2017

acm

icpc International Collegiate Programming Contest

icpc.foundation

Central Europe Regional Contest

University of Zagreb

Faculty of Electrical Engineering and Computing

November 17–19, 2017

Hosted by:

UNIVERSITAS STUDIORUM ZAGRABIENSIS MDCLXIX

FER

Sponsored by:

JET BRAINS icpc global programming tools sponsor

GDi Creating Efficiencies. Stimulating Growth.

IBM

KSU

Organized by:

CROATIAN COMPUTER SCIENCE ASSOCIATION

Problem A: Assignment Algorithm

Time limit: 1 s
Memory limit: 512 MiB

A low-budget airline is designing a sophisticated algorithm that will assign more desirable seats to passengers who buy tickets earlier. Their airplane has r rows of seats, where r is an even integer. There are also 3 *exit rows* in the airplane; those rows do not contain any seats but only provide access to the emergency exits. One exit row is in the very front of the airplane (before the first row of seats), one in the very back (behind the last row of seats) and one right in the middle. The rows are numbered with integers 1 through $r + 3$ with row numbers increasing from the front to the back of the airplane. Rows numbered 1, $r/2 + 2$ and $r + 3$ are exit rows while all the other rows are seat rows.

The seating configuration is "3–3–3" — each seat row contains three groups of three seats with the passenger aisles between the groups. Seats in the same row are denoted with consecutive letters left to right corresponding to the pattern "`ABC.DEF.GHI`".

When a passenger purchases a ticket, she is assigned a seat according to the following rules:

1. If there is an empty seat in a row directly after an exit row, all other rows are ignored in the following step (but they are not ignored when balancing the airplane in the last step).
2. First, we select a seat row with the largest number of empty seats. If there are multiple such rows, we select the one closest to an exit row (distance between rows a and b is simply $|a - b|$). If there are still multiple such rows, we select the one with the lowest number.
3. Now, we consider empty seats in the selected row and select one with the highest *priority*. Seat priorities, from highest to lowest are as follows:
   (a) Aisle seats in the middle group (`D` or `F`).
   (b) Aisle seats in the first and third group (`C` or `G`).
   (c) Window seats (`A` or `I`).
   (d) Middle seat in the middle group (`E`).
   (e) Other middle seats (`B` or `H`).

   If there are two empty seats with the same highest priority, we consider the balance of the entire airplane. The airplane's *left side* contains all seats with letters `A`, `B`, `C` or `D`, while the *right side* contains all seats with letters `F`, `G`, `H` or `I`. We select an empty seat in the side with more empty seats. If both sides have the same number of empty seats, we select the seat in the left side of the airplane.

Some of the airplane's seats are already reserved (possibly using a completely different procedure from the one described above). Determine the seats assigned to the next n passengers purchasing a ticket.

Input

The first line contains two integers r and n ($2 \le r \le 50$, $1 \le n \le 26$) — the number of seat rows in the airplane (always an even integer) and the number of new passengers purchasing tickets. The following $r + 3$ lines contain the current layout of the airplane. The j-th line contains exactly 11 characters — the layout of the row j of the airplane. Exit rows and aisles are denoted with the "`.`" characters. The "`#`" character denotes a reserved seat, while the "`-`" character denotes a seat that is currently empty. You may assume there will be at least n empty seats in the airplane.

Output

Output $r + 3$ lines containing the final layout of the airplane. The layout should be exactly the same as in input with the following exception: the seat assigned to the j-th passenger purchasing a ticket should be denoted with the *j*-th lowercase letter of the English alphabet.

Example

input

```
2 17
...........
---.#--.---
...........
---.---.---
...........
```

output

```
...........
hnd.#lb.fpj
...........
kqg.cma.eoi
...........
```

input

```
6 26
...........
---.---.###
#-#.---.---
---.###.---
...........
---.###.---
#--.#-#.--#
#--.--#.#-#
...........
```

output

```
...........
gke.aic.###
#-#.mzo.r-v
x-p.###.n-t
...........
fjb.###.dlh
#-s.#-#.w-#
#-u.qy#.#-#
...........
```

Problem B: Buffalo Barricades

Time limit: 5 s
Memory limit: 512 MiB

A pasture in the wild west can be represented as a rectangular grid embedded in the upper-right quadrant of a standard coordinate system. A herd of n buffalos is scattered throughout the grid, each occupying a unit square. Buffalos are numbered 1 through n; buffalo j is located in the unit square whose upper-right corner is the point with integer coordinates (x_j, y_j). The coordinate axes represent two rivers meeting at the origin, restricting buffalo movement downwards and leftwards.

A total of m settlers arrive, one by one, and each claims a piece of land using the following procedure:

1. The settler picks a point with integer coordinates and installs a single fence post at that point. The point he picks is guaranteed to be free of any previously installed fence posts or fences. Moreover, no two fence posts will have the same x-coordinate and no two fence posts will have the same y-coordinates.

2. Starting from the fence post, the settler builds horizontal and vertical fence segments leftwards and downwards, respectively. Each segment is built to be as long as possible — i. e. until it reaches the river or another fence.

3. The settler claims all the land in the connected area bounded with fences and rivers whose upper-right corner is his fence post. Of course, he claims all the buffalos inside as well. Note that settlers arriving later may claim pieces of land already claimed by earlier settlers.

For each settler, find the total number of buffalos he claimed when he arrived.

Input

The first line contains an integer n ($1 \le n \le 300\,000$) — the number of buffalos. The j-th of the following n lines contains two integers x_j and y_j ($1 \le x_j, y_j \le 10^9$) — the location of the j-th buffalo. No two buffalos will share the same location.

The following line contains an integer m ($1 \le m \le 300\,000$) — the number of settlers. The j-th of the following m lines contains two integers x'_j and y'_j ($1 \le x'_j, y'_j \le 10^9$) — the coordinates of the fence post installed by the j-th settler. All x'_j are different and all y'_j are different.

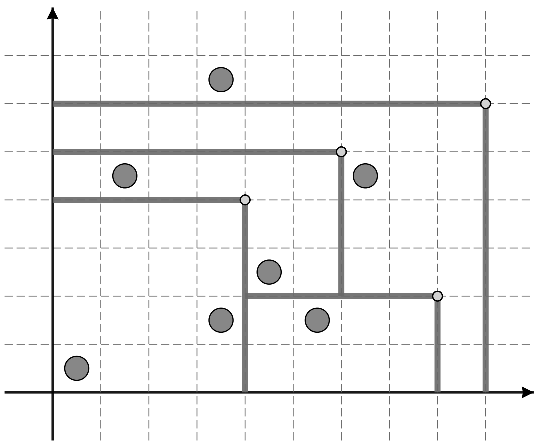
Output

Output m lines. The j-th line should contain the number of buffalos claimed by the j-th settler upon arrival.

Example

input

```
7
1 1
4 2
6 2
5 3
2 5
4 7
7 5
4
4 4
8 2
9 6
6 5
```



output

```
2
1
3
2
```

Problem C: Cumulative Code

Time limit: 7 s
Memory limit: 512 MiB

As you probably know, a *tree* is a graph consisting of n nodes and $n-1$ undirected edges in which any two nodes are connected by exactly one path. In a *labeled tree* each node is labeled with a different integer between 1 and n.

The *Prüfer code* of a labeled tree is a unique sequence associated with the tree, generated by repeatedly removing nodes from the tree until only two nodes remain. More precisely, in each step we remove the *leaf* with the smallest label and append the label of its *neighbour* to the end of the code. Recall, a leaf is a node with exactly one neighbour. Therefore, the Prüfer code of a labeled tree is an integer sequence of length $n-2$. It can be shown that the original tree can be easily reconstructed from its Prüfer code.

The *complete binary tree of depth* k, denoted with C_k, is a labeled tree with $2^k - 1$ nodes where node j is connected to nodes $2j$ and $2j+1$ for all $j < 2^{k-1}$. Denote the Prüfer code of C_k with $p_1, p_2, \ldots, p_{2^k-3}$.

Since the Prüfer code of C_k can be quite long, you do not have to print it out. Instead, you need to answer n questions about the sums of certain elements on the code. Each question consists of three integers: a, d and m. The answer is the sum of the of the C_k's Prüfer code elements p_a, p_{a+d}, p_{a+2d}, $\ldots$, $p_{a+(m-1)d}$.

Input

The first line contains two integers k and q ($2 \le k \le 30$, $1 \le q \le 300$) — the depth of the complete binary tree and the number of questions. The j-th of the following q lines contains the j-th question: three positive integers a_j, d_j and m_j such that a_j, d_j and $a_j + (m_j - 1)d_j$ are all at most $2^k - 3$.

Output

Output 1 lines. The j-th line should contain a single integer — the answer to the j-th question.

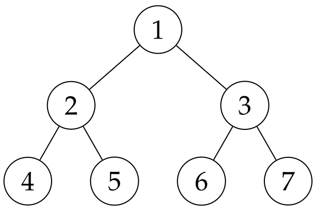
Example

input

```
3 5
1 1 1
2 1 1
3 1 1
4 1 1
5 1 1
```

output

```
2
2
1
3
3
```



input

```
4 4
2 1 5
4 4 3
4 8 1
10 3 2
```

output

```
18
15
5
13
```

input

```
7 1
1 1 125
```

output

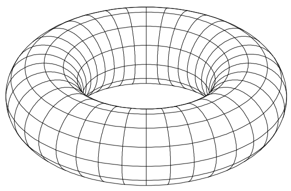
```
4031
```

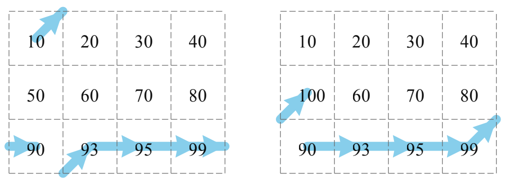
In the first example above, when constructing the Prüfer code for C_3 the nodes are removed in the following order: 4, 5, 2, 1, 6. Therefore, the Prüfer code of C_3 is 2, 2, 1, 3, 3.

Problem D: Donut Drone

Time limit: 8
Memory limit: 512 MiB

You are building a simulation in which a drone explores a volatile torus-shaped planet. Technically, the drone is moving across a *toroidal grid* — a rectangular grid that wraps around circularly in both dimensions. The grid consists of cells organized into r rows numbered 1 through r top to bottom and c columns numbered 1 through c left to right. Each grid cell has a certain *elevation* — a positive integer.


A toroidal grid.



The paths in two `move` events in the second example input.

The drone is initially located in the cell in the first row and first column. In each *step* the drone considers three cells: the cell directly to the right, the cell diagonally right-down and the cell diagonally right-up (wrapping around if necessary). The drone flies to the cell with the largest elevation of the three.

Two types of events may happen during the simulation:

- "`move` k" — The drone makes k steps.
- "`change` a b e" — The elevation of the cell in row a column b changes to e.

Find the drone's position immediately after each `move` event. You may assume that at each point in time, no sequence of three circularly consecutive cells in the same column will have the same elevation. Hence, each drone step is well defined.

Input

The first line contains two integers r and c ($3 \le r, c \le 2\,000$) — the number of rows and the number of columns of the toroidal grid. The i-th of the following r lines contains a sequence of c integers $e_{i,1}, e_{i,2}, \ldots, e_{i,c}$ ($1 \le e_{i,j} \le 10^9$) — the initial elevations of cells in row i.

The following line contains an integer m ($1 \le m \le 5\,000$) — the number of events. The j-th of the following m lines contains the j-th event and is either of the form "`move` k" where k is an integer such that $1 \le k \le 10^9$ or "`change` a b e" where a, b and e are integers such that $1 \le a \le r$, $1 \le b \le c$ and $1 \le e \le 10^9$.

Output

Output w lines where w is the number of `move` events in the input — the j-th line should contain the drone's position (row and column numbers) after the j-th `move` event in the input.

Example

input

```
4 4
1 2 9 3
3 5 4 8
4 3 2 7
5 8 1 6
4
move 1
move 1
change 1 4 100
move 1
```

output

```
4 2
1 3
1 4
```

input

```
3 4
10 20 30 40
50 60 70 80
90 93 95 99
3
move 4
change 2 1 100
move 4
```

output

```
3 1
2 1
```

Problem E: Embedding Enumeration

Time limit: 4 s
Memory limit: 512 MiB

As you probably know, a *tree* is a graph consisting of n nodes and $n-1$ undirected edges in which any two nodes are connected by exactly one path. In a *labeled tree* each node is labeled with a different integer between 1 and n. In general, it may be hard to visualize trees nicely, but some trees can be neatly embedded in rectangular grids.

Given a labeled tree G with n nodes, a *2 by n embedding* of G is a mapping of nodes of G to the cells of a rectangular grid consisting of 2 rows and n columns such that:

- Node 1 is mapped to the cell in the upper-left corner.
- Nodes connected with an edge are mapped to neighboring grid cells (up, down, left or right).
- No two nodes are mapped to the same cell.

Find the number of 2 by n embeddings of a given tree, modulo $10^9 + 7$.

Input

The first line contains an integer n ($1 \le n \le 300\,000$) — the number of nodes in G. The j-th of the following $n-1$ lines contains two different integers a_j and b_j ($1 \le a_j, b_j \le n$) — the endpoints of the j-th edge.

Output

Output the number of 2 by n embeddings of the given tree, modulo $10^9 + 7$.

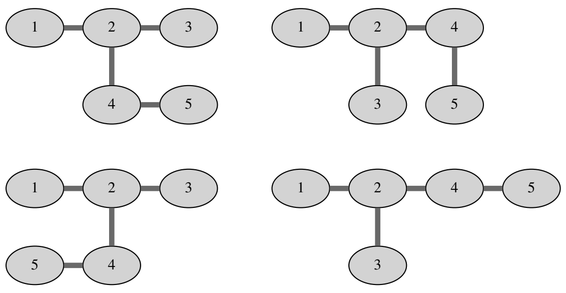
Example

input

```
5
1 2
2 3
2 4
4 5
```

output

```
4
```



All 4 embeddings of the tree in the example input are given in the figure above.

Problem F: Faulty Factorial

Time limit: 3 s
Memory limit: 512 MiB

The *factorial* of a natural number is the product of all positive integers less than or equal to it. For example, the factorial of 4 is $1 \cdot 2 \cdot 3 \cdot 4 = 24$. A *faulty factorial* of length n is similar to the factorial of n, but it contains a fault: one of the integers is *strictly smaller* than what it should be (but still at least 1). For example, $1 \cdot 2 \cdot 2 \cdot 4 = 16$ is a faulty factorial of length 4.

Given the length n, a *prime* modulus p and a target remainder r, find some faulty factorial of length n that gives the remainder r when divided by p.

Input

The first line contains three integers n, p and r ($2 \leq n \leq 10^{18}, 2 \leq p < 10^7, 0 \leq r < p$) — the length of the faulty factorial, the prime modulus and the target remainder as described in the problem statement.

Output

If there is no faulty factorial satisfying the requirements output "-1 -1". Otherwise, output two integers — the index k of the fault ($2 \leq k \leq n$) and the value v at that index ($1 \leq v < k$). If there are multiple solutions, output any of them.

Examples

input

```
4 5 1
```

output

```
3 2
```

input

```
4 127 24
```

output

```
-1 -1
```

In the first example, the output describes the faulty factorial $1 \cdot 2 \cdot 2 \cdot 4 = 2^4 \equiv 1 (\bmod 5)$.

Problem G: Gambling Guide

Time limit: 3 s
Memory limit: 512 MiB

A railroad network in a nearby country consists of n cities numbered 1 through n, and m two-way railroad tracks each connecting two different cities. Tickets can only be purchased at automated machines installed at every city. Unfortunately, hackers have tampered with the ticket machines and now they all work as follows: when a single coin is inserted in the machine installed at city a, the machine dispenses a single one-way ticket from a to a *random* neighboring city. More precisely, the destination city is chosen uniformly at random among all cities directly connected to a with a railroad track. Destinations on different tickets originating in the same city are independent.

A computer science student needs to travel from city 1 (where she lives) to city n (where a regional programming contest has already started). She knows how the machines work (but of course cannot predict the random choices) and has a map of the railway network. In each city, when she purchases a ticket, she can either immediately use it and travel to the destination city on the ticket, or discard the ticket and purchase a new one. She can keep purchasing tickets indefinitely. The trip is finished as soon as she reaches city n.

After doing some calculations, she has devised a traveling strategy with the following properties:

- The probability that the trip will eventually finish is 1.
- The expected number of coins spent on the trip is the smallest possible.

Find the expected number of coins she will spend on the trip.

Input

The first line contains two integers n and m ($1 \le n, m \le 300\,000$) — the number of cities and the number of railroad tracks. Each of the following m lines contains two different integers a and b ($1 \le a, b \le n$) which describe a railroad track connecting cities a and b. There will be at most one railroad track between each pair of cities. It will be possible to reach city n starting from city 1.

Output

Output a single number — the expected number of coins spent. The solution will be accepted if the absolute or the relative difference from the judges solution is less than 10^{-6}.

