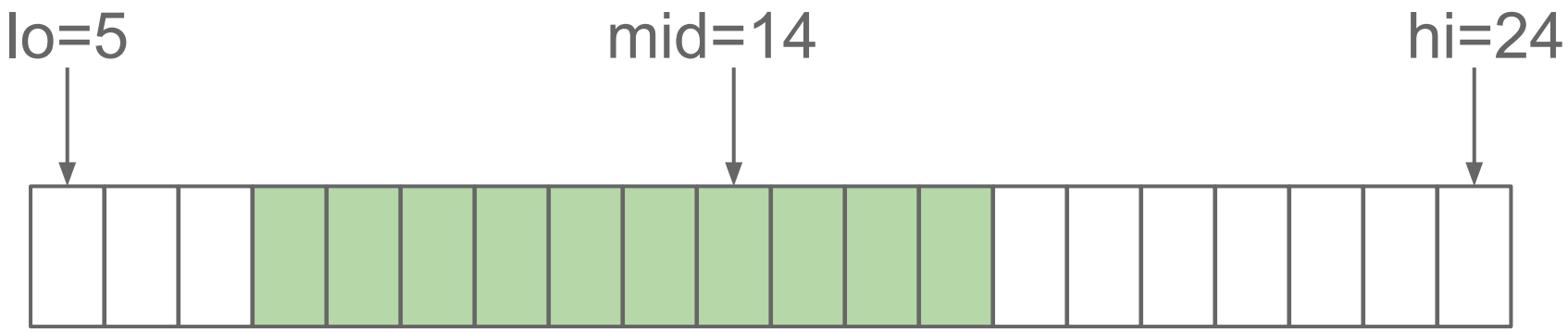


Starting from subsequence [mid, mid + 1], we expand it to the left, storing all intervals we encounter until we exit the [lo, hi] window.



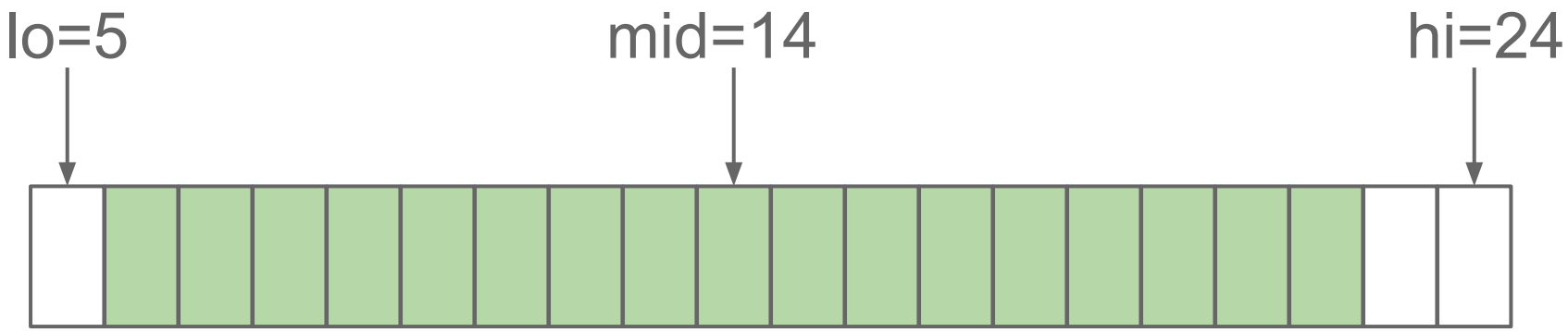
Left intervals: [12, 15]

Starting from subsequence [mid, mid + 1], we expand it to the left, storing all intervals we encounter until we exit the [lo, hi] window.



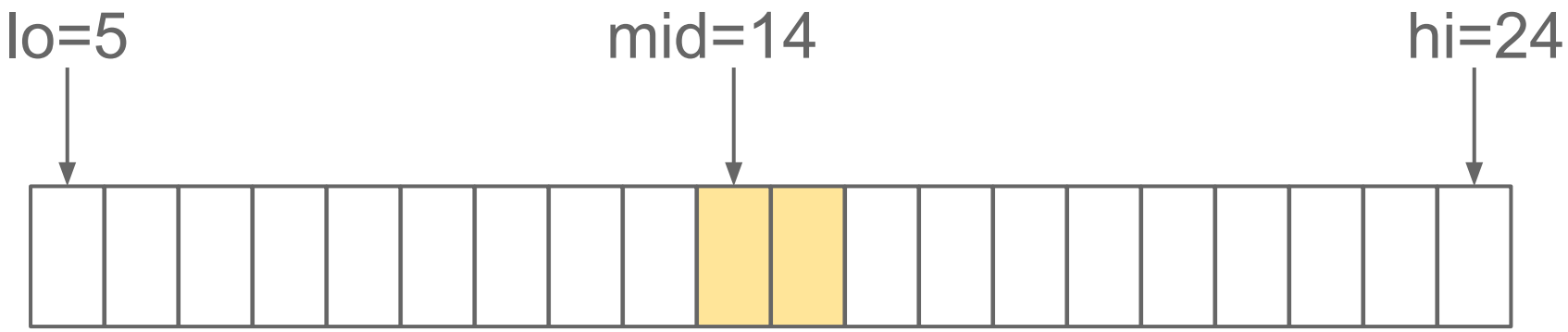
Left intervals: [12, 15], [8, 17]

Starting from subsequence [mid, mid + 1], we expand it to the left, storing all intervals we encounter until we exit the [lo, hi] window.



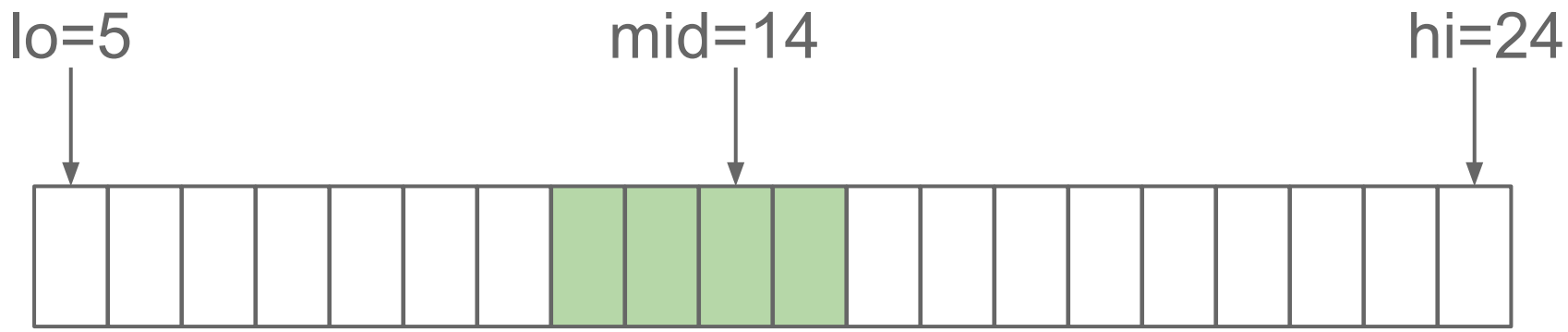
Left intervals: [12, 15], [8, 17], [6, 22]

Starting from subsequence [mid, mid + 1], we expand it to the left, storing all intervals we encounter until we exit the [lo, hi] window.



Left intervals: [12, 15], [8, 17], [6, 22]

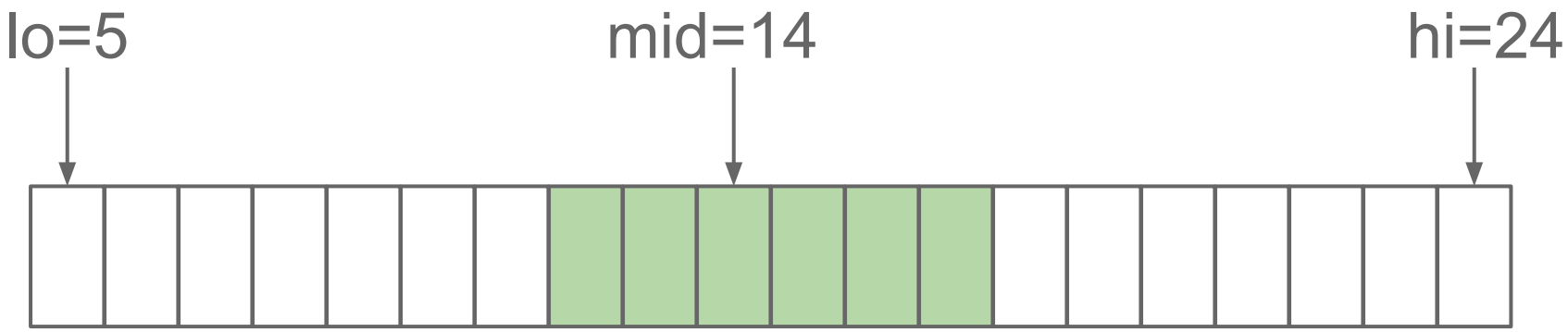
Again, starting from subsequence [mid, mid + 1], we expand it to the right, storing all intervals we encounter until we exit the [lo, hi] window.



Left intervals: [12, 15], [8, 17], [6, 22]

Right intervals: [12, 15]

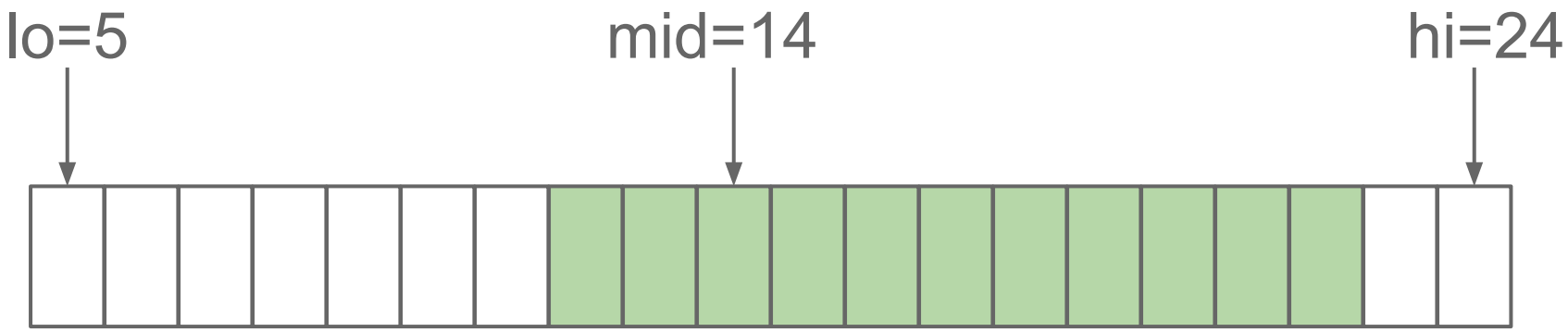
Again, starting from subsequence [mid, mid + 1], we expand it to the right, storing all intervals we encounter until we exit the [lo, hi] window.