Example

input

```
4 4
1 2
1 3
2 4
3 4
```

output

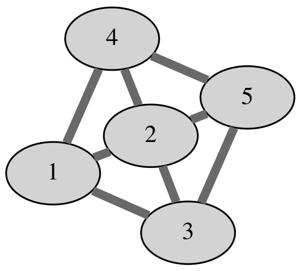
```
3.0000000000
```

input

```
5 8
1 2
1 3
1 4
2 3
2 4
3 5
5 4
2 5
```

output

```
4.1111111111
```



Problem H: Hidden Hierarchy

Time limit: 1 s
Memory limit: 512 MiB

You are working on the user interface for a simple text-based file explorer. One of your tasks is to build a navigation pane displaying the *directory hierarchy*. As usual, the filesystem consists of directories which may contain files and other directories, which may, in turn, again contain files and other directories etc. Hence, the directories form a hierarchical tree structure. The top-most directory in the hierarchy is called the *root* directory. If directory d directly contains directory e we will say that d is the *parent directory* of e while e is a *subdirectory* od d. Each file has a *size* expressed in bytes. The directory size is simply the total size of all files directly or indirectly contained inside that directory.

All files and all directories except the root directory have a *name* — a string that always starts with a letter and consists of only lowercase letters and "`.`" (dot) characters. All items (files and directories) directly inside the same parent directory must have unique names. Each item (file and directory) can be uniquely described by its *path* — a string built according to the following rules:

- Path of the root directory is simply "/" (forward slash).
- For a directory d, its path is obtained by concatenating the directory names top to bottom along the hierarchy from the root directory to d, preceding each name with the "/" character and placing another "/" character at the end of the path.
- For a file f, its path is the concatenation of the parent directory path and the name of file f.

We display the directory hierarchy by *printing* the root directory. We print a directory d by outputting a line of the form "$m_d \sqcup p_d \sqcup s_d$" where p_d and s_d are the path and size of directory d respectively, while m_d is its *expansion marker* explained shortly. If d contains other directories we must choose either to *collapse* it or to *expand* it. If we choose to expand d we print (using the same rules) all of its subdirectories in lexicographical order by name. If we choose to collapse directory d, we simply ignore its contents.

The expansion marker m_d is a single blank character when d does not have any subdirectories, "+" (plus) character when we choose to collapse d or a "-" (minus) character when we choose expand d.

Given a list of files in the filesystem and a threshold integer t, display the directory hierarchy ensuring that each directory of size at least t is printed. Additionally, the total number of directories printed should be minimal. Assume there are no empty directories in the filesystem — the entire hierarchy can be deduced from the provided file paths. Note that the root directory has to be printed regardless of its size. Also note that a directory of size at least t only has to be *printed*, but not necessarily *expanded*.

Input

The first line contains an integer n ($1 \le n \le 1\,000$) — the number of files. Each of the following n lines contains a string f and an integer s ($1 \le s \le 10^6$) — the path and the size of a single file. Each path is at most 100 characters long and is a valid file path according to the rules above. All paths will be different.

The following line contains an integer t ($1 \le t \le 10^9$) — the threshold directory size.

Output

Output the minimal display of the filesystem hierarchy for the given threshold as described above.

Example

input

```
9
/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
10000
```

output

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

input

```
8
/b/test/in.a 100
/b/test/in.b 1
/c/test/in.a 100
/c/test/in.b 1
/c/test/pic/in.a.svg 10
/c/test/pic/in.b.svg 10
/a/test/in.a 99
/a/test/in.b 1
101
```

output

```
- / 322
+ /a/ 100
- /b/ 101
  /b/test/ 101
- /c/ 121
+ /c/test/ 121
```

input

```
2
/a/a/a 100
/b.txt 99
200
```

output

```
+ / 199
```

Problem I: Intrinsic Interval

Time limit: 3 s
Memory limit: 512 MiB

Given a permutation π of integers 1 through n, an *interval* in π is a consecutive subsequence consisting of consecutive numbers. More precisely, for indices a and b where $1 \le a \le b \le n$, the subsequence $\pi_a^b = (\pi_a, \pi_{a+1}, \ldots, \pi_b)$ is an interval if sorting it would yield a sequence of consecutive integers. For example, in permutation $\pi = (3, 1, 7, 5, 6, 4, 2)$, the subsequence π_3^6 is an interval (it contains the numbers 4 through 7) while π_1^3 is not.

For a subsequence π_x^y its *intrinsic interval* is any interval π_a^b that contains the given subsequence ($a \le x \le y \le b$) and that is, additionally, as short as possible. Of course, the *length* of an interval is defined as the number of elements it contains.

Given a permutation π and m of its subsequences, find some intrinsic interval for each subsequence.

Input

The first line contains an integer n ($1 \le n \le 100\,000$) — the size of the permutation π. The following line contains n different integers $\pi_1, \pi_2, \ldots, \pi_n$ ($1 \le \pi_j \le n$) — the permutation itself.

The following line contains an integer m ($1 \le m \le 100\,000$) — the number of subsequences. The j-th of the following m lines contains integers x_j and y_j ($1 \le x_j \le y_j \le n$) — the endpoints of the j-th subsequence.

Output

Output m lines. The j-th line should contain two integers a_j and b_j where $1 \le a_j \le b_j \le n$ — the endpoints of some intrinsic interval of the j-th subsequence $\pi_{x_j}^{y_j}$.

Example

input

```
7
3 1 7 5 6 4 2
3
3 6
7 7
1 3
```

output

```
3 6
7 7
1 7
```

input

```
10
2 1 4 3 5 6 7 10 8 9
5
2 3
3 7
4 7
4 8
7 8
```

output

```
1 4
3 7
3 7
3 10
7 10
```

Problem J: Justified Jungle

Time limit: 6 s
Memory limit: 512 MiB

As you probably know, a *tree* is a graph consisting of n nodes and $n-1$ undirected edges in which any two nodes are connected by exactly one path. A *forest* is a graph consisting of one or more trees. In other words, a graph is a forest if every connected component is a tree. A forest is *justified* if all connected components have the same number of nodes.

Given a tree G consisting of n nodes, find all positive integers k such that a justified forest can be obtained by erasing exactly k edges from G. Note that erasing an edge never erases any nodes. In particular when we erase all $n-1$ edges from G, we obtain a justified forest consisting of n one-node components.

Input

The first line contains an integer n ($2 \le n \le 1\,000\,000$) — the number of nodes in G. The k-th of the following $n-1$ lines contains two different integers a_k and b_k ($1 \le a_k, b_k \le n$) — the endpoints of the k-th edge.

Output

The first line should contain all wanted integers k, in increasing order.

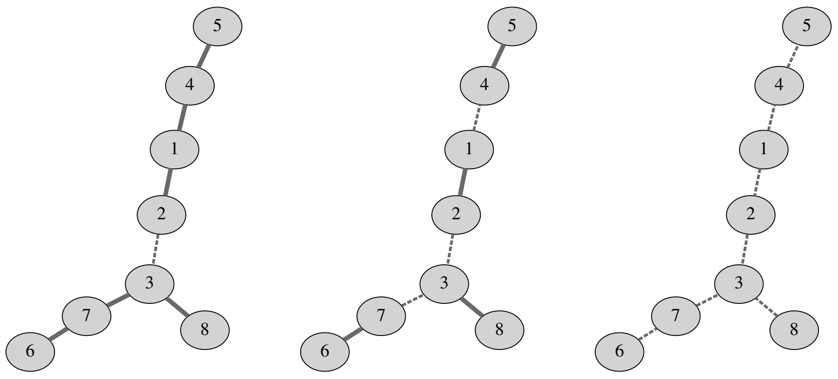
Example

input

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

output

```
1 3 7
```

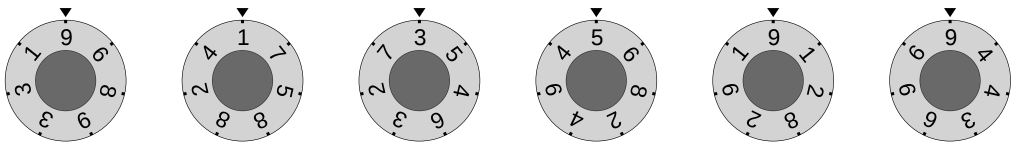


Figures depict justified forests obtained by erasing 1, 3 and 7 edges from the tree in the example input.

Problem K: Kitchen Knobs

Time limit: 3 s
Memory limit: 512 MiB

You are cooking on a gigantic stove at a large fast-food restaurant. The stove contains n heating elements arranged in a line and numbered with integers 1 through n left to right. Each element is operated by its *control knob*. The knobs are a bit unusual: each knob is marked with seven non-zero digits evenly distributed around a circle. The *power* of the heating element is equal to the positive integer obtained by reading the digits on its control knob clockwise starting from the top of the knob.



Initial positions of the control knobs in the first example input below.

In a single step, you can rotate one or more *consecutive* knobs by any number of positions in any direction. However, all knobs rotated in one step need to be rotated by the same number of positions in the same direction.

Find the smallest number of steps needed to set all the heating elements to maximal possible power.

Input

The first line contains an integer n ($1 \leq n \leq 501$) — the number of heating elements. The j-th of the following n lines contains an integer x_j — the initial power of the j-th heating element. Each x_j consists of exactly seven non-zero digits.

Output

Output a single integer — the minimal number of steps needed.

Example

input

```
6
9689331
1758824
3546327
5682494
9128291
9443696
```

output

```
3
```

input

```
7
5941186
3871463
8156346
9925977
8836125
9999999
5987743
```

output

```
2
```

In the first example, one of the ways to achieve maximal possible power is: rotate knobs 2 through 3 by 3 positions in the counterclockwise direction, rotate knob 3 by 3 positions in the counterclockwise direction, and rotate knobs 4 through 6 by 2 positions in the clockwise direction.

Problem L: Lunar Landscape

Time limit: 2 s
Memory limit: 512 MiB

A satellite is surveying a possible rover landing area on the moon. The landing area is modeled as a square grid embedded in the standard coordinate system.

The satellite has taken n photos, each capturing a square area of the surface. Careful camera calibration has ensured that all photos are aligned with the grid — all four vertices have integer coordinates. Due to the satellite's changing orbit there are two types of photos:

- Photos of type `A` have sides that are parallel to coordinate axes. Such a photo is specified by giving the integer coordinates (x, y) of the square's middle point and the length of its side a — always an even integer.
- Photos of type `B` have sides at a $45°$ angle to the coordinate axes. Such a photo is specified by giving the integer coordinates (x, y) of the square's middle point and the length of its diagonal d — always an even integer.

Find the total surface area captured in the satellite photos.

Input

The first line contains an integer n ($1 \le n \le 200\,000$) — the number of photos. The j-th of the following n lines is either of the form "`A` x_j y_j a_j" or "`B` x_j y_j d_j" representing a photo of type `A` or `B`, respectively. The x_j and y_j are the integer coordinates of the middle point of the photo ($-1\,000 \le x_j, y_j \le 1\,000$). The a_j and d_j are even integers ($2 \le a_j, d_j \le 1\,000$) — the side length and the diagonal length, respectively.

Output

Output a number with exactly two digits after the decimal point — the total area of the surface. The answer has to exactly correspond to the judge's solution (no rounding errors are tolerated).

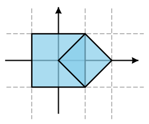
Example

input

```
2
A 0 0 2
B 1 0 2
```

output

```
5.00
```

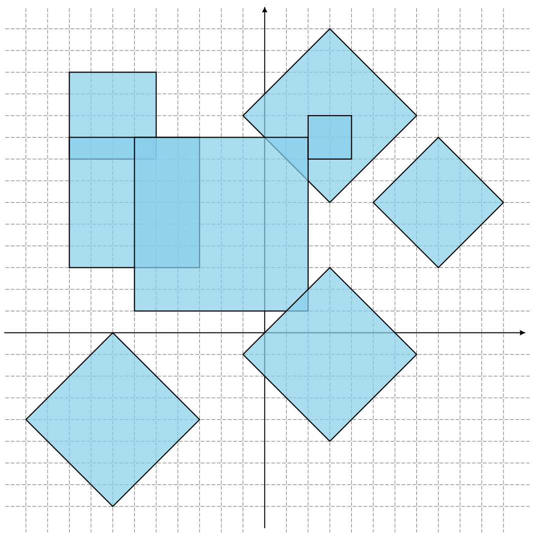


input

```
8
A -7 10 4
B 3 10 8
A -6 6 6
A -2 5 8
B 3 -1 8
B -7 -4 8
A 3 9 2
B 8 6 6
```

output

```
205.50
```



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 <= 10 000 000, 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:

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

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:

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

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

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:

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

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:

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

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

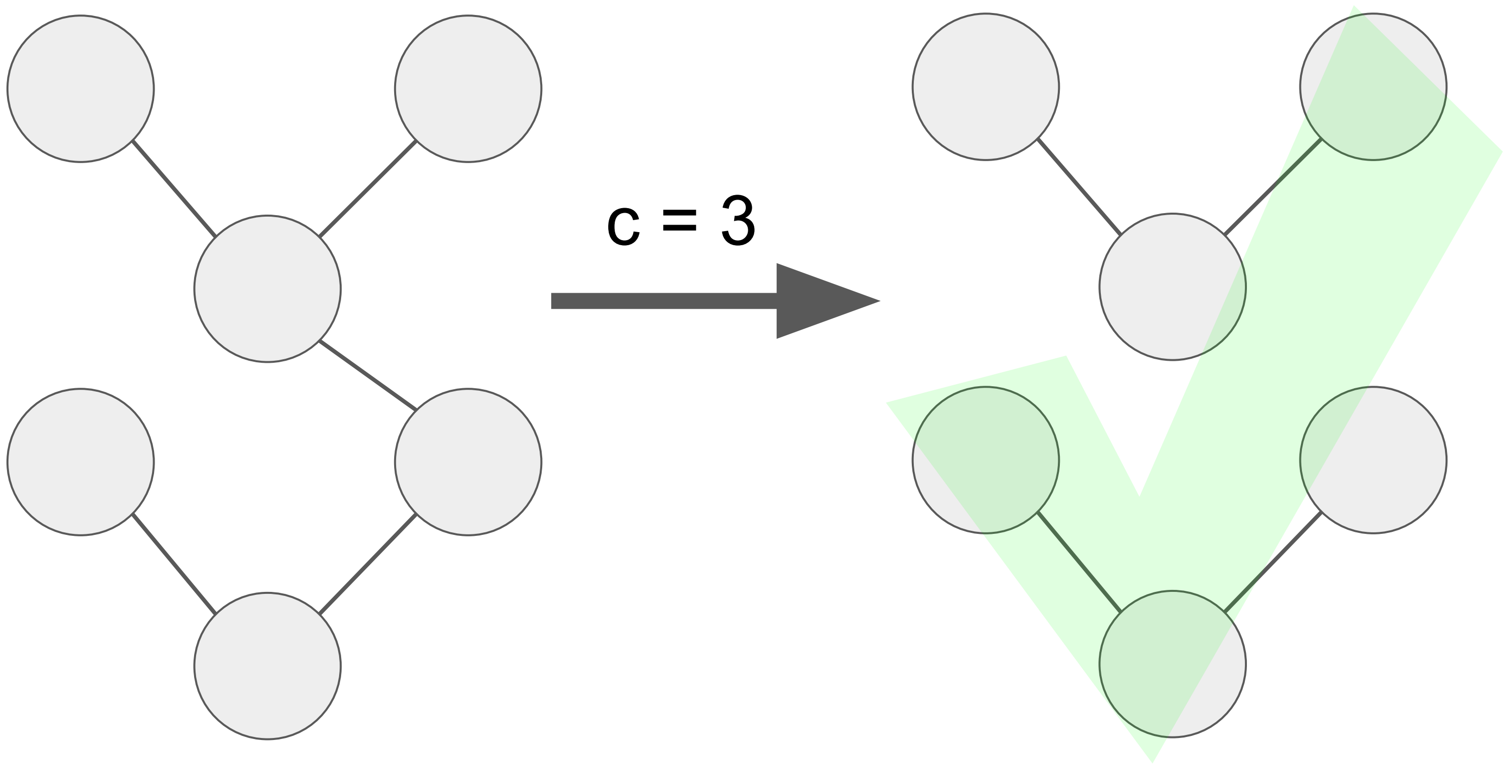
Problem J
Justified Jungle

Submits: 203
Accepted: at least 17

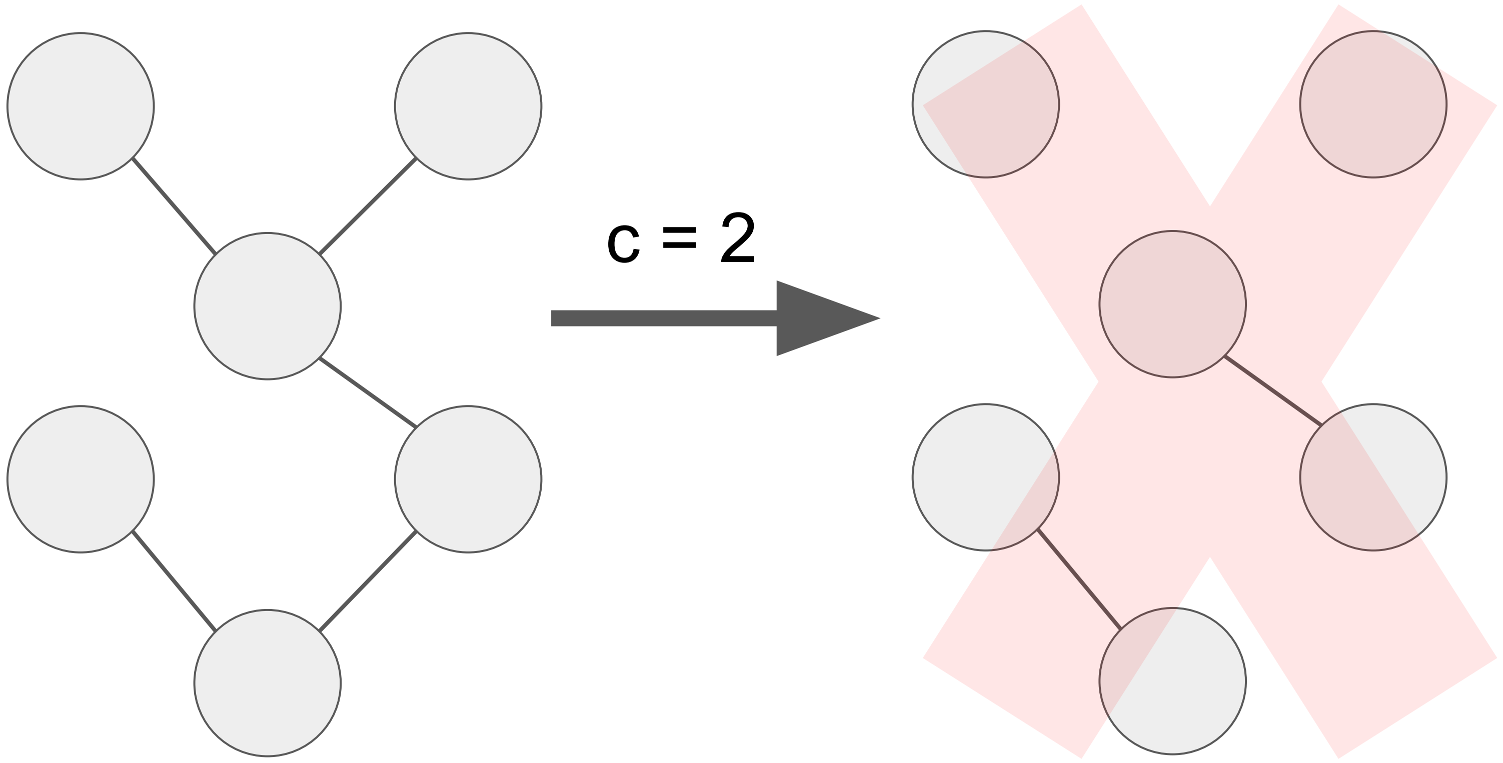
First solved by: Jagiellonian 1
Jagiellonian University in Krakow
(Hlembotskyi, Stokowacki, Zieliński)
00:16:32

Author: Luka Kalinovčić, Ivan Katanić

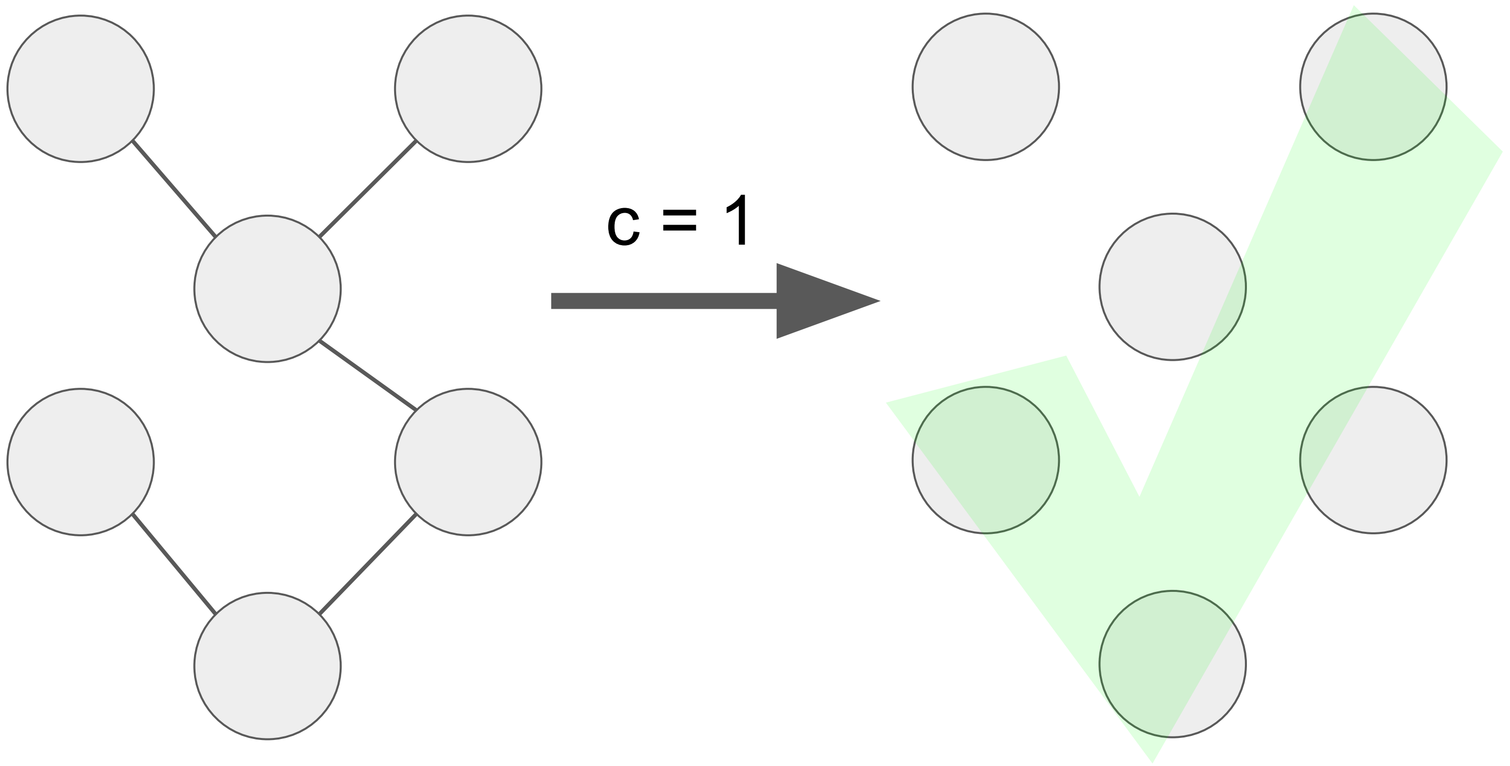
Problem: Given a tree, find all integers c, such that we can cut a tree into components of size c.



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.



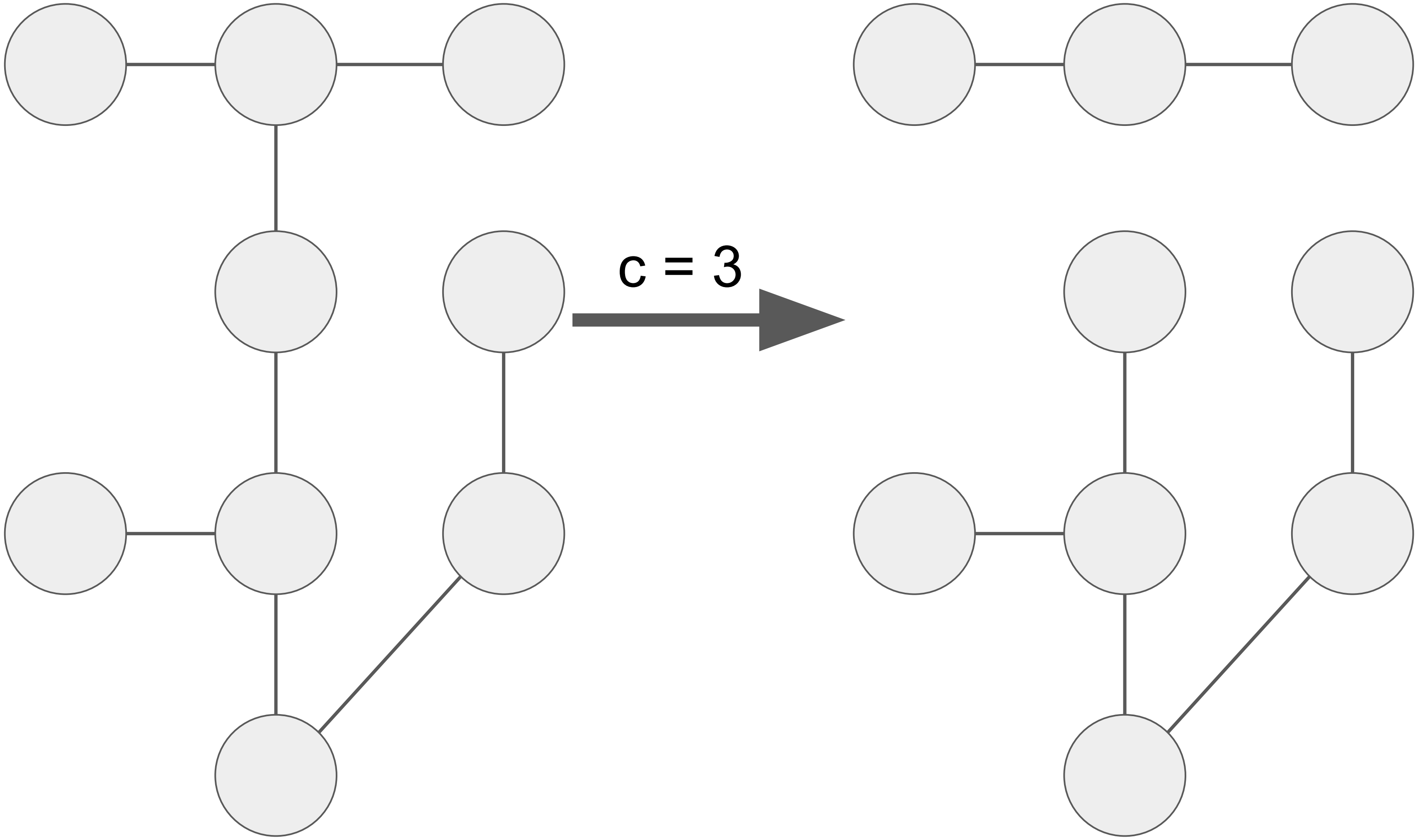
Problem: Given a tree, find all integers c, such that we can cut a tree into components of size c.

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.

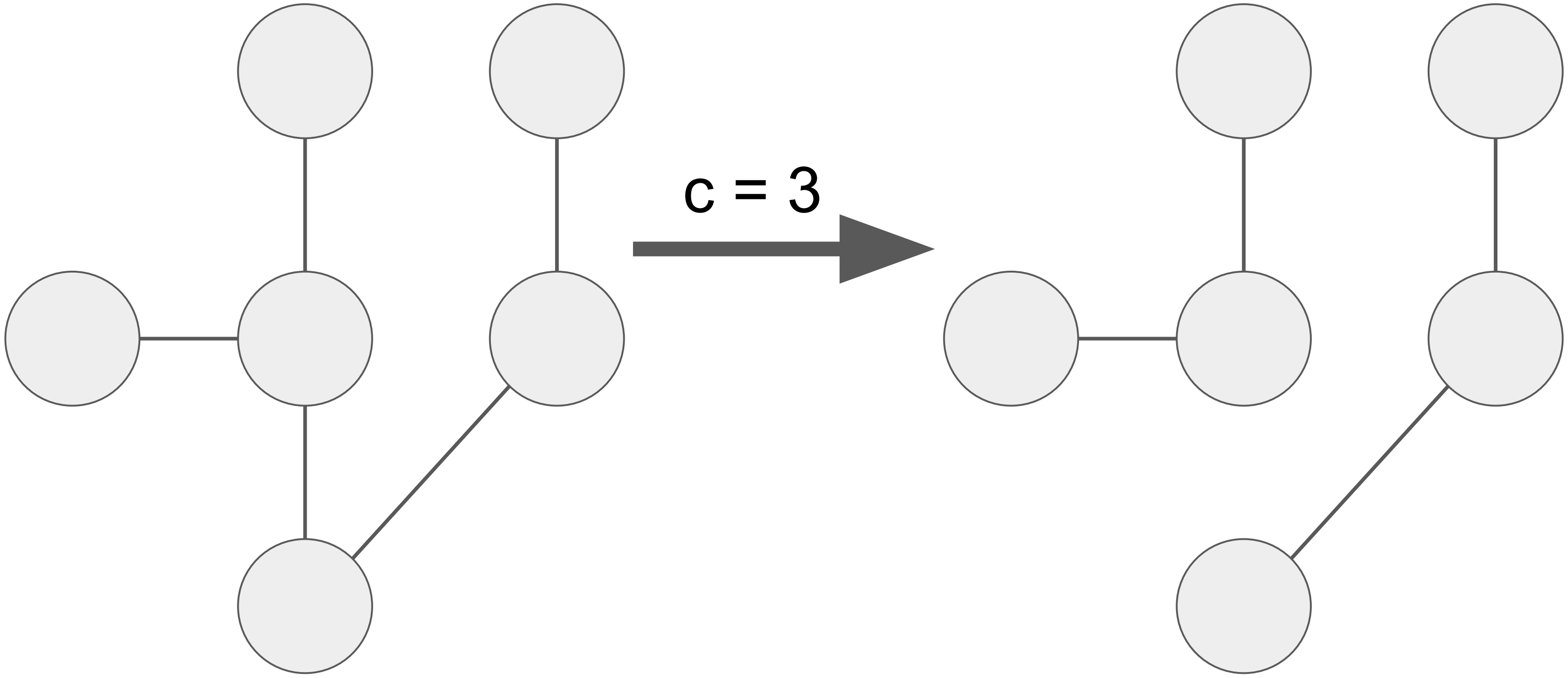
Problem: Given a tree of size n and integer c, such that $c \mid n$, can we cut it into components of size c?

```
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 $n - c$.
```

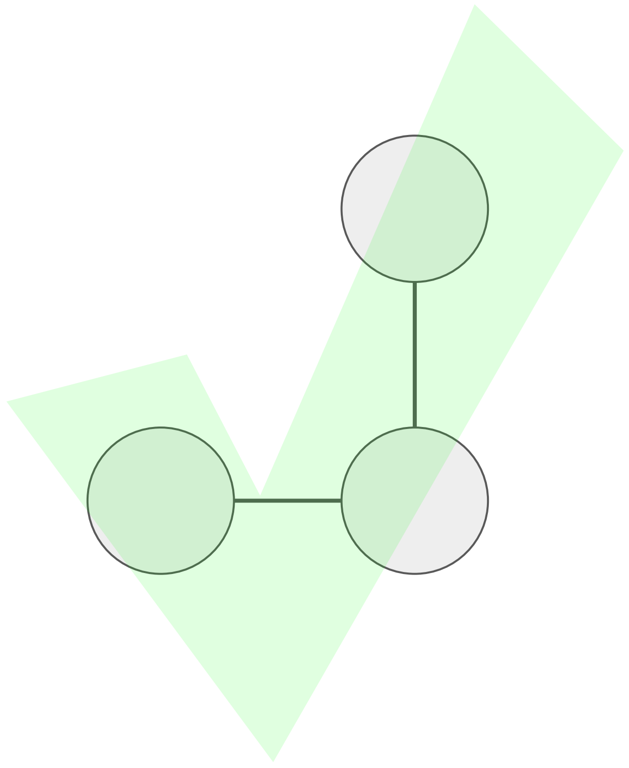
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