Left intervals: [12, 15], [8, 17], [6, 22]

Right intervals: [12, 15], [12, 17]

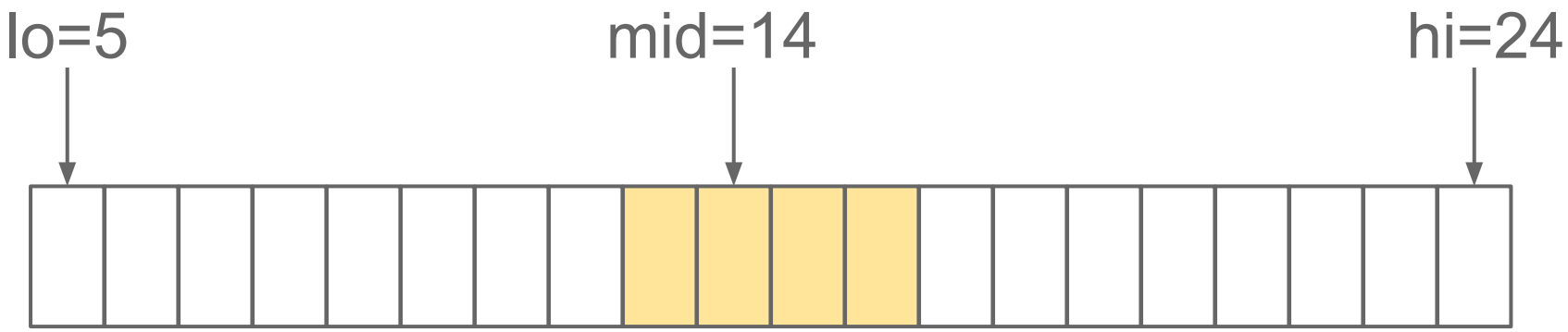
Again, starting from subsequence [mid, mid + 1], we expand it to the right, storing all intervals we encounter until we exit the [lo, hi] window.



Left intervals: [12, 15], [8, 17], [6, 22]

Right intervals: [12, 15], [12, 17], [12, 22]

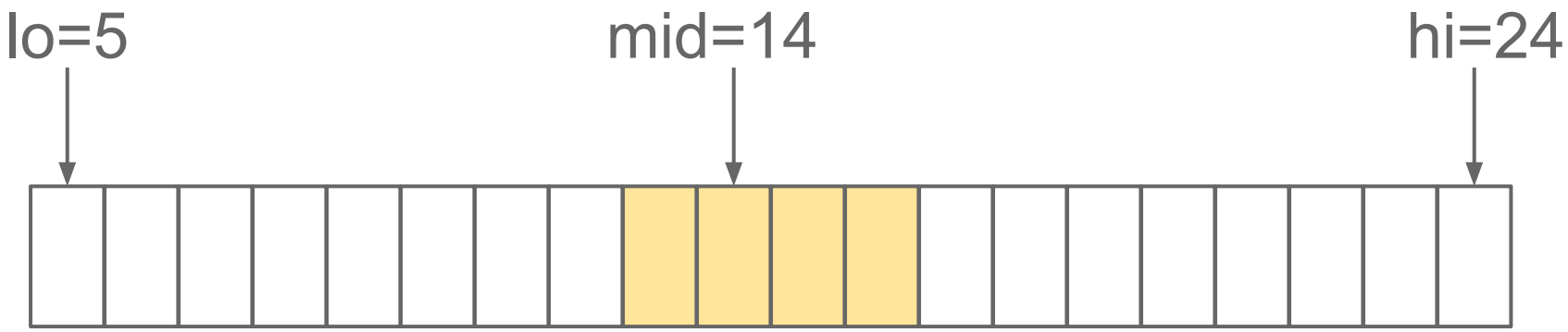
Again, starting from subsequence [mid, mid + 1], we expand it to the right, storing all intervals we encounter until we exit the [lo, hi] window.



Left intervals: [12, 15], **[8, 17]**, [6, 22]

Right intervals: [12, 15], **[12, 17]**, [12, 22]

Finally, for each query [a, b] we find the smallest left interval that contains it and the smallest right interval that contains it. The union of these two intervals is the smallest interval within [lo, hi] that contains the query.



Left intervals: [12, 15], **[8, 17]**, [6, 22]

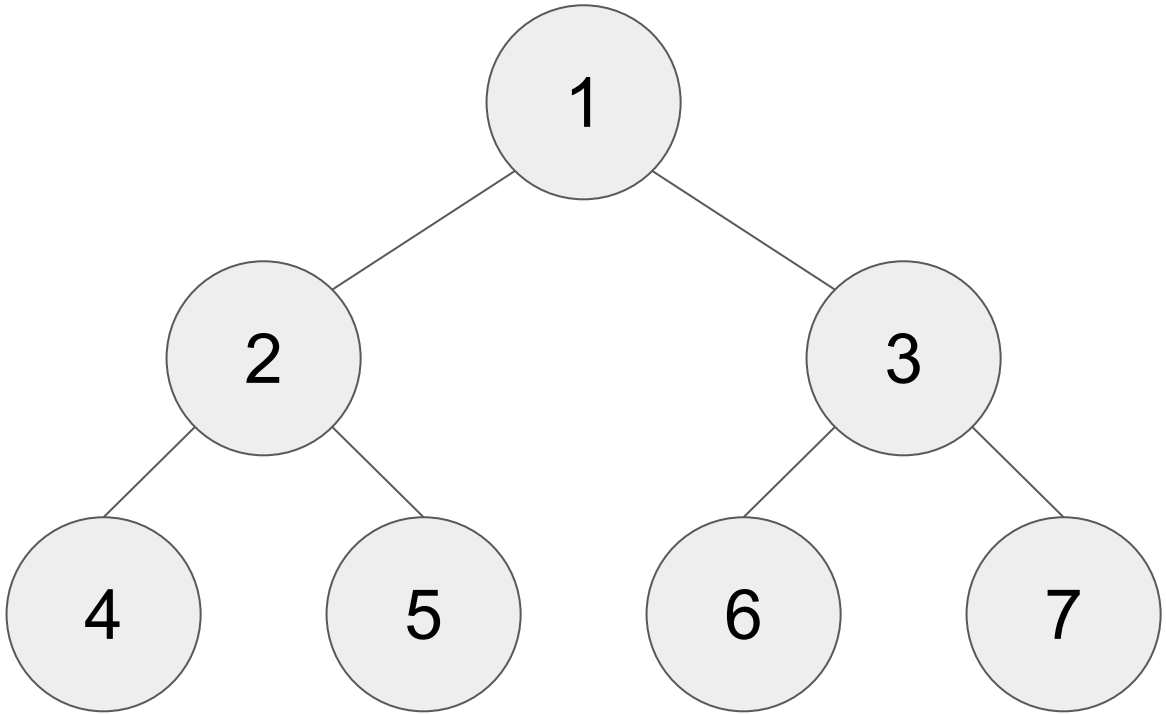
Right intervals: [12, 15], **[12, 17]**, [12, 22]

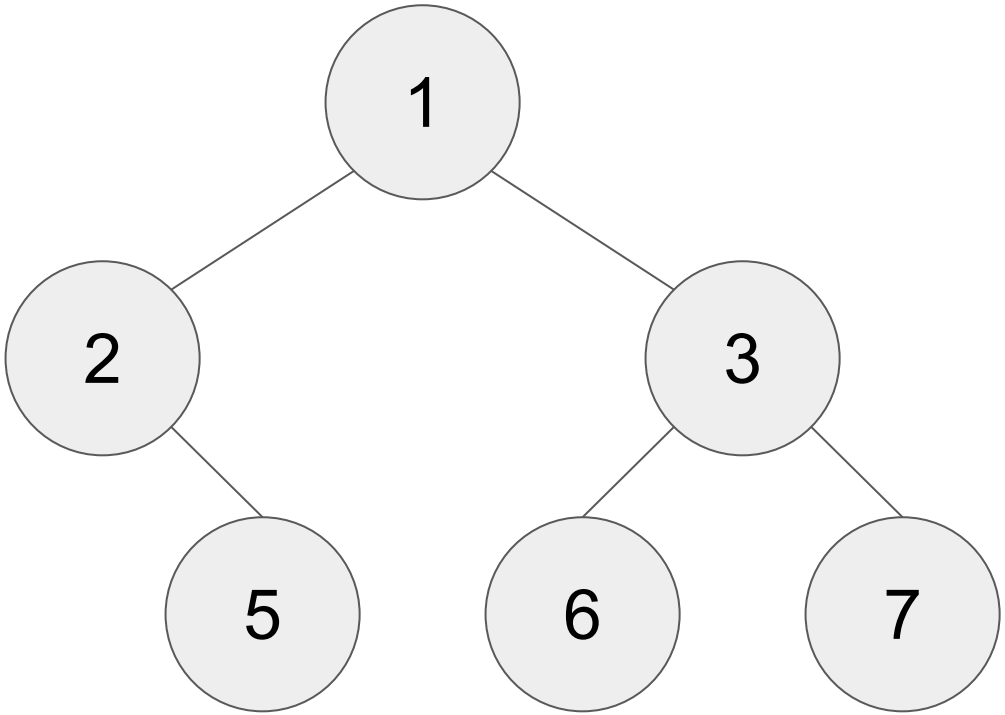
We can implement ImproveViaMid(queries, lo, mid, hi) in O(|hi - lo| + queries.size()), for overall complexity of O((N + Q) log N).