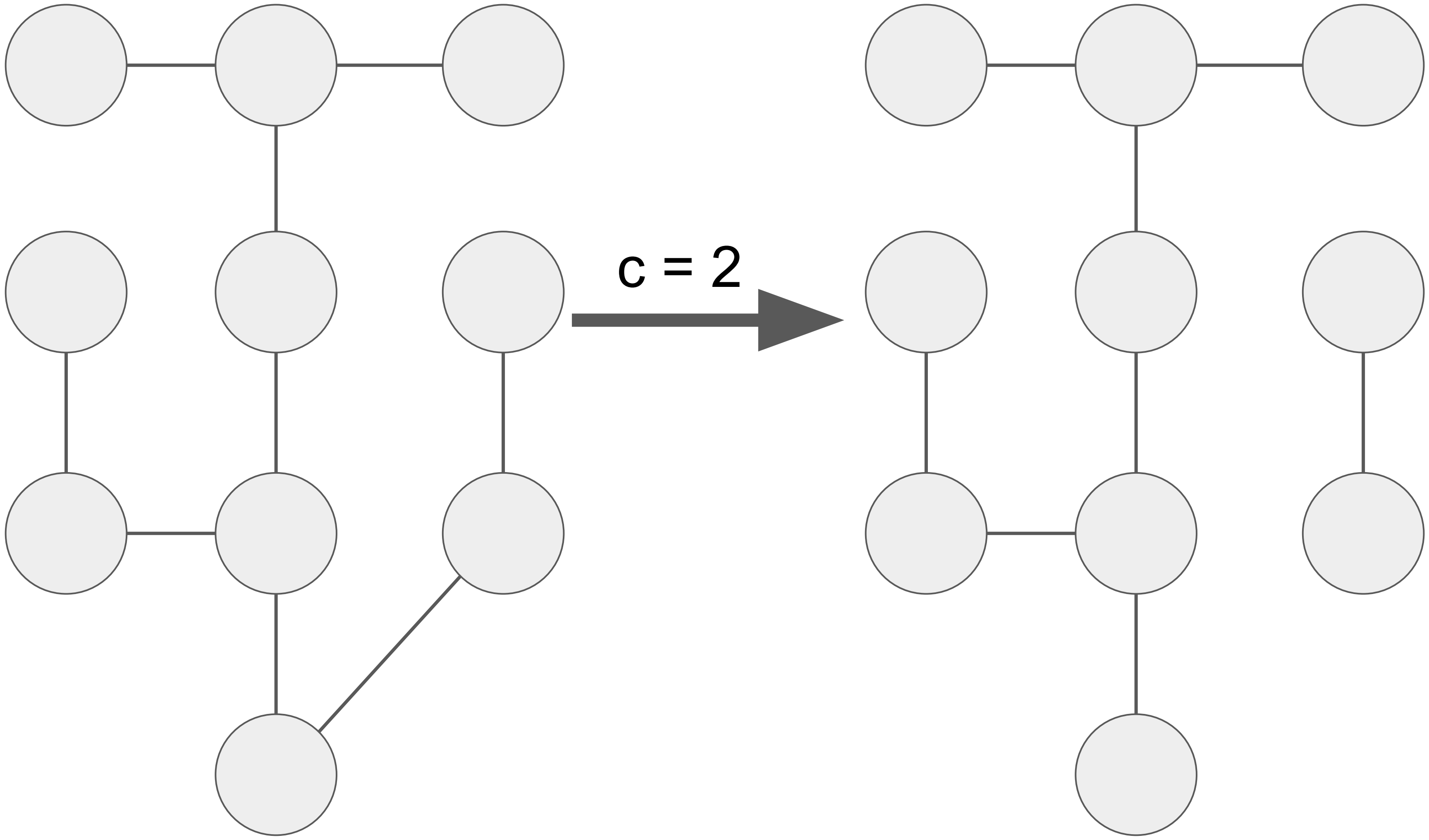
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



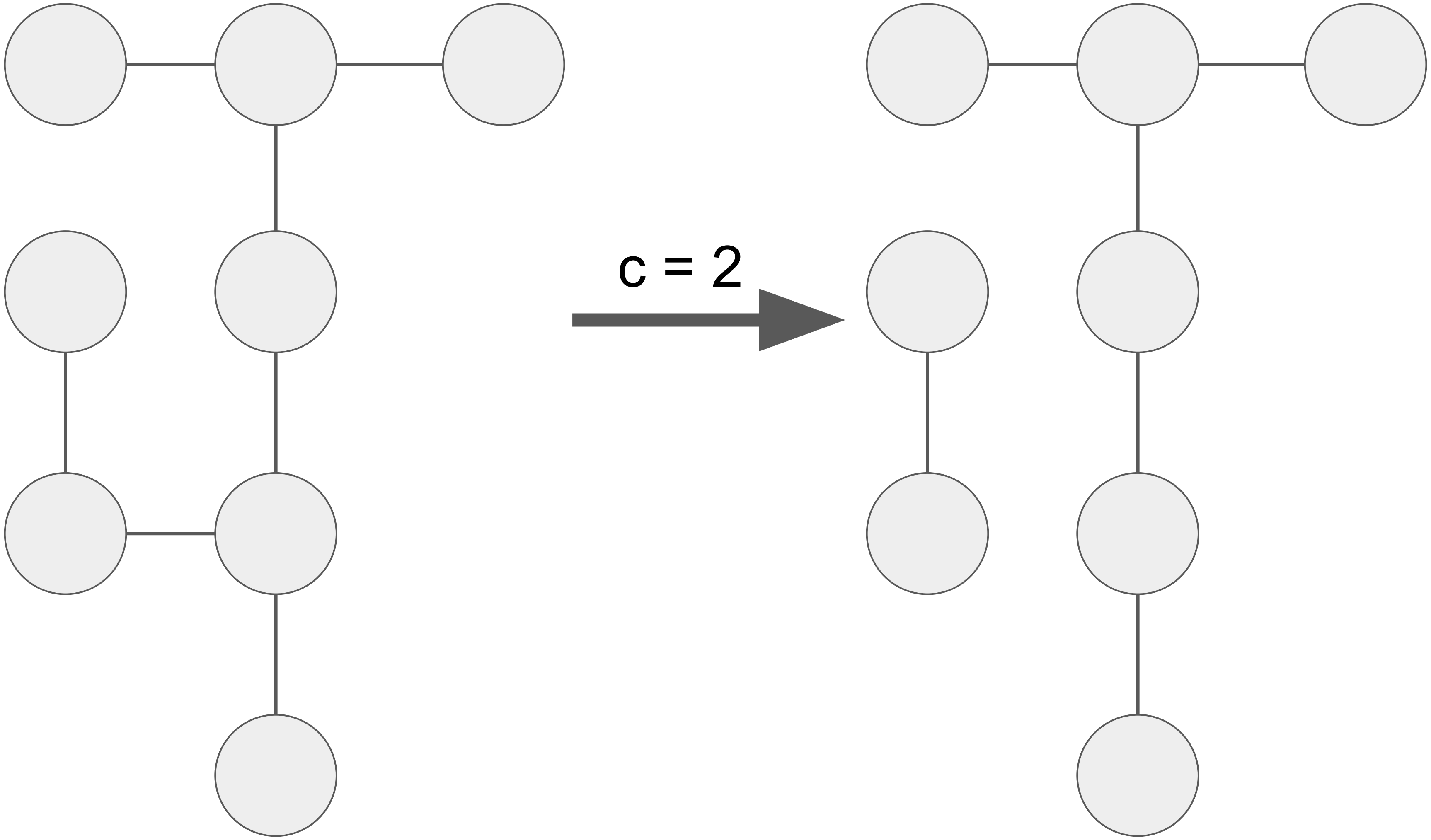
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



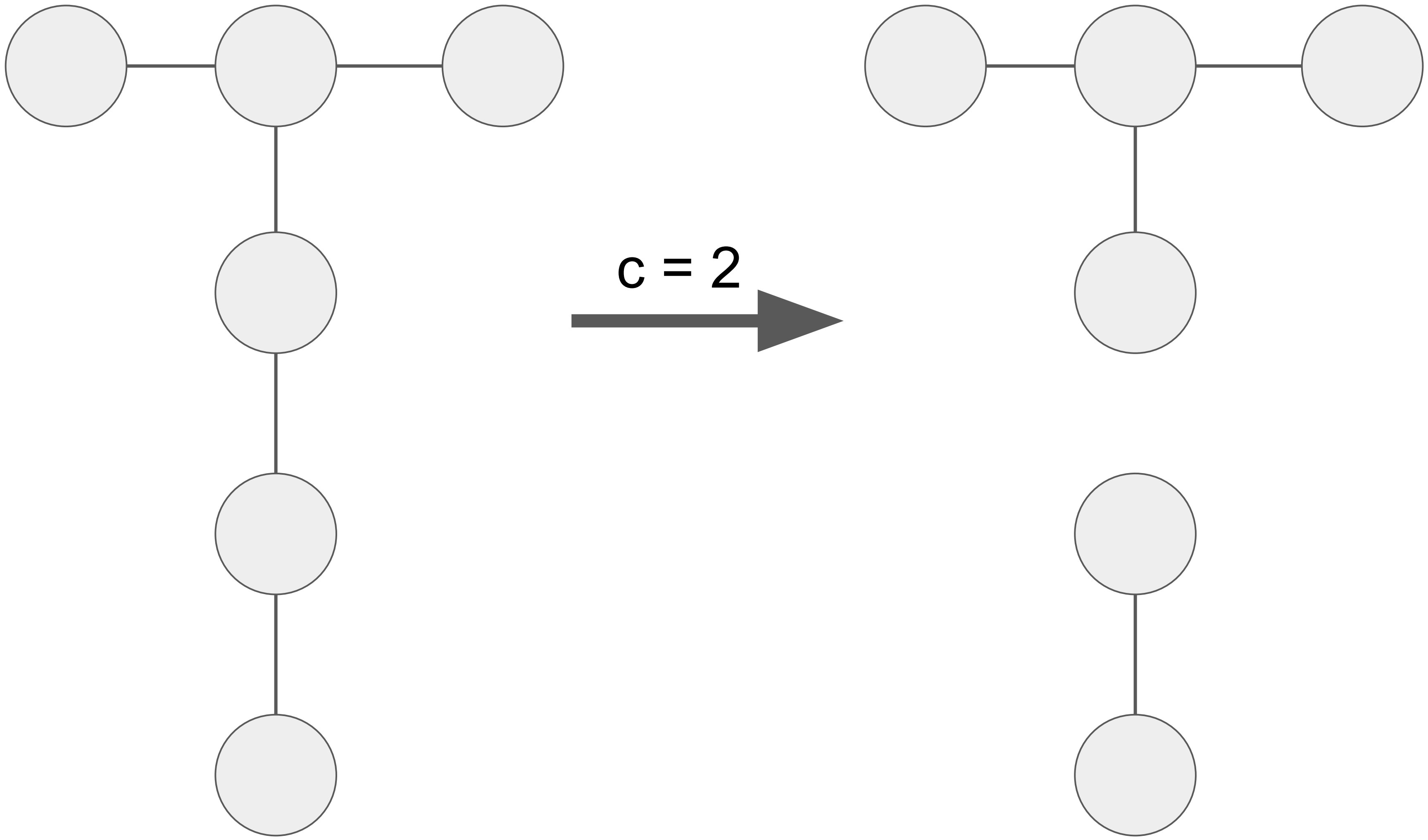
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



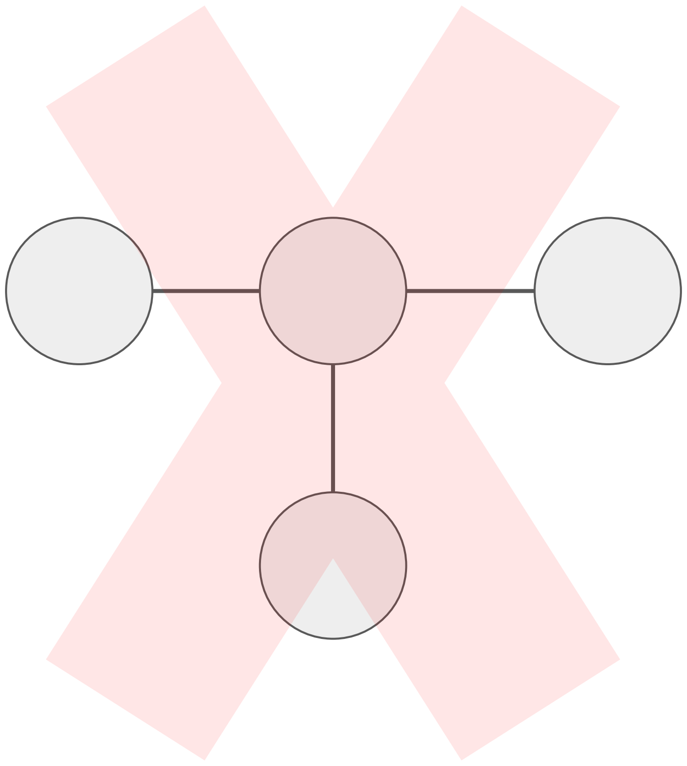
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?



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?



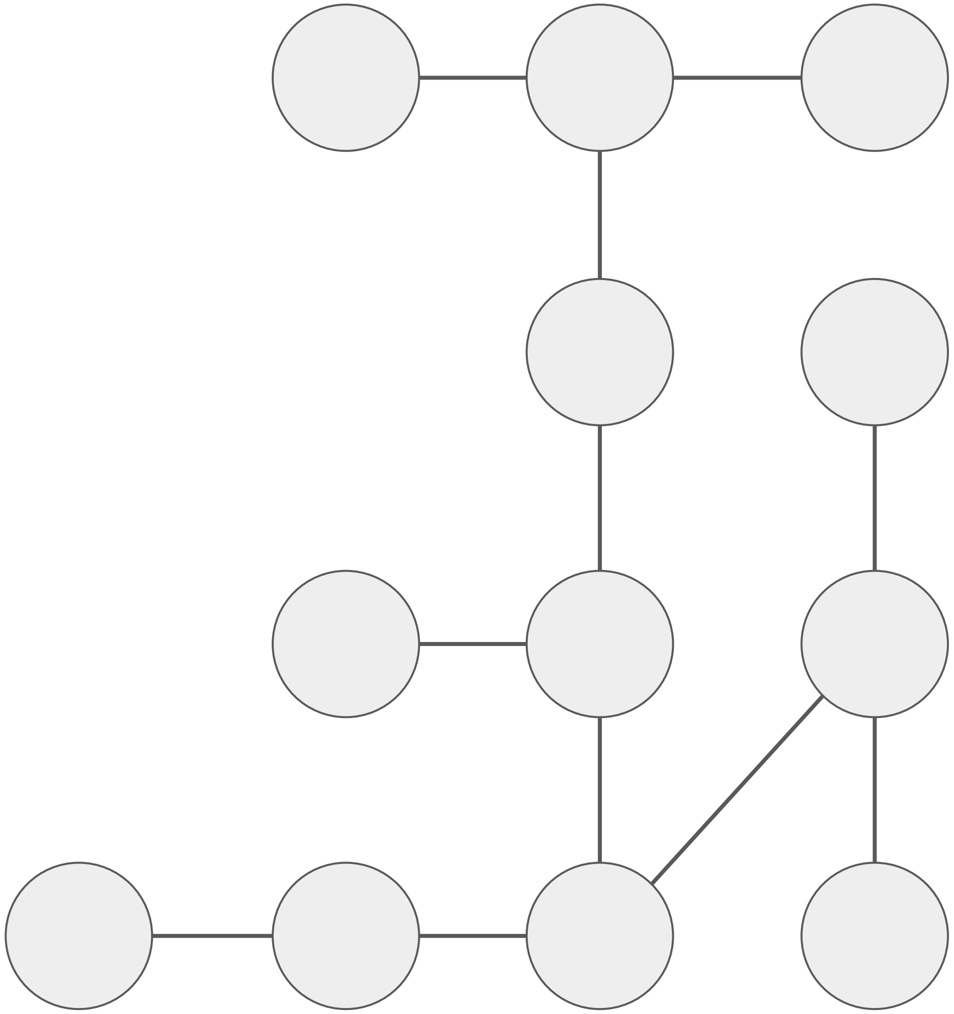
Problem: Given a tree of size n and integer c, such that c | n, can we cut it into components of size c?

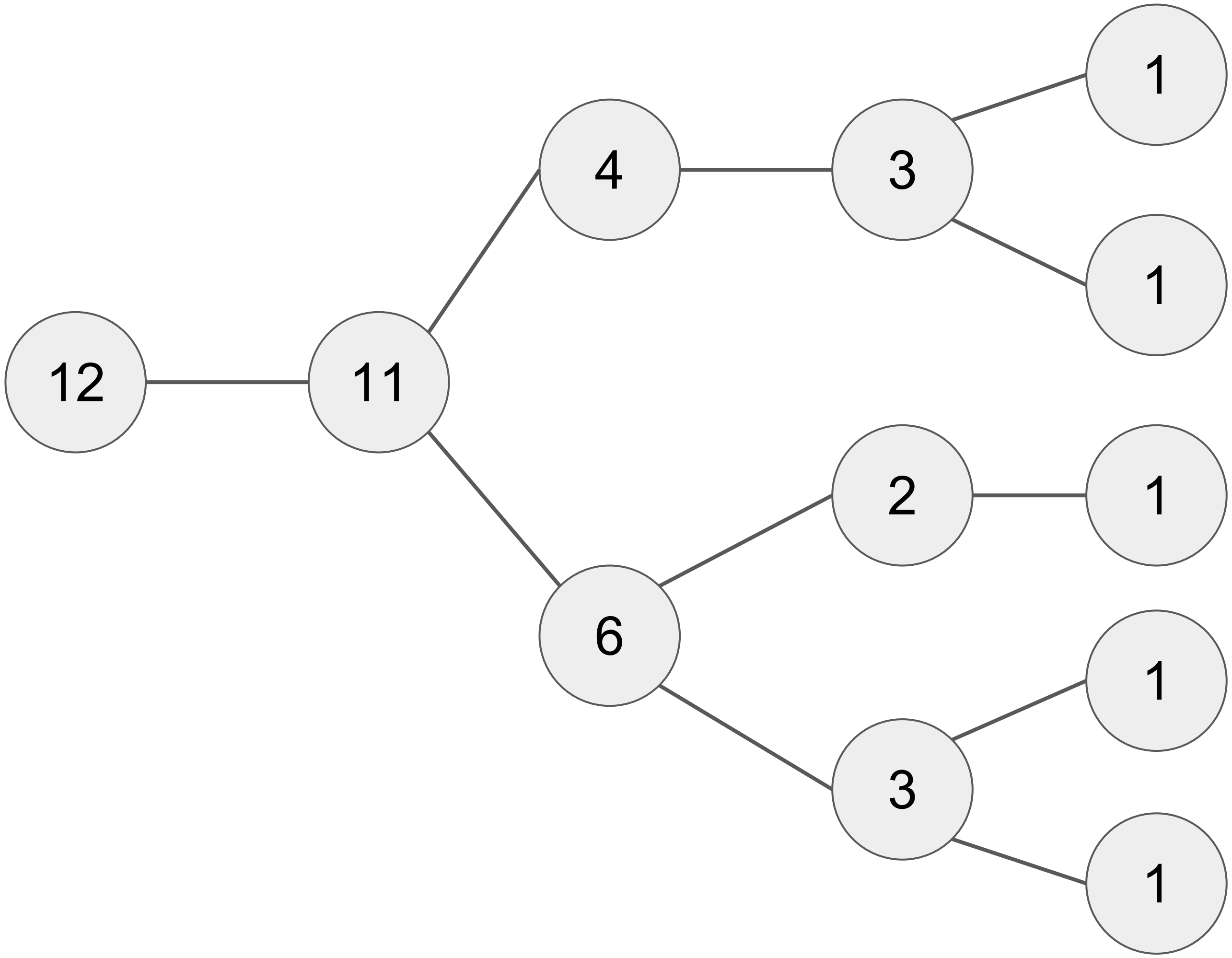
Iterative algorithm is difficult to implement in O(n), and might time out.

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

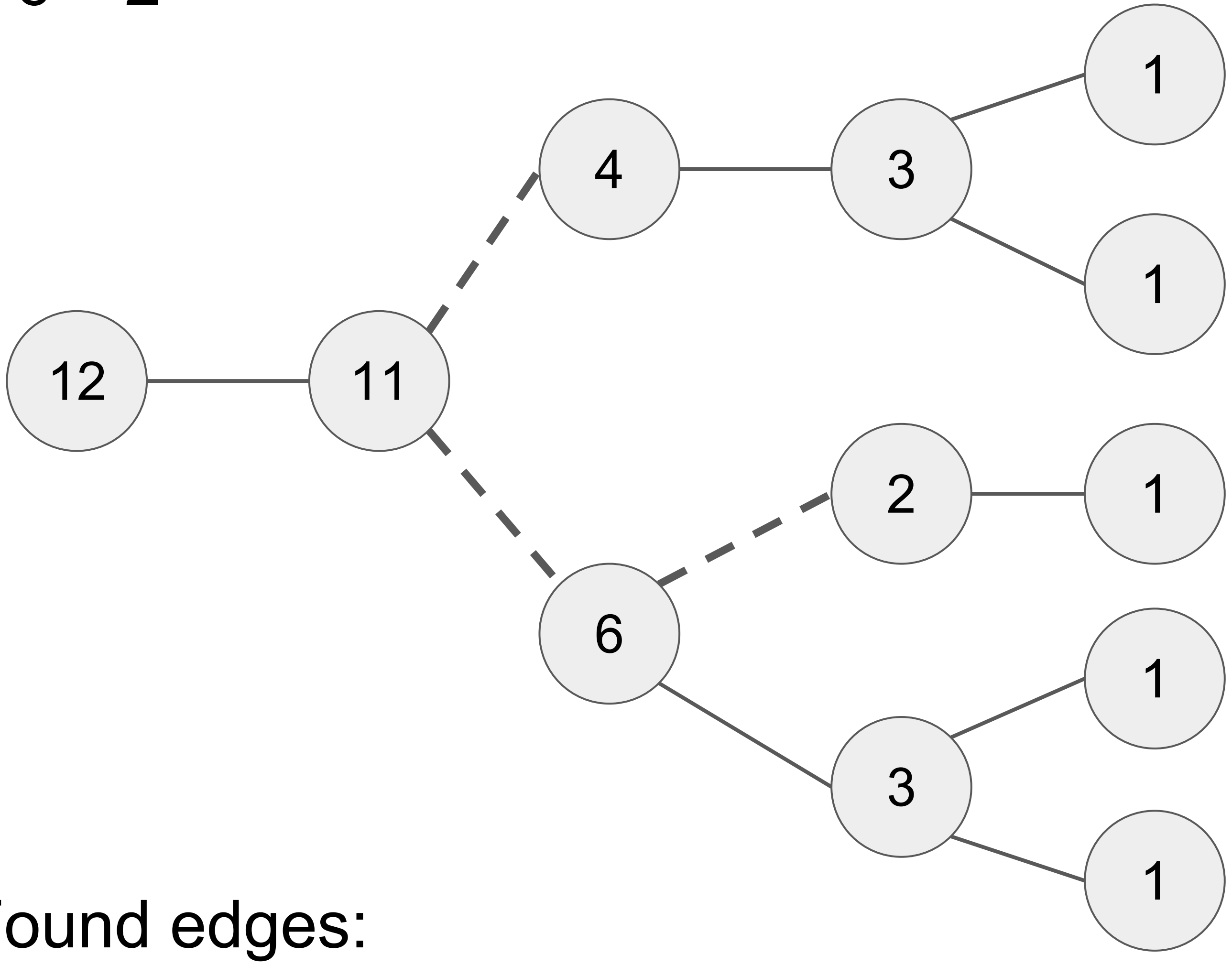
Find edges with subtrees sizes equal to a multiple of c. Those are the ones we'll end up cutting.

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



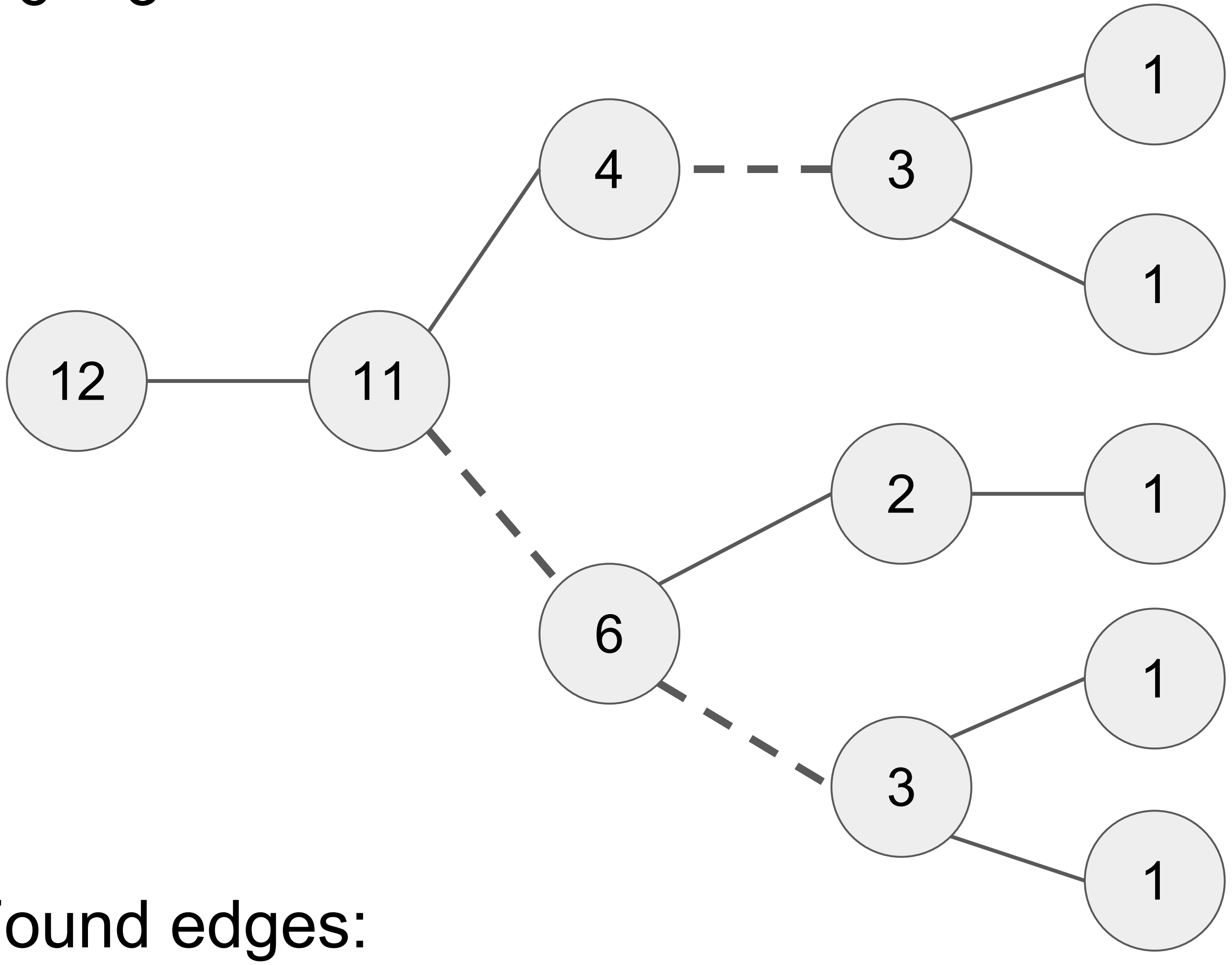

12
11
4
3
1
1
6
2
1
3
1
1

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.

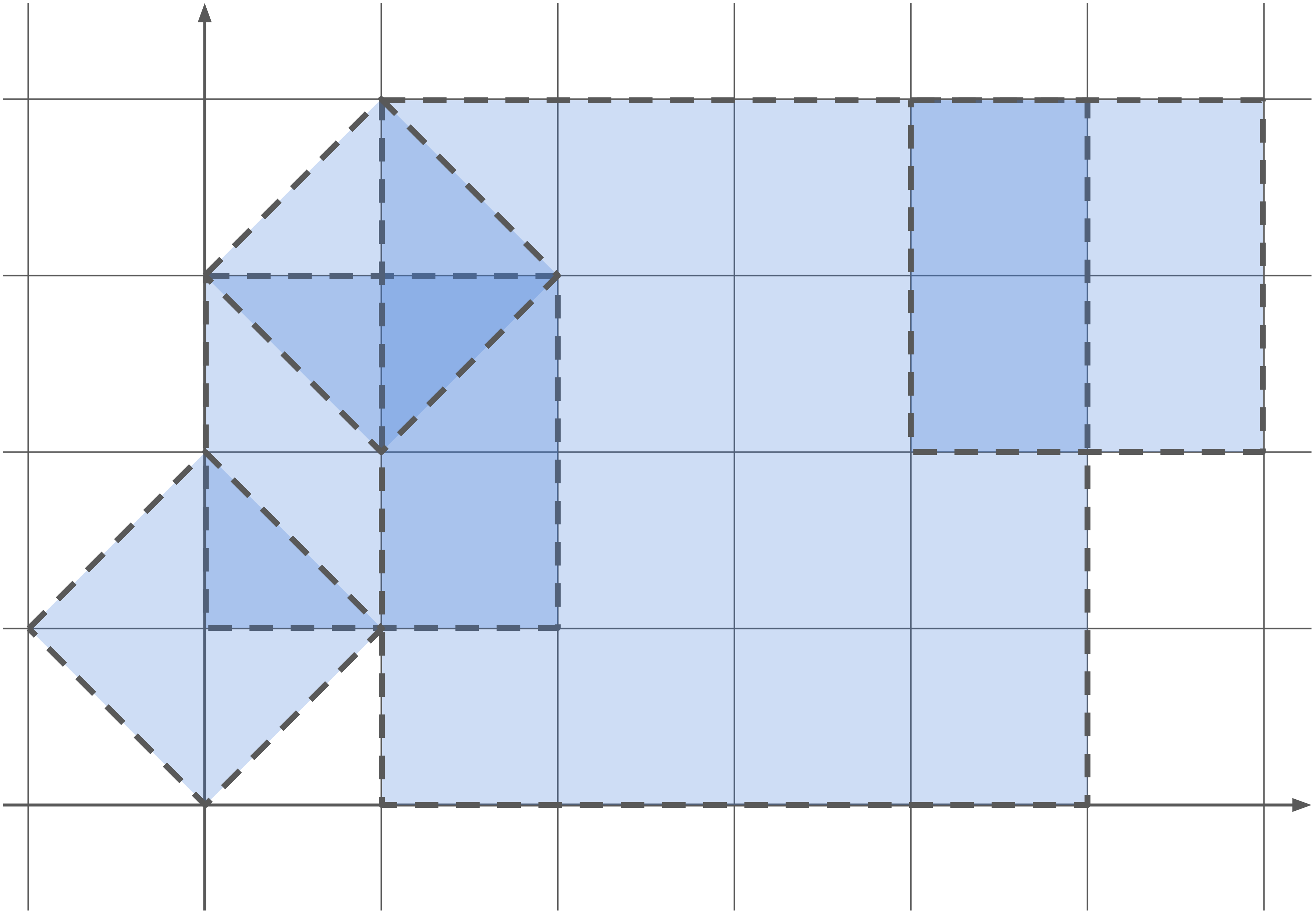
$O(n + \sigma(n)^2)$ is possible with an extra insight.