Problem C
Cumulative Code

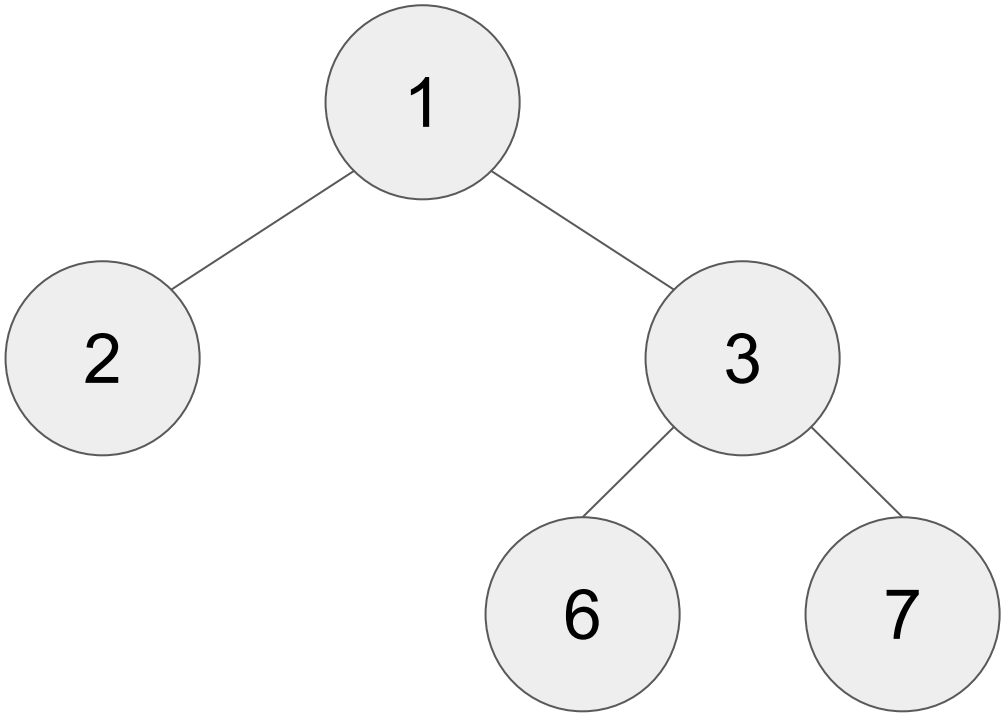
Submits: 2
Accepted: ?

Author: Ivan Paljak, Luka Kalinovčić

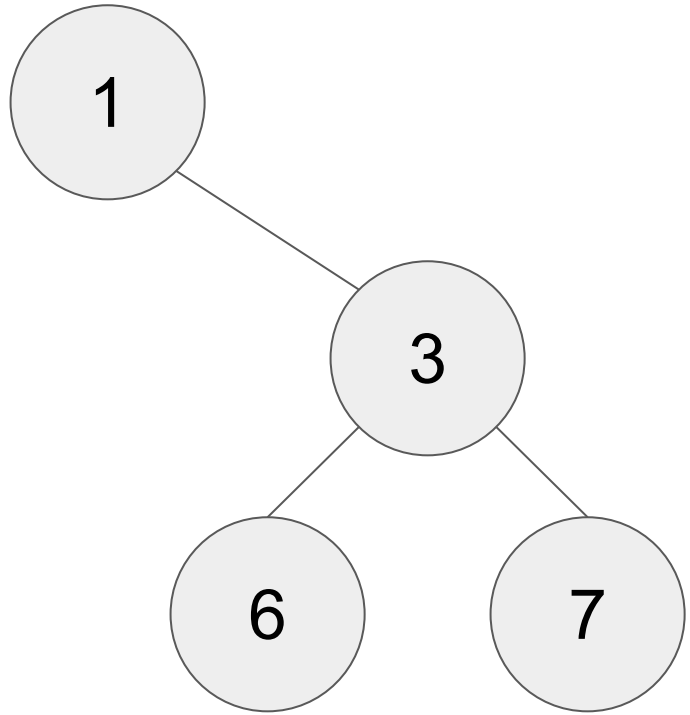

1
2
3
4
5
6
7



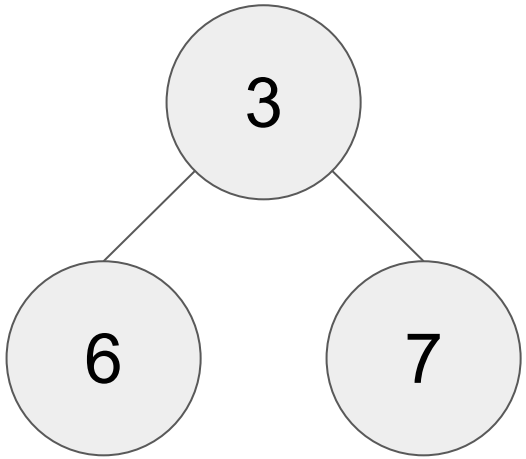
Code: 2



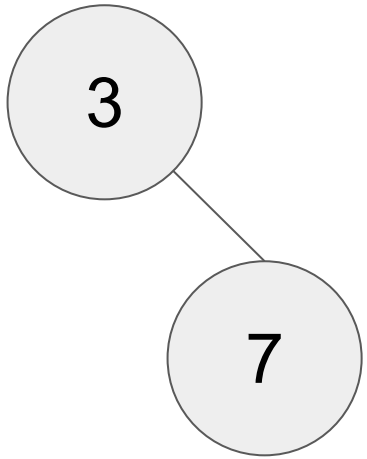
Code: 2 2



Code: 2 2 1

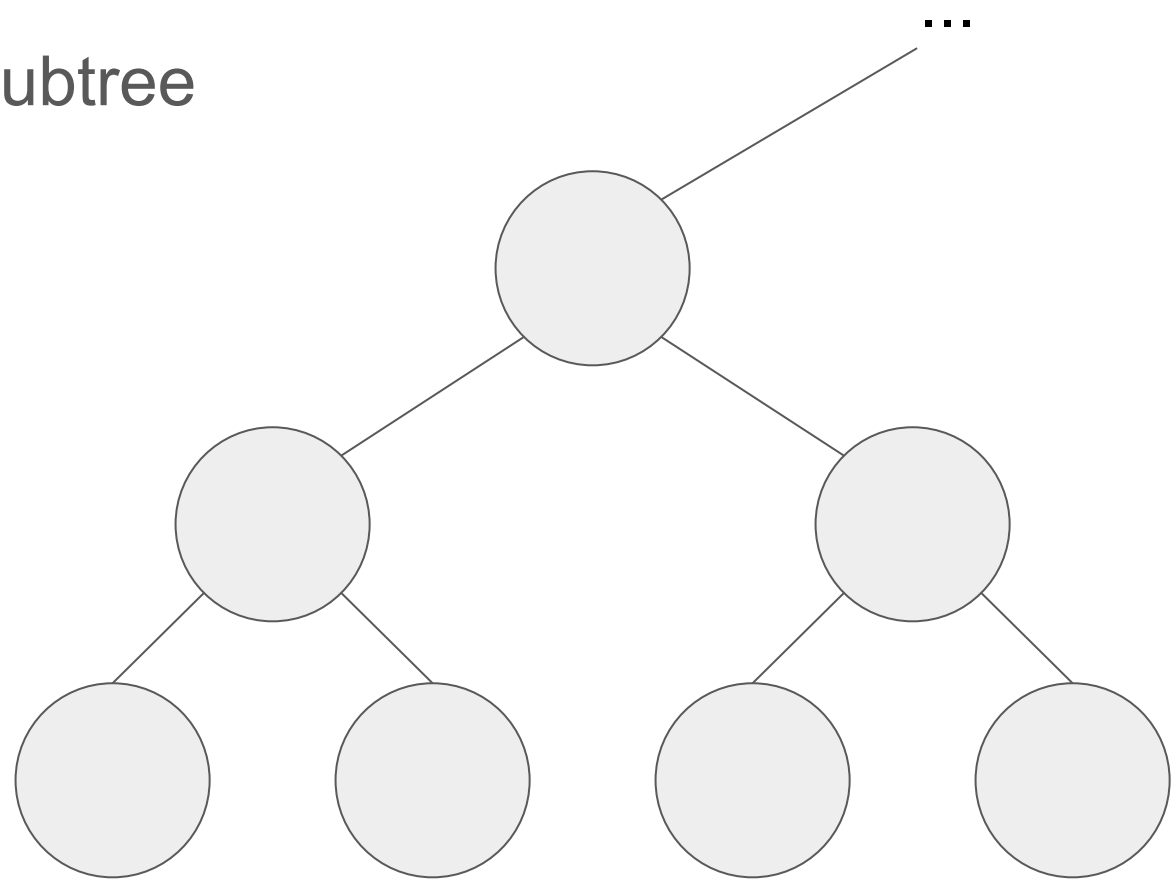


Code: 2 2 1 3



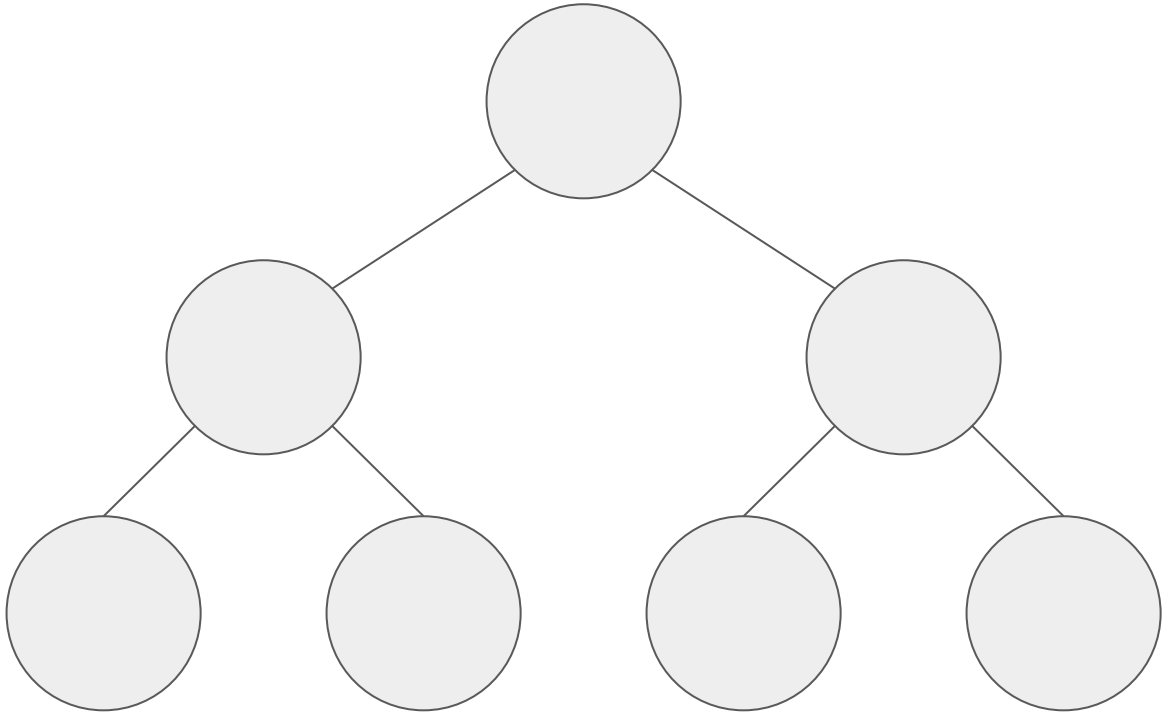
Code: 2 2 1 3 3

Type A subtree



The removal order: left subtree, right subtree, root node.