Problem L
Lunar Landscape

Submits: 41
Accepted: at least 5

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

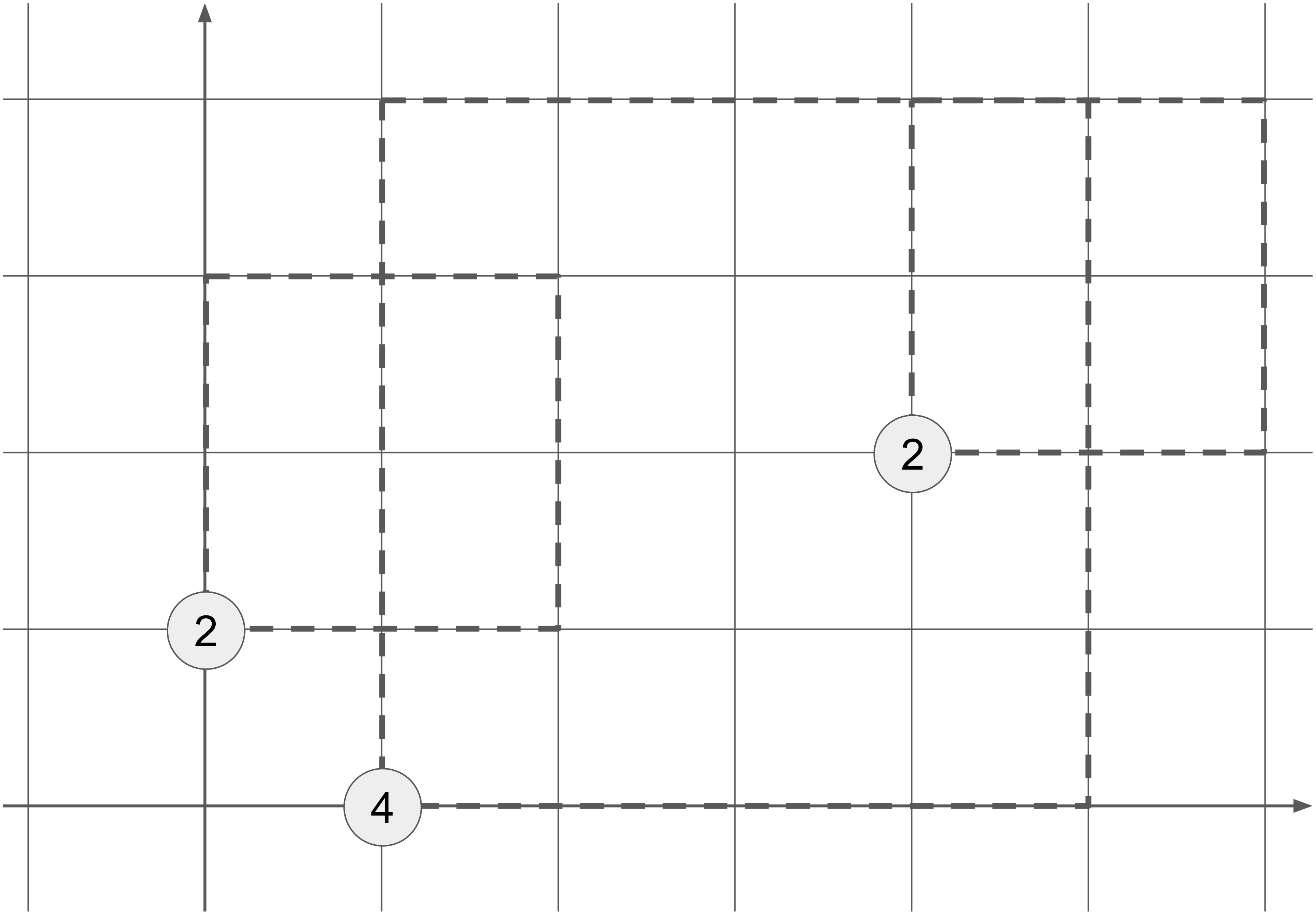
Author: Luka Kalinovčić

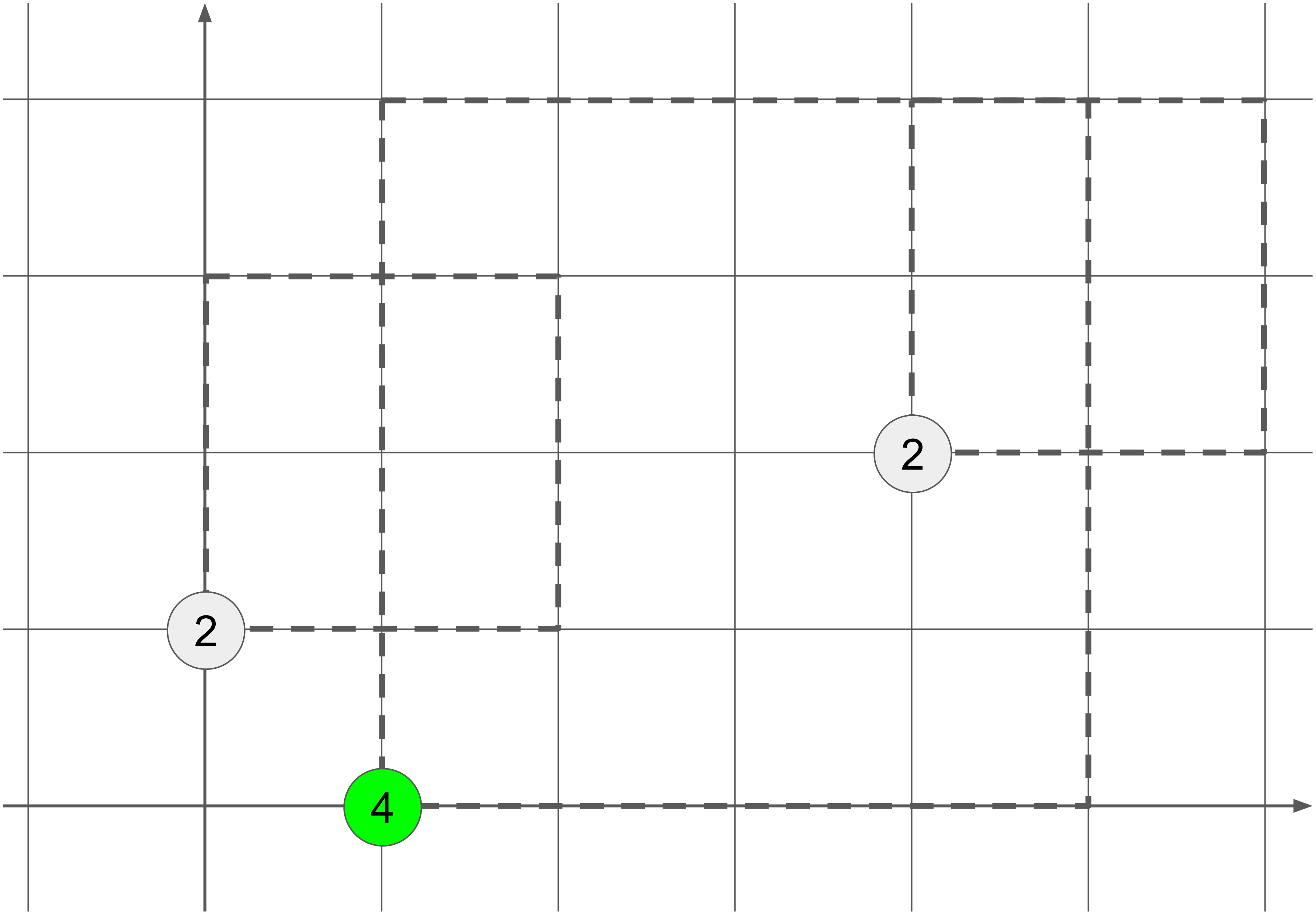


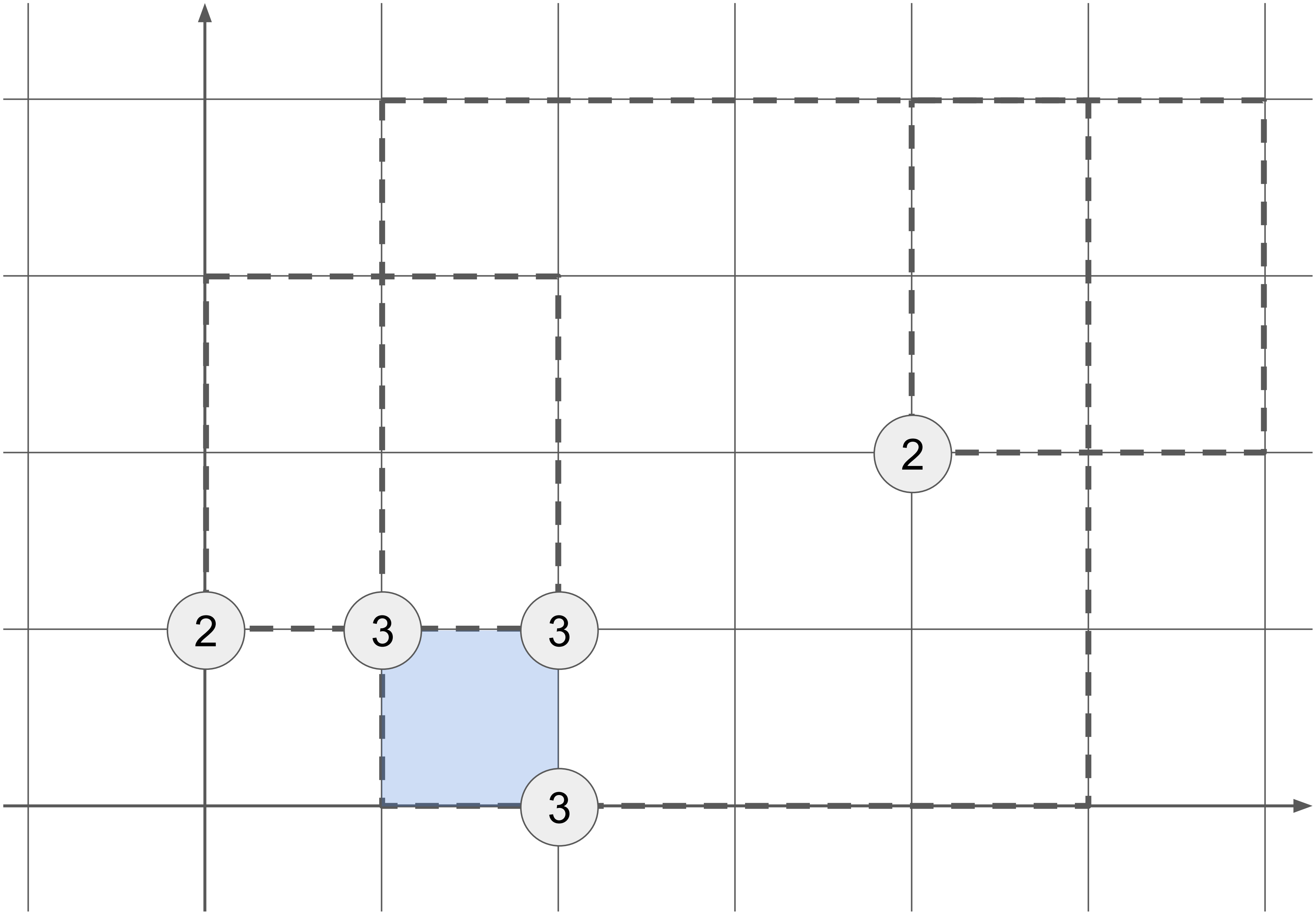
Key observation: The grid is small enough to iterate over each unit square and “paint it blue” in memory.

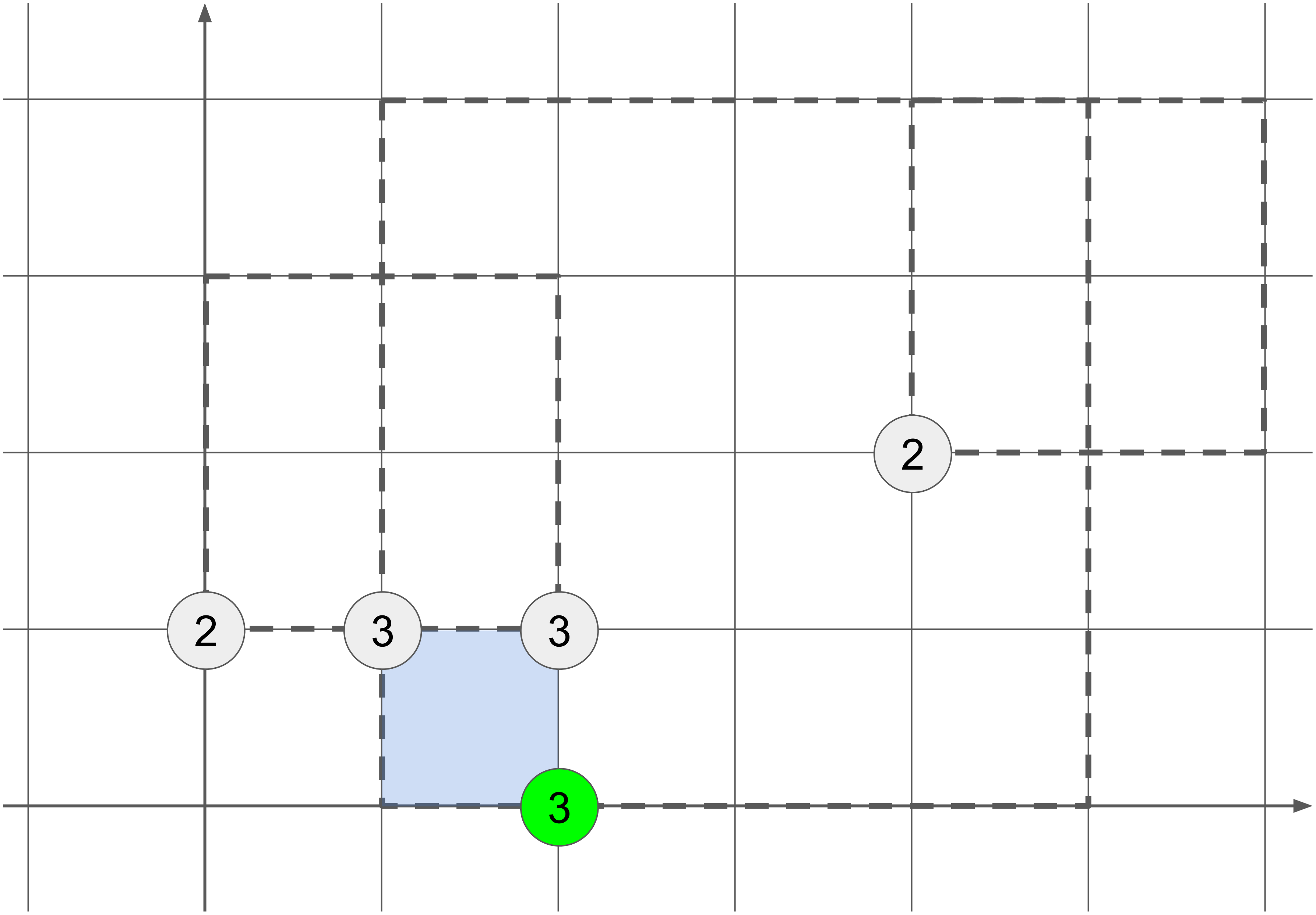
However, the naive algorithm that iterates through each unit square of each frame is too slow.

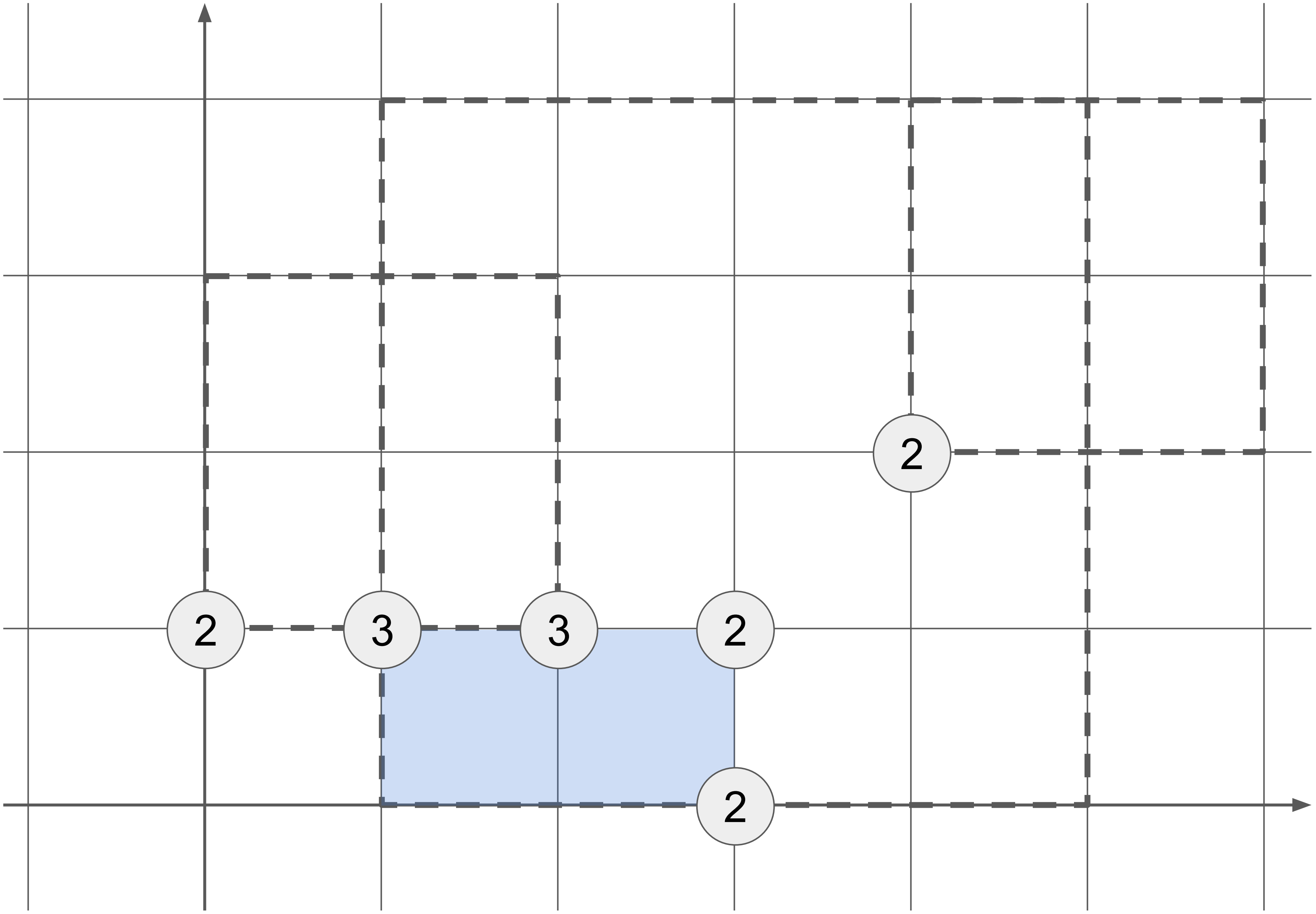
Instead, let’s first place a “bucket full of paint” in a corner of each frame.
Then, we’ll sweep across the grid and whenever we encounter a bucket, we’ll paint one unit square and propagate the bucket to neighbouring squares.

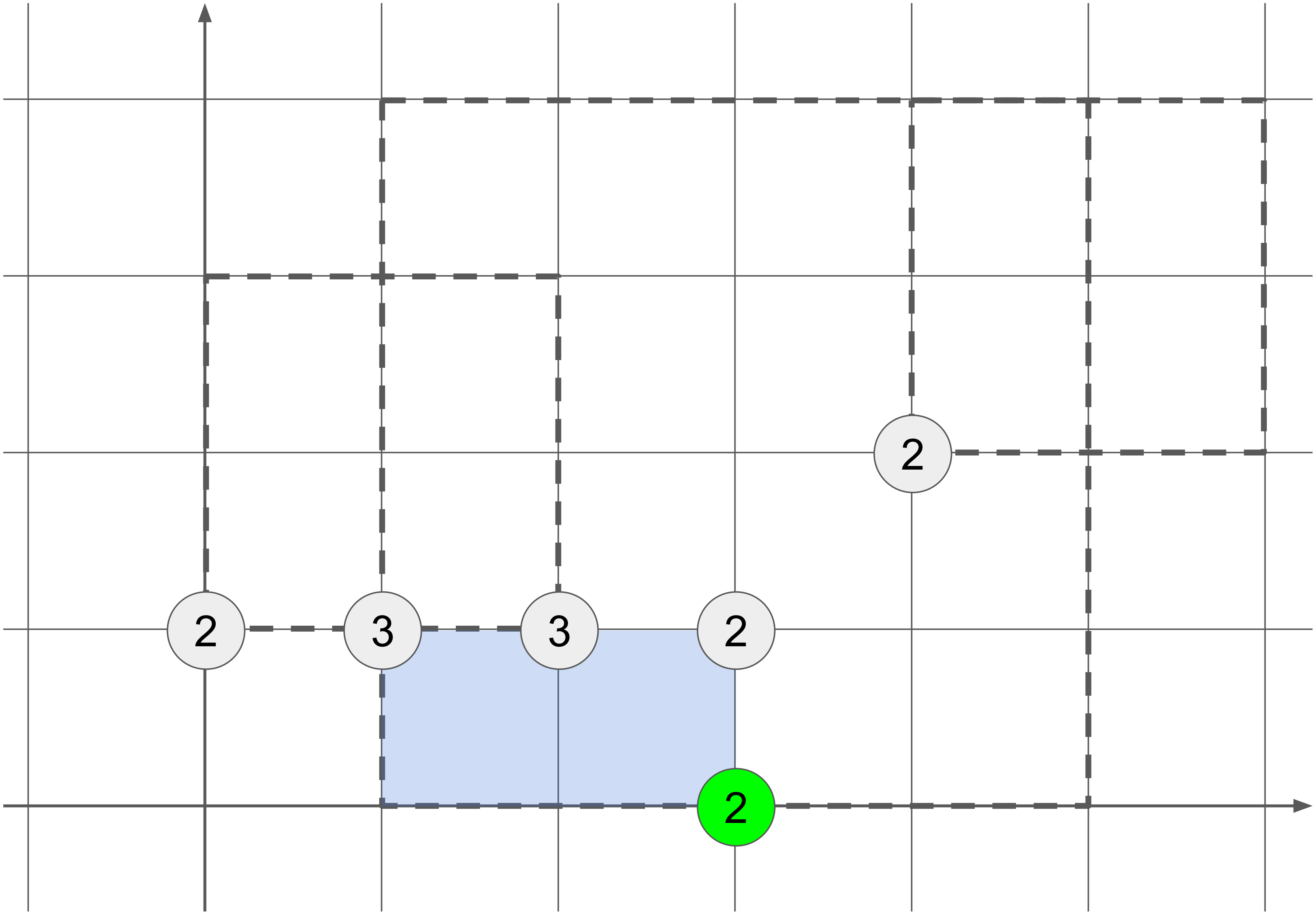

2
2
4

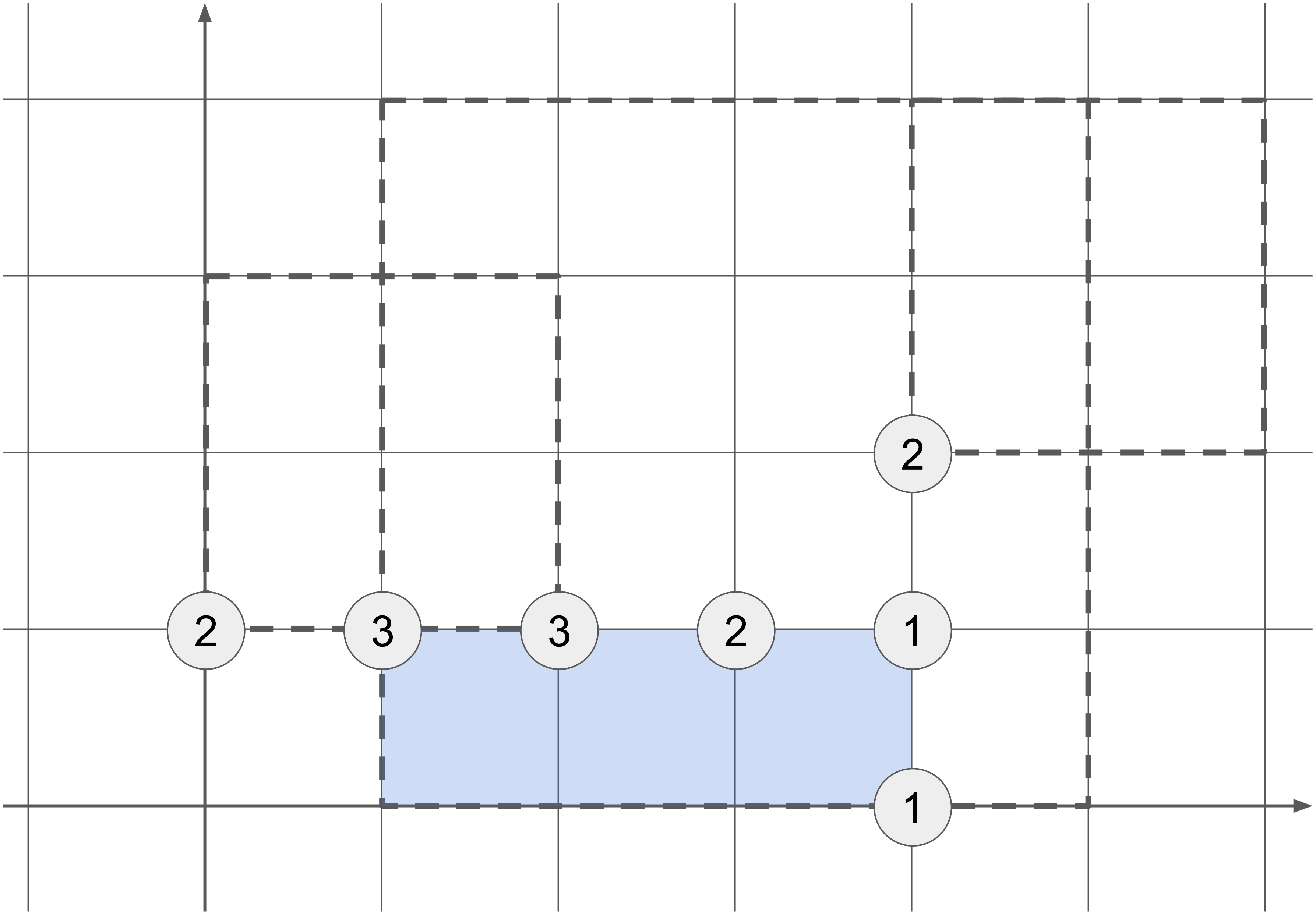

2
2
4

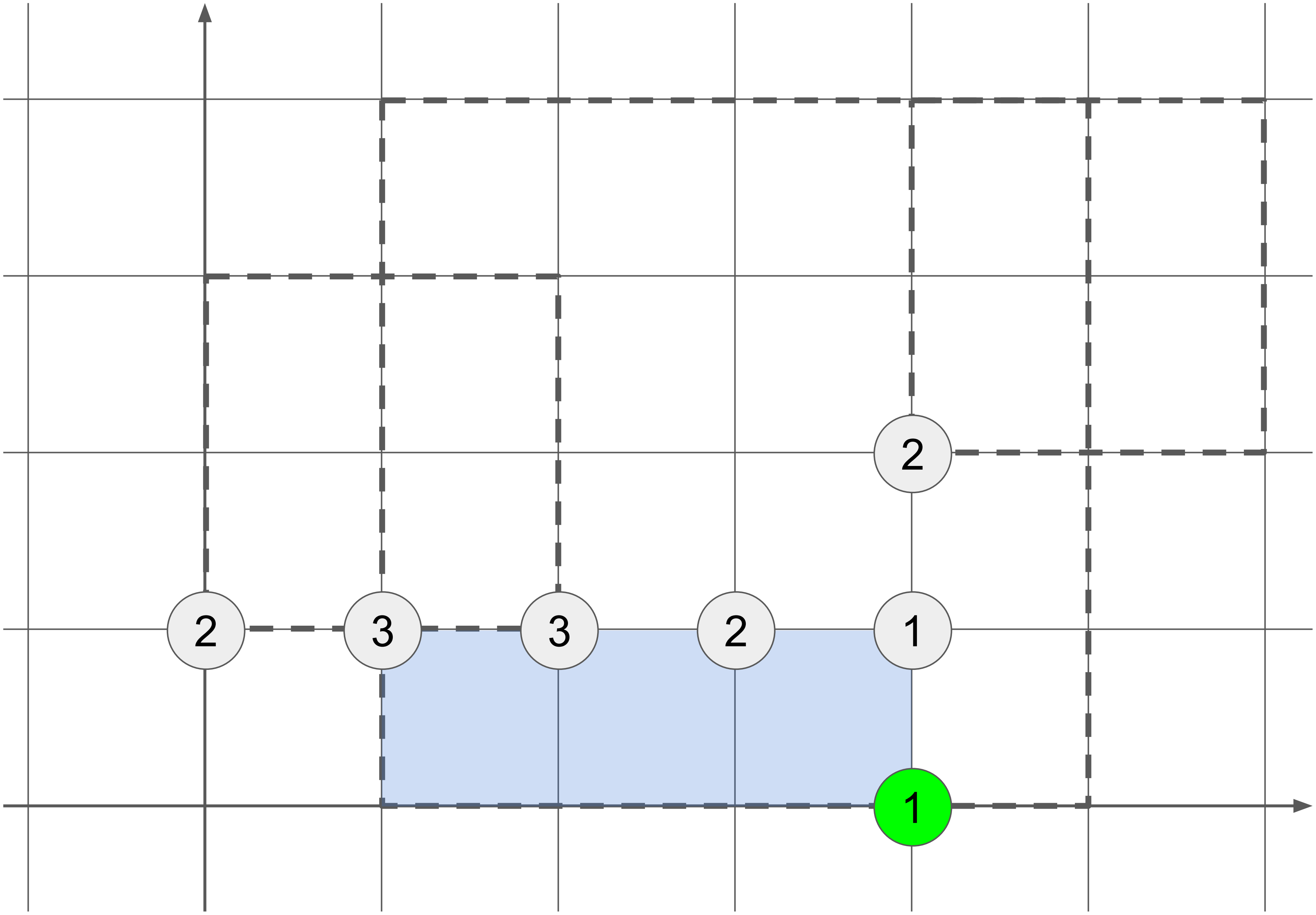

2
3
3
3
2

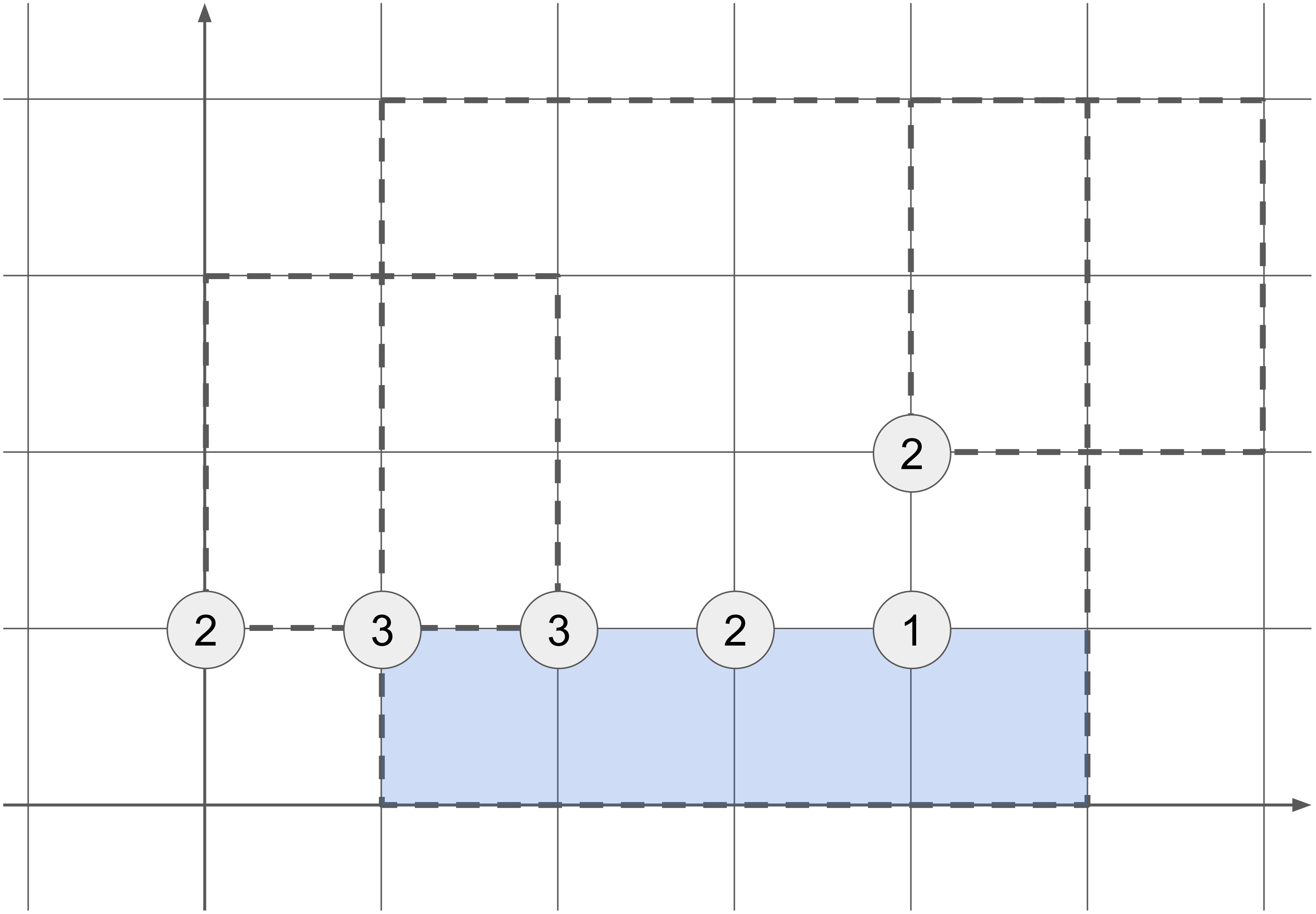

2
2
3
3
3

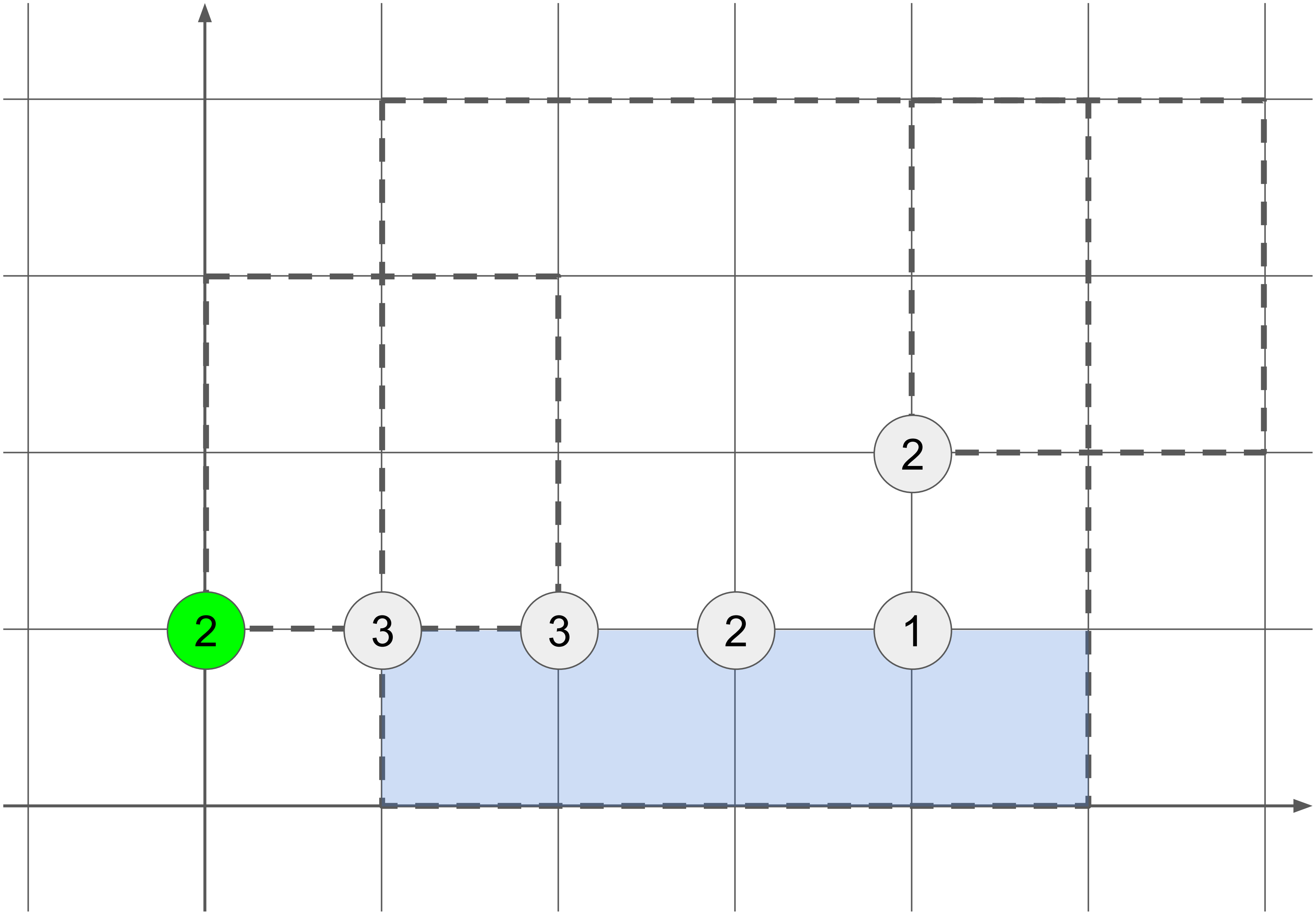