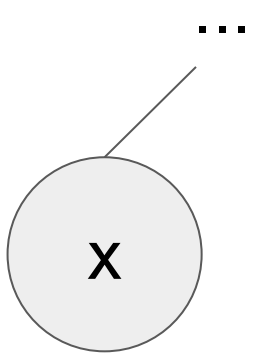
Type B subtree



The removal order: left subtree, root node, right subtree.

In the analysis we'll focus on type A trees only. Type B is dealt with the same way.

Let's start simple and find a recursive formula $f_x(k)$ to sum up the code generated by a type A subtree of depth k, where root is labeled with number x.

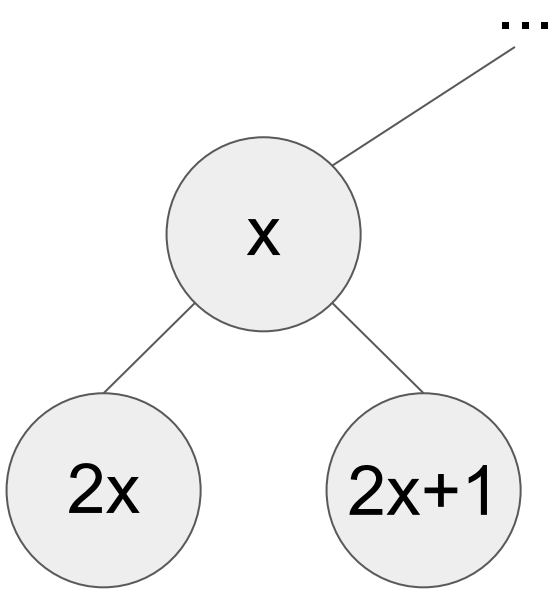


For k = 1, there is only a single node in the subtree.

As we remove it, we append (x div 2) to the code.

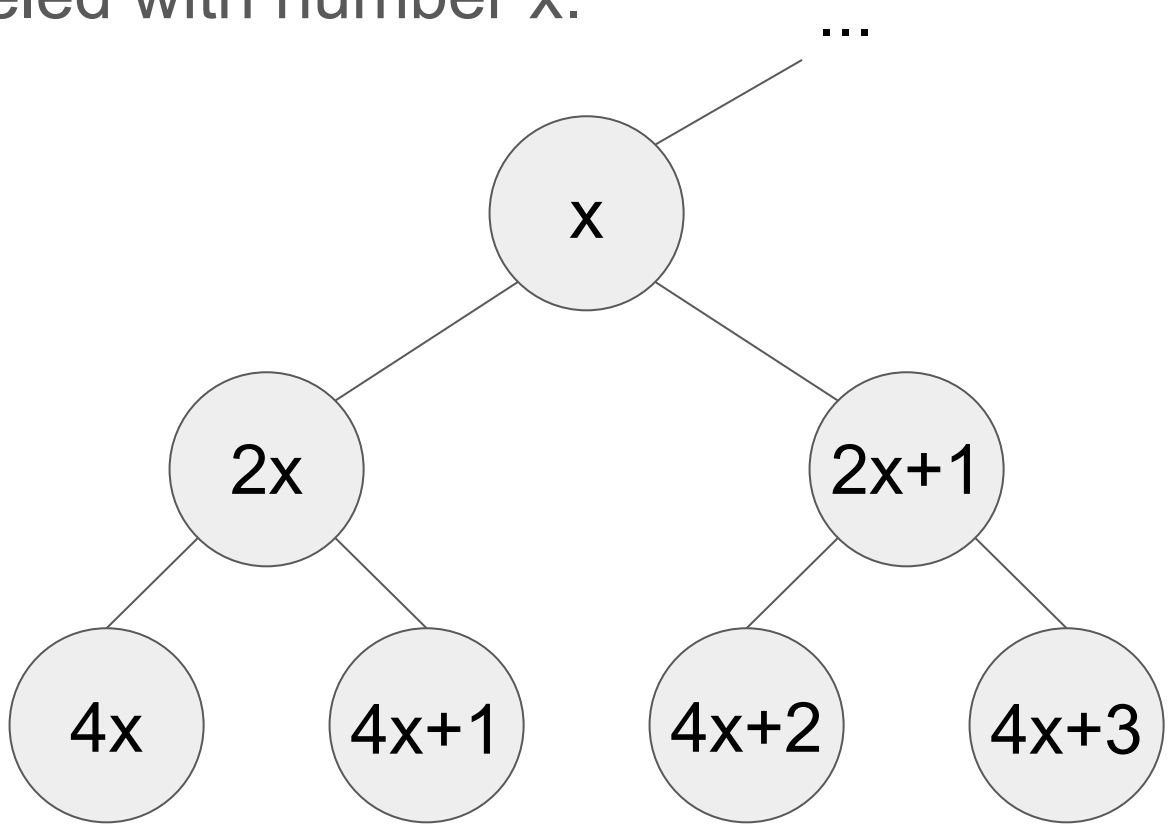
$f_x(1) = (x \text{ div } 2)$

Let's start simple and find a recursive formula $f_x(k)$ to sum up the code generated by a type A subtree of depth k, where root is labeled with number x.



$f_x(2) = x + x + (x \text{ div } 2) = 2x + (x \text{ div } 2)$

Let's start simple and find a recursive formula $f_x(k)$ to sum up the code generated by a type A subtree of depth k, where root is labeled with number x.



$f_x(3) = 2x + 2x + x + 2x+1 + 2x+1 + x + (x \text{ div } 2)$

$= 10x + 2 + (x \text{ div } 2)$

In general, $f_x(k) = a_k \cdot x + b_k + c_k \cdot (x \text{ div } 2)$ and we can compute it recursively:

$$f_x(k) = f_{2x}(k-1) + f_{2x+1}(k-1) + (x \text{ div } 2)$$

$$f_{2x}(k-1) = a_{k-1} \cdot 2x + b_{k-1} + c_{k-1} \cdot (2x \text{ div } 2)$$

$$= (2a_{k-1} + c_{k-1})x + b_{k-1}$$

$$f_{2x+1}(k-1) = a_{k-1} \cdot (2x + 1) + b_{k-1} + c_{k-1} \cdot ((2x + 1) \text{ div } 2)$$

$$= (2a_{k-1} + c_{k-1})x + a_{k-1} + b_{k-1}$$

$$f_x(k) = (4a_{k-1} + 2c_{k-1})x + a_{k-1} + 2b_{k-1} + (x \text{ div } 2)$$

$$a_k = 4a_{k-1} + 2c_{k-1} \qquad b_k = a_{k-1} + 2b_{k-1} \qquad c_k = 1$$

Now, let's come up with a formula that only sums up code elements at indices in the query

$$Q = \{a, a + d, a + 2 \cdot d, ..., a + (m - 1) \cdot d\}.$$

Let $next_Q(i)$ be the smallest index in Q greater than or equal to i.

Let $g_x(k, i)$ be the sum of elements at the required indices, given a subtree of depth k with root labeled x, and given that there are already i elements in the output code before we process the subtree.

$$g_x(k, i) = g_{2x}(k-1, i) + g_{2x+1}(k-1, i + 2^{k-1} - 1)$$
$$+ ((i + 2^k - 1) \in Q) \cdot (x \text{ div } 2)$$

The recursive formula we have is still summing elements one-by-one. We need to optimize it a bit.

1) If no index in $[i + 1, i + 2^k - 1]$ is in query Q, return 0 immediately.

2) Memoize function calls where:

- $k \le K/2$ and
- $[i + 1, i + 2^k - 1]$ is entirely within the query interval $[a, a + a + (m - 1) \cdot d]$.

The key for the memoization is $(k, next_Q(i) - i)$.

Because of 1), $next_Q(i) \le i + 2^k - 1$, so we have $O(2^{K/2})$ states to memoize.

The remaining cases where we don't return 0 or memoize are:

1) Cases for type B subtrees. There are only O(K) such function calls.

2) Cases with k > K/2. There are $O(2^{K/2})$ function calls.

3) Cases where [i + 1, i + 2^k - 1] intersects with the query interval [a, a + a + (m − 1)·d], but is not entirely within. There are only O(K) such function calls.

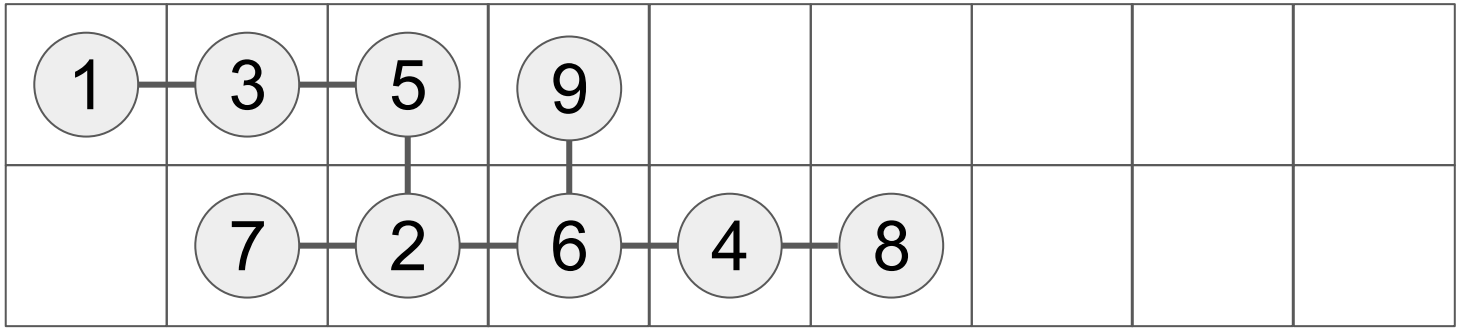
Overall complexity of the algorithm is $O(2^{K/2})$ per query.

Problem E
Embedding Enumeration

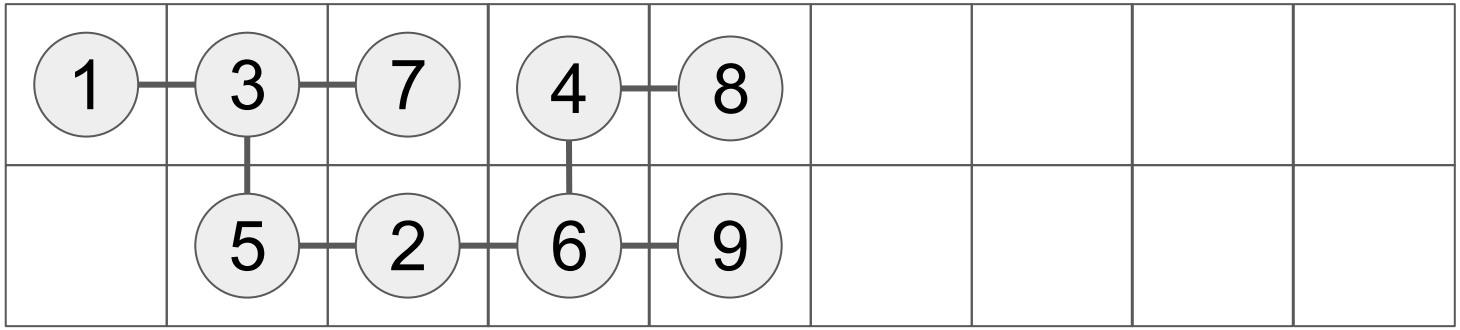
Submits: 1
Accepted: ?

Author: Luka Kalinovčić

Problem: Given a tree, count the number of ways to embed it in a 2 by N grid, such that two nodes connected by an edge are adjacent in the grid. Node 1 has to be in top-left cell.



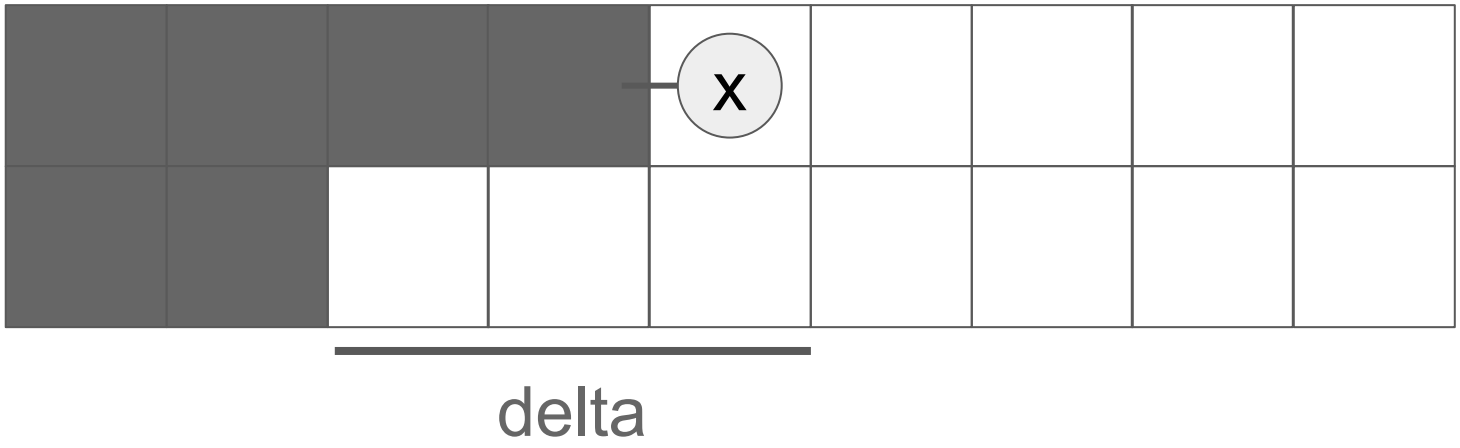
Problem: Given a tree, count the number of ways to embed it in a 2 by N grid, such that two nodes connected by an edge are adjacent in the grid. Node 1 has to be in top-left cell.



Observation: When we root the tree at node 1, it has to be a binary tree. Otherwise, we have a node with degree greater than three, which can't be embedded.

Let's build a dynamic programming solution that enumerates all embeddings. We can describe the state as (x, delta).

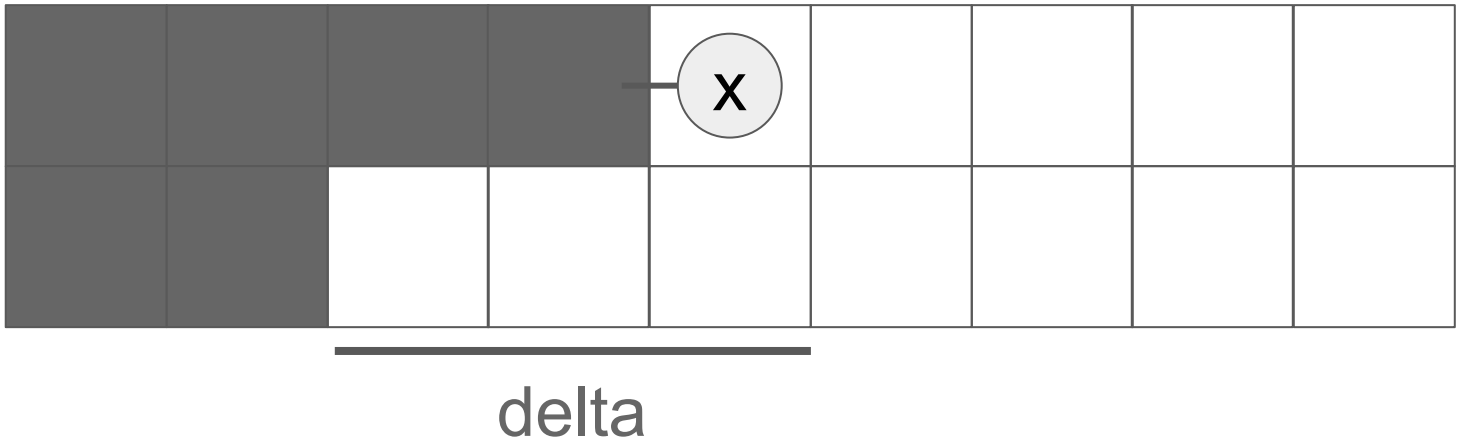
At this state, we have embedded all nodes except for those in x's subtree. Node x is embedded at the last cell of the longer of the two rows, and the delta is the difference in length between the two rows.



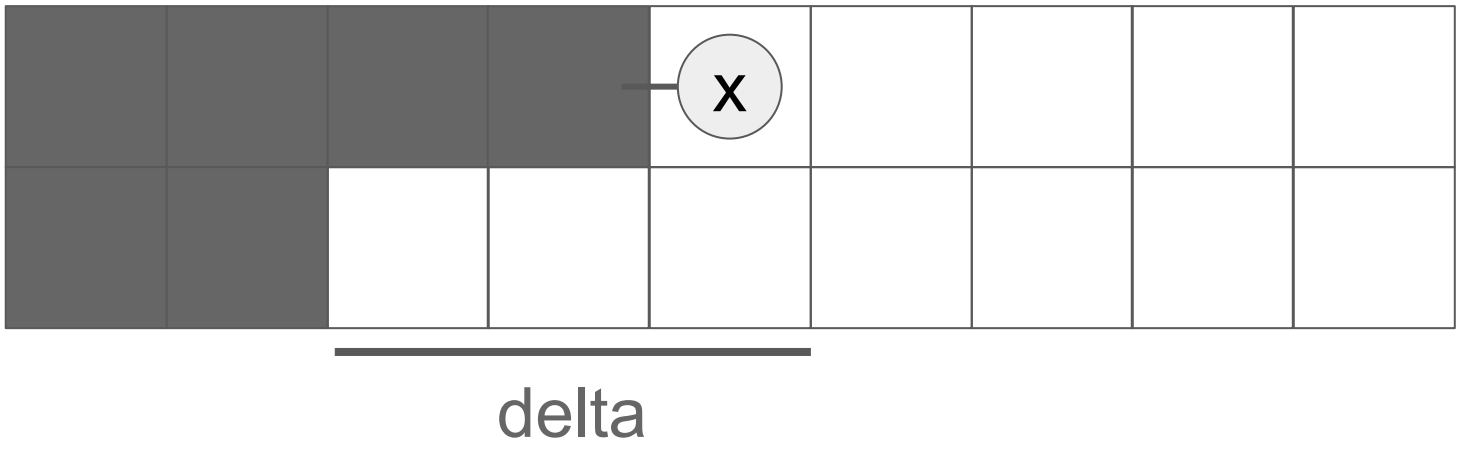
To make the transition, we'll try every possible assignment of node x's children to neighboring cells.

If assignment assigns a node y to a cell below x, we also try every possible assignment of y's children to neighboring cell.

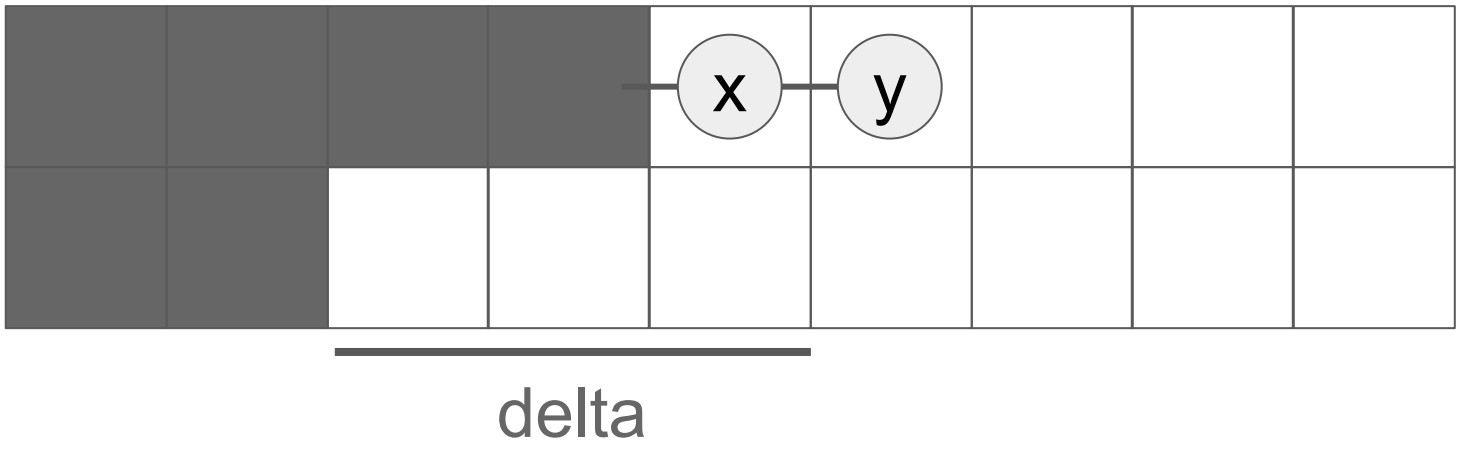
Let's analyze possible outcomes of such assignments.



Trivial case: x has no children. We've found one valid embedding.

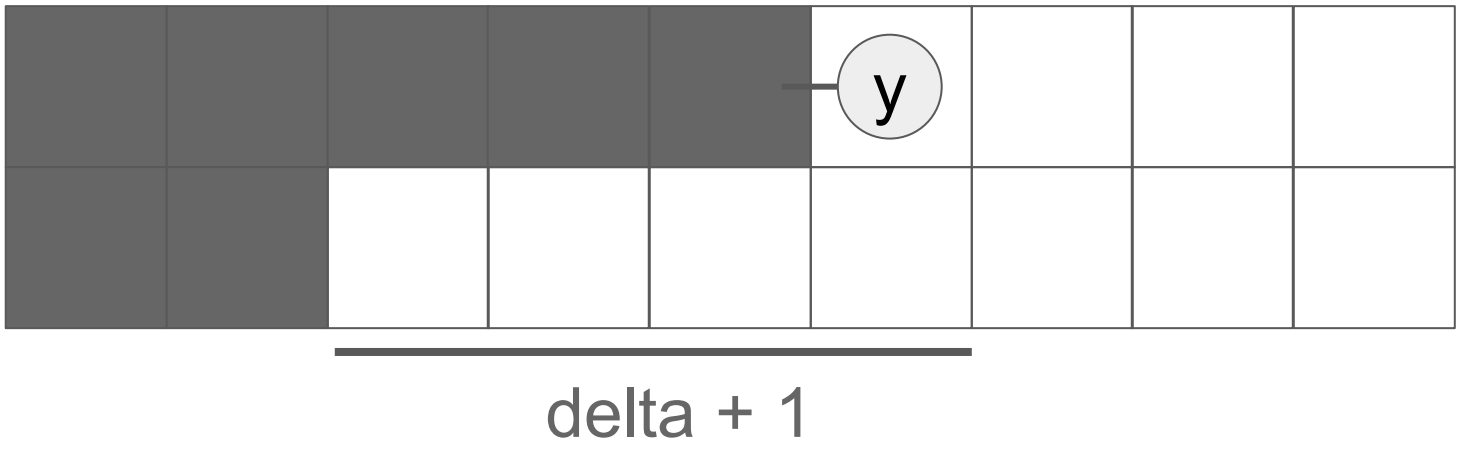


Node x has one child node y that was assigned to the right cell.

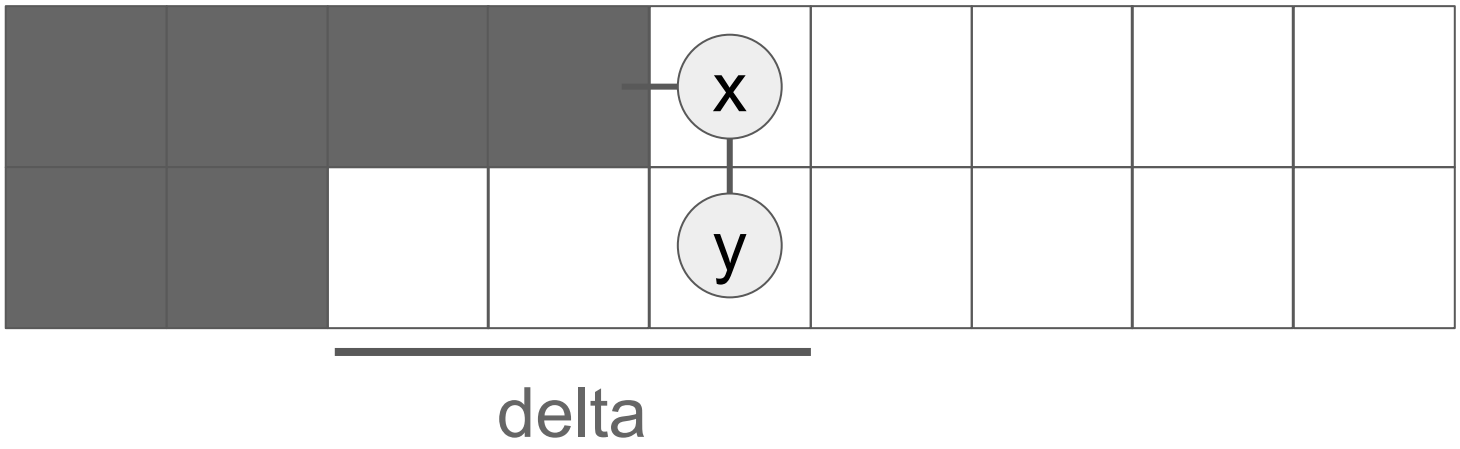


Node x has one child node y that was assigned to the right cell.

We transition to state (y, delta + 1)

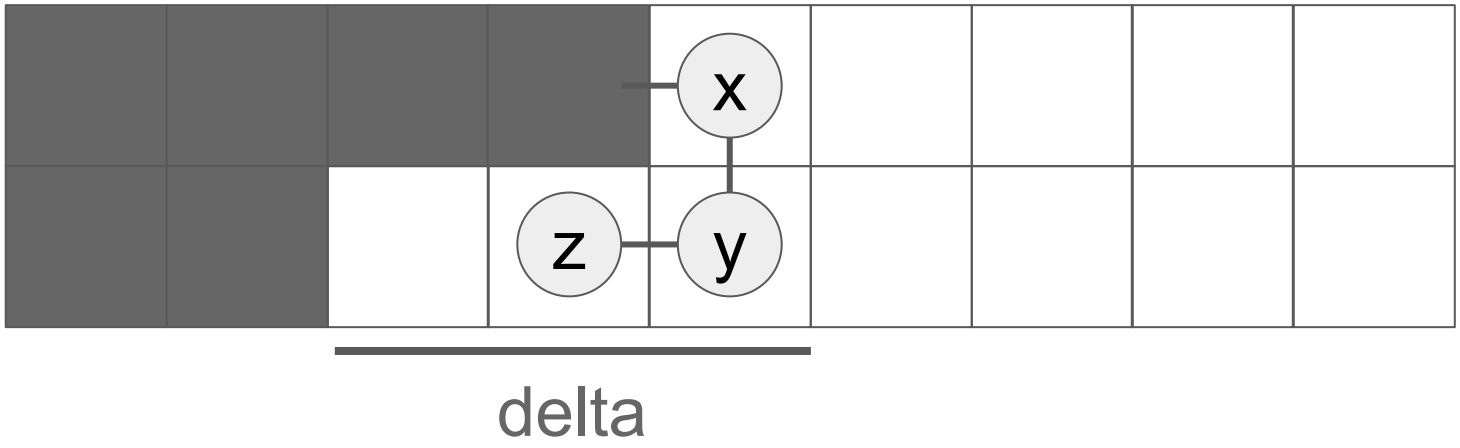


Node x has one child node y that was assigned to the bottom cell. We also assign y's children to neighboring cells.



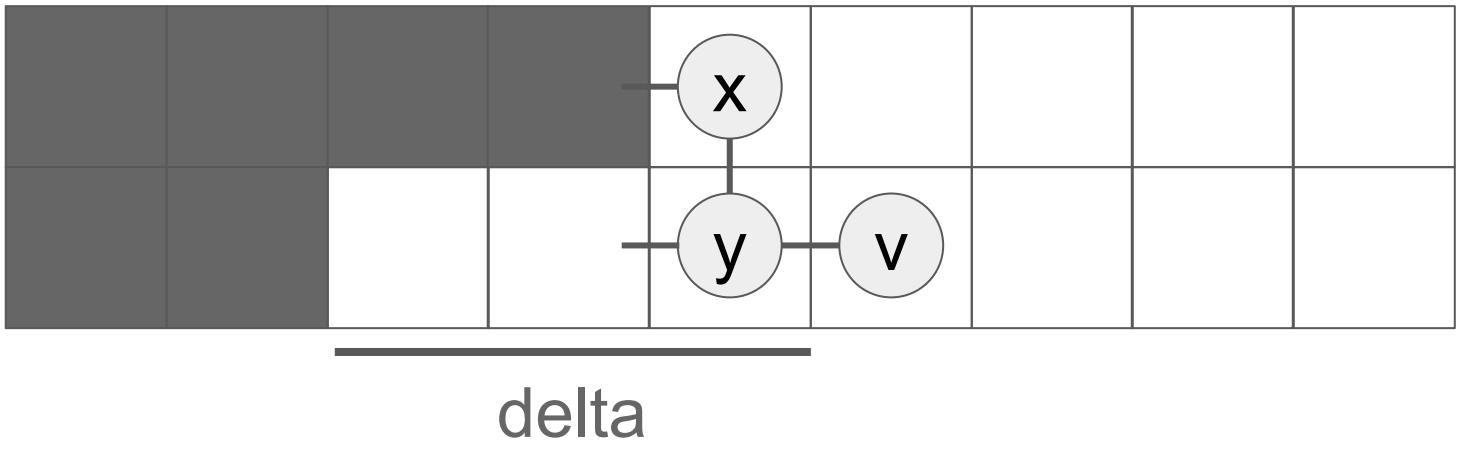
Node x has one child node y that was assigned to the bottom cell. We also assign y's children to neighboring cells.

If it there is a child node z assigned to the left, we know that its subtree has form a simple chain of length up to (delta - 1). Otherwise we can't make a valid embedding from this assignment.



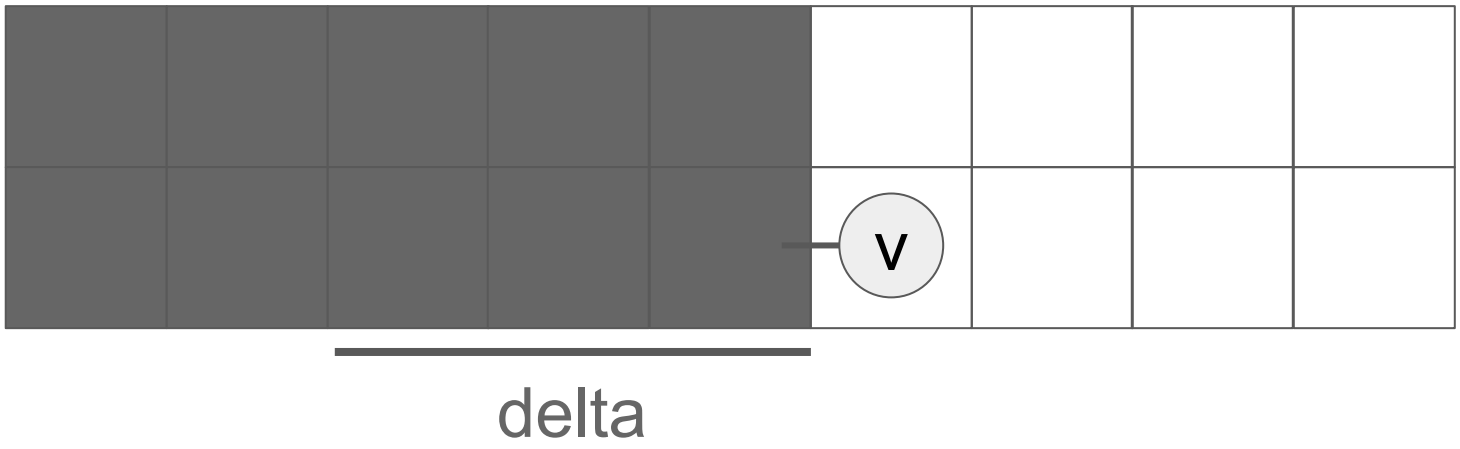
Node x has one child node y that was assigned to the bottom cell. We also assign y's children to neighboring cells.

If it there is a child node v assigned to the right, we transition to state (v, 1).



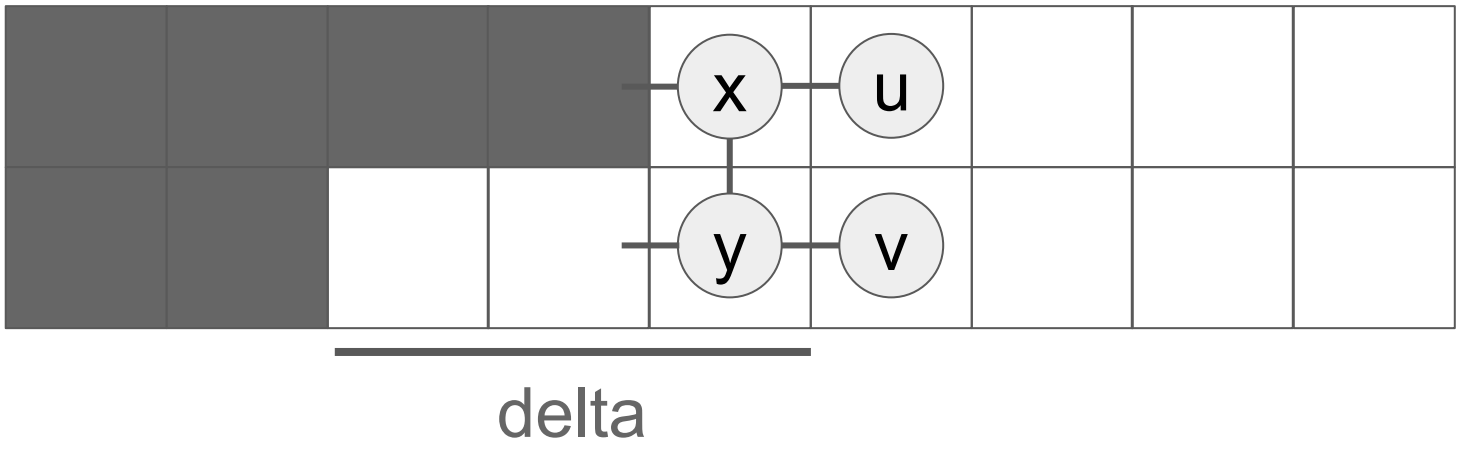
Node x has one child node y that was assigned to the bottom cell. We also assign y's children to neighboring cells.

If it there is a child node v assigned to the right, we transition to state (v, 1).

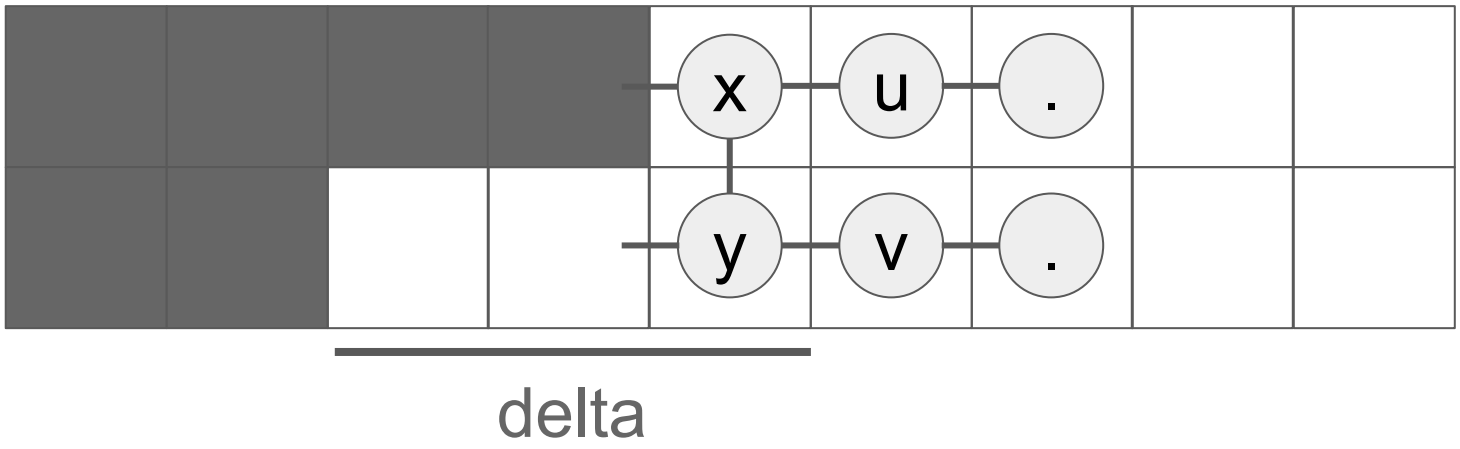


In the general case, x has two children y and u, and y has a child v assigned to the lower right cell.

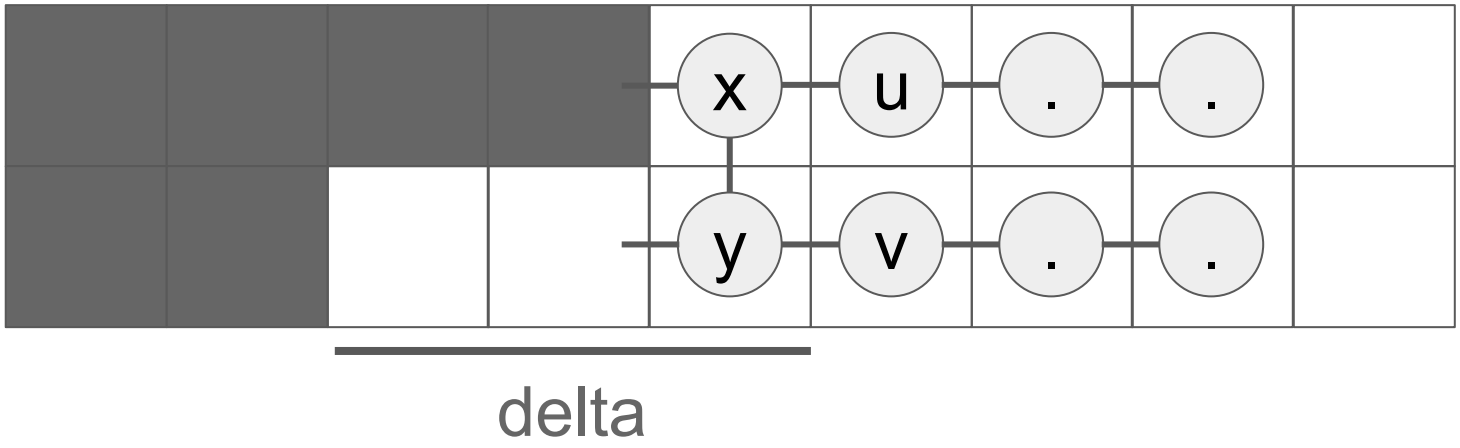
We now have two nodes, u and v, whose subtrees are not yet embedded, so we can't transition to any simple state just yet.



We keep appending children to the right until one of the chain runs out of nodes (or we encounter a node with two children which would make this assignment invalid).

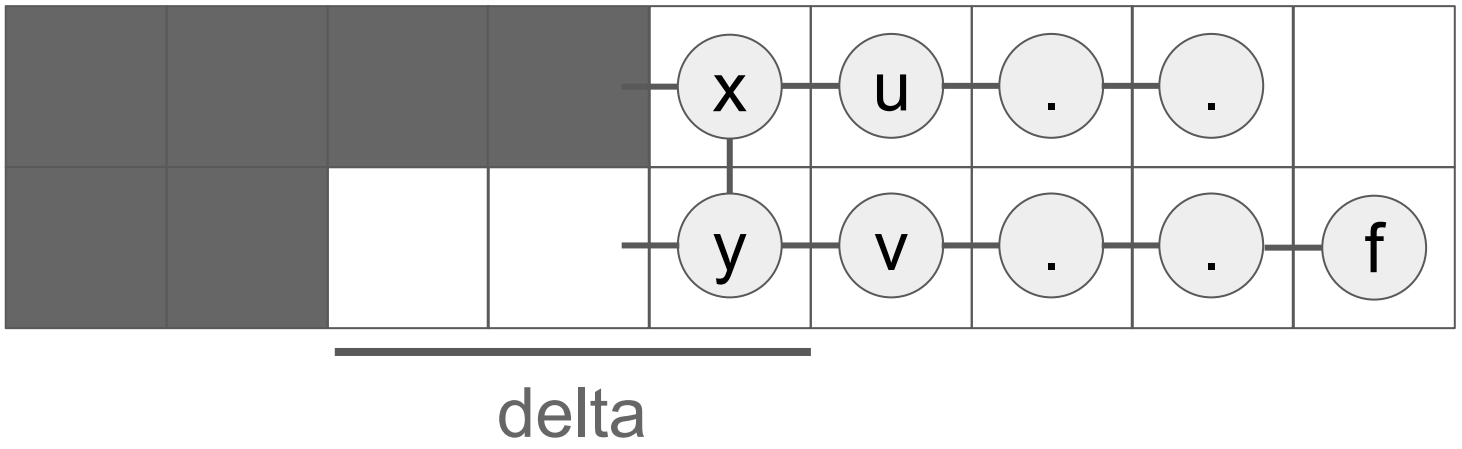


We keep appending children to the right until one of the chain runs out of nodes (or we encounter a node with two children which would make this assignment invalid).



We keep appending children to the right until one of the chain runs out of nodes (or we encounter a node with two children which would make this assignment invalid).

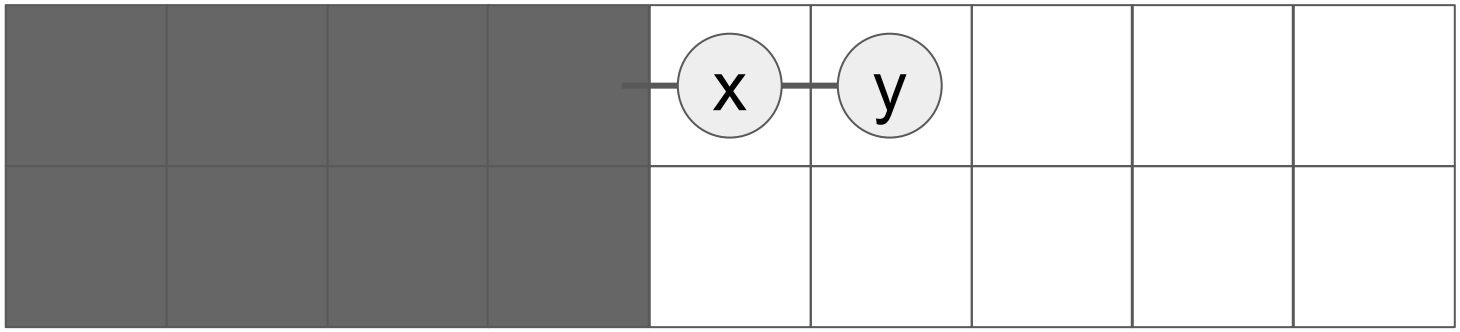
Once that happens, we can transition to state (f, 1).



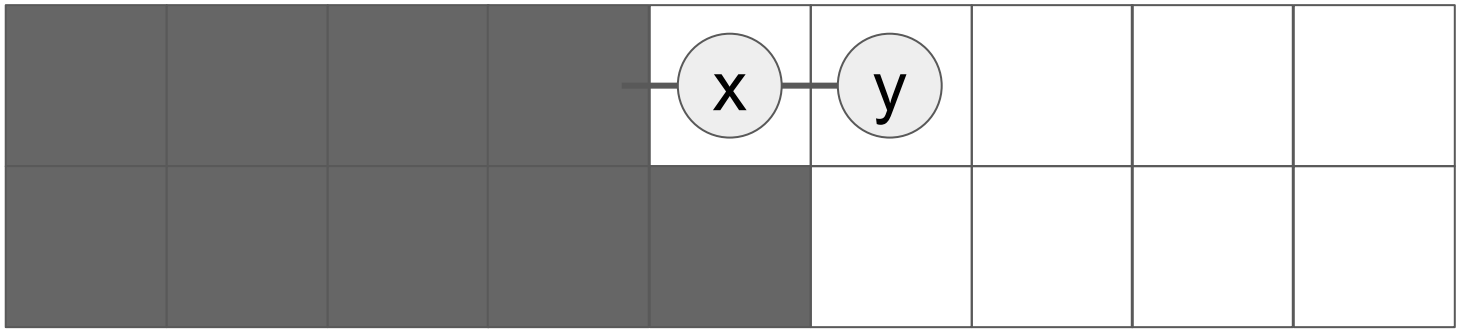
There are $O(N^2)$ states, and it's possible to implement all transitions in O(1) with some precomputation.

To speed it up, let's try to fix delta at 1, and see what breaks.

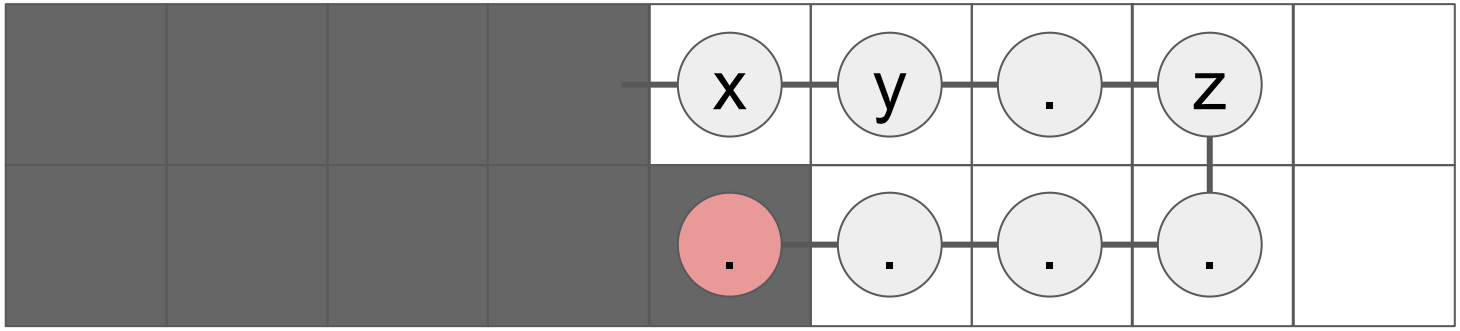
The only case where we actually increase the delta is the one where node x has one child assigned to the right cell.



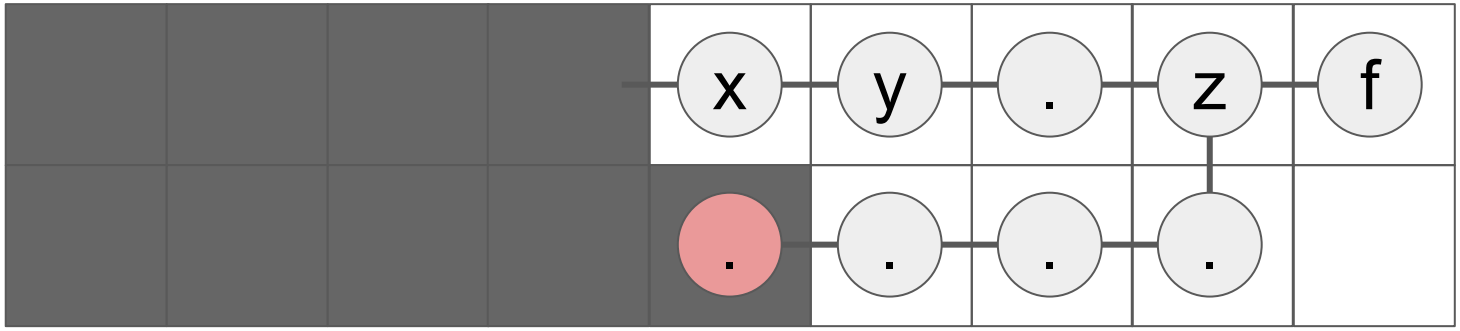
Originally we would transition to state (y, 2), but what kinds of embeddings would we miss if we transitioned to (y, 1) instead?



Originally we would transition to state (y, 2), but what kinds of embeddings would we miss if we transitioned to (y, 1) instead?

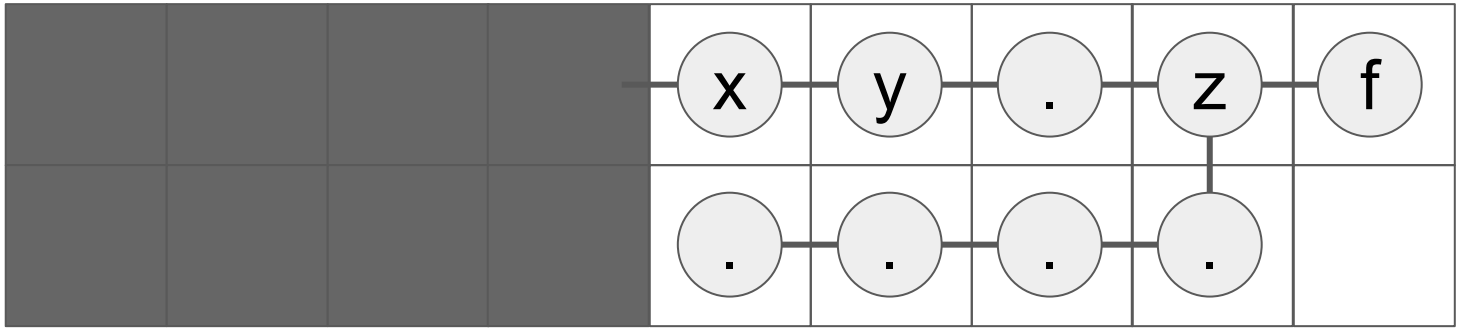


Originally we would transition to state (y, 2), but what kinds of embeddings would we miss if we transitioned to (y, 1) instead?



Originally we would transition to state (y, 2), but what kinds of embeddings would we miss if we transitioned to (y, 1) instead?

We need to identify the node z in the subtree, and assign its neighbour, and verify that there is a chain of the right size going back all the way in the other row.



We've reduced the number of states to O(N) and with some careful programming and precomputation, all the transitions can be done in O(1), so the overall complexity is O(N).

Thanks!