2
3
3
2
2
2

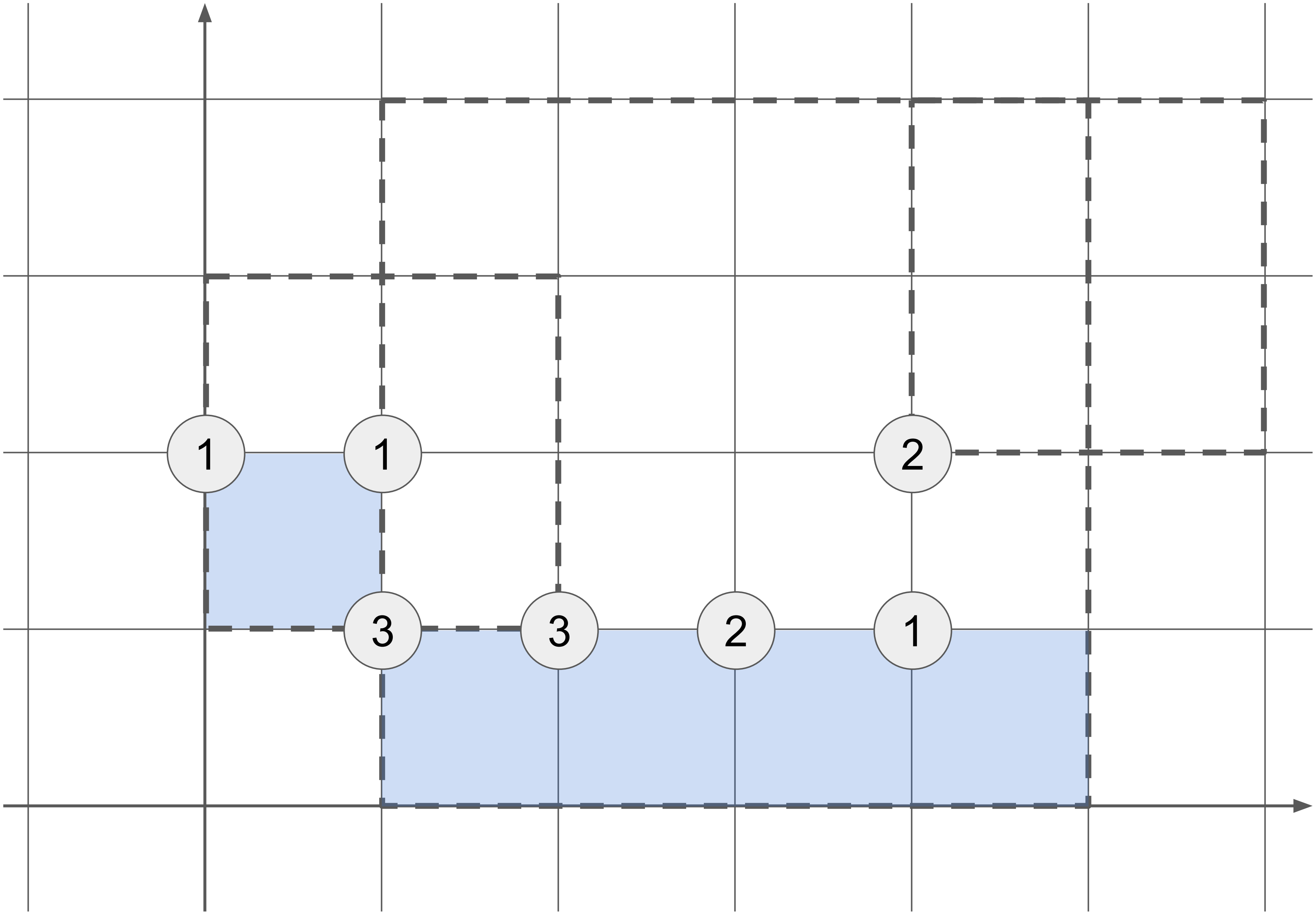

2
3
3
2
2
2

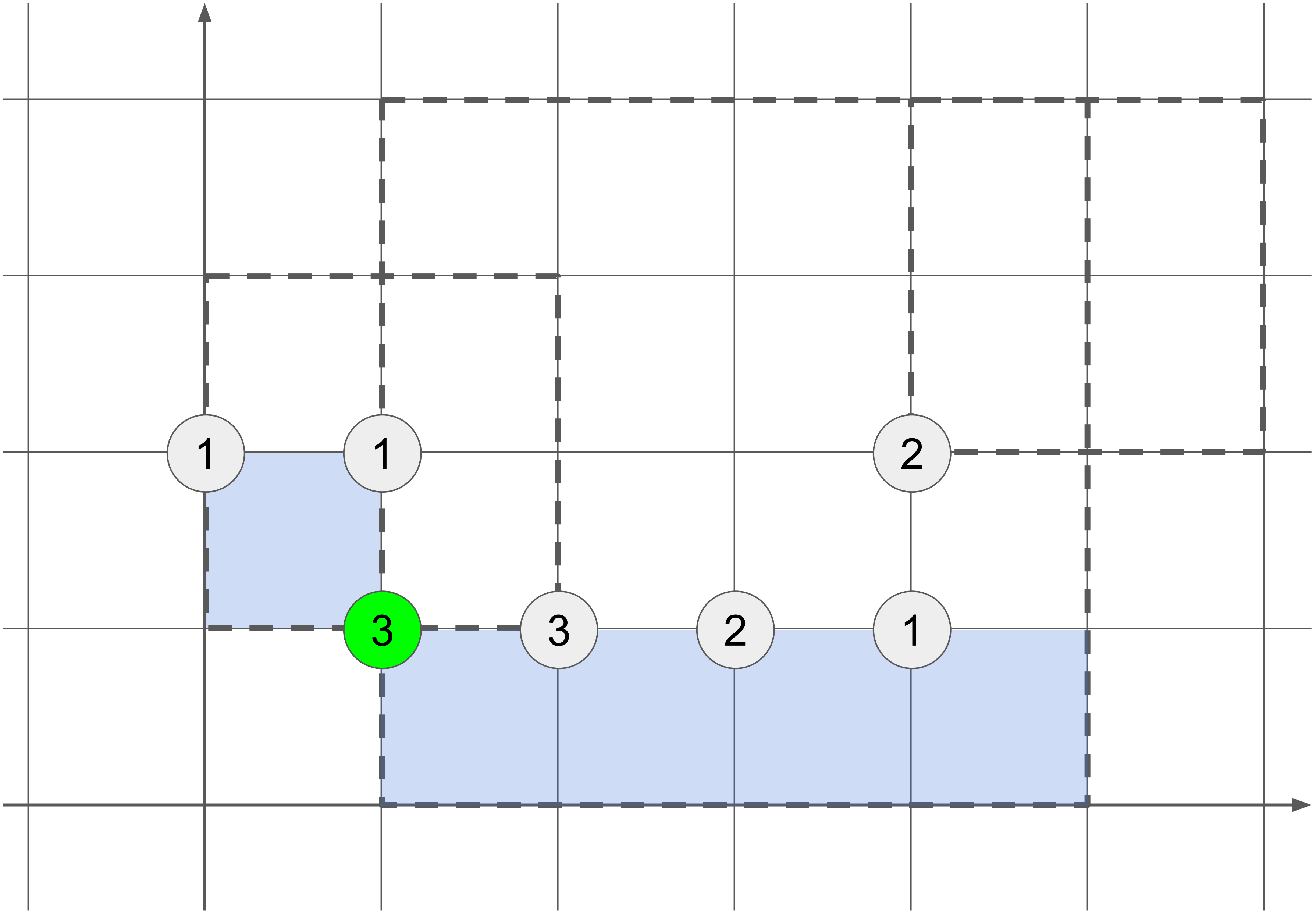

2
2
3
3
2
1
1

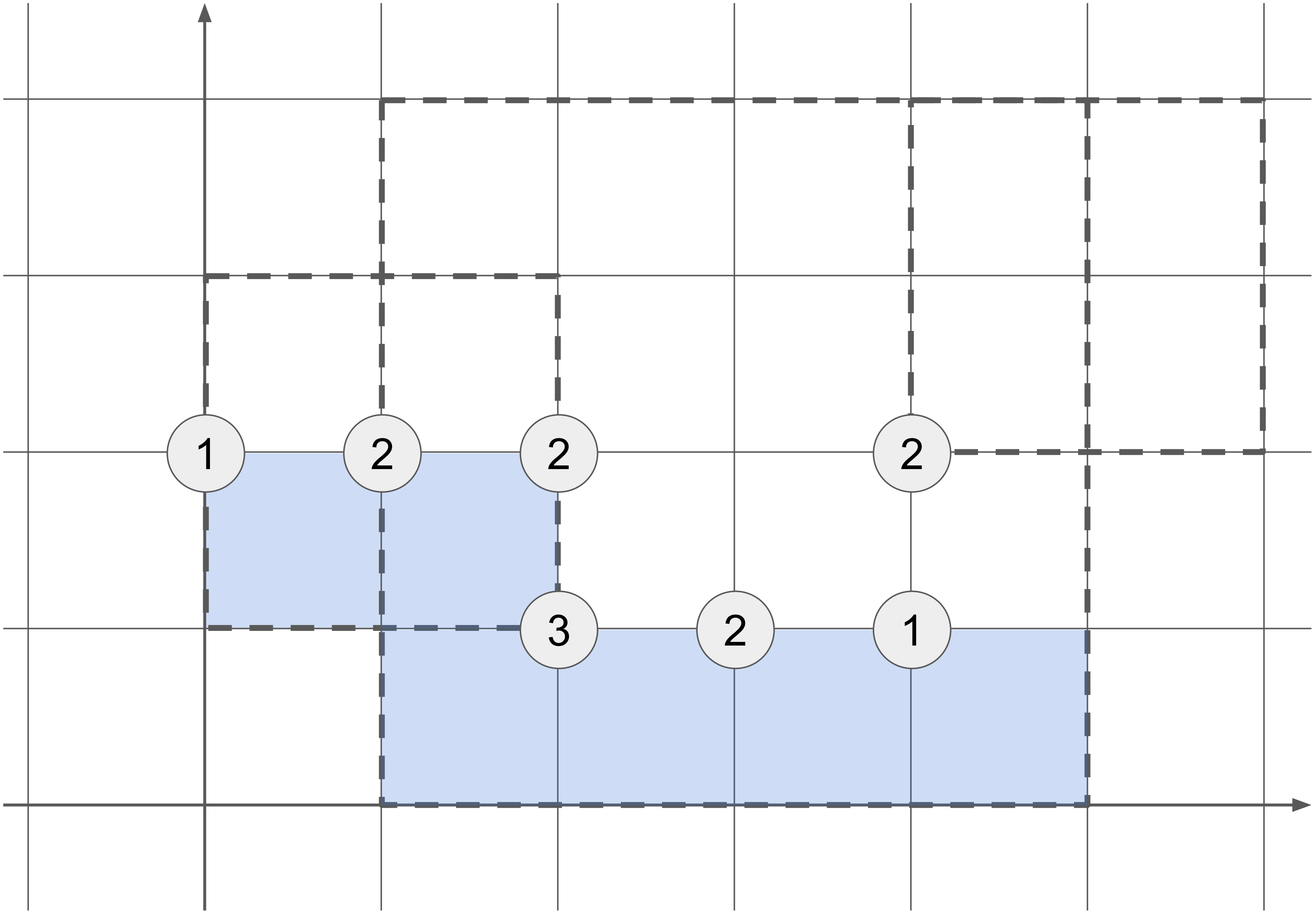

2
2
3
3
2
1
1

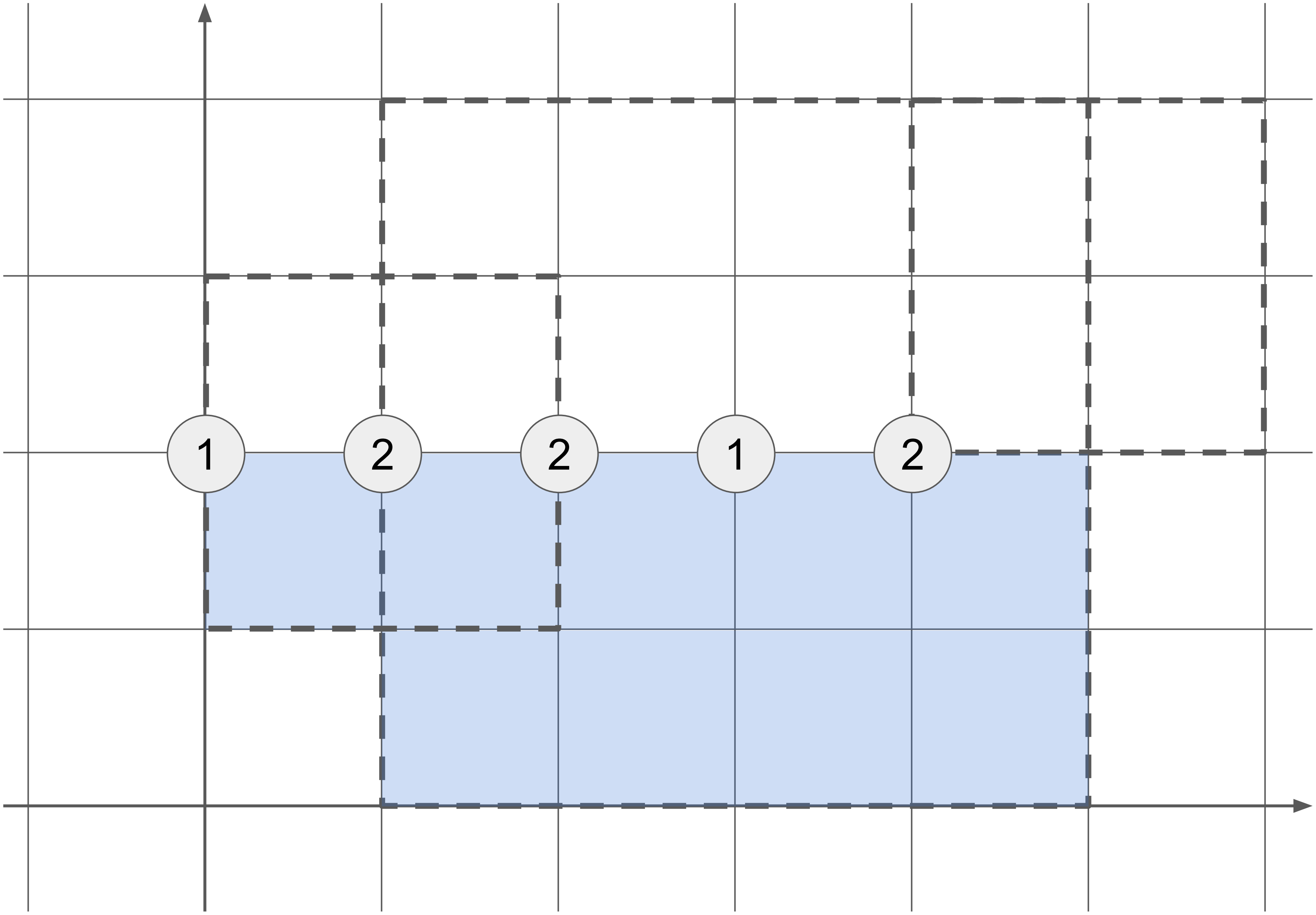

2
2
3
3
2
1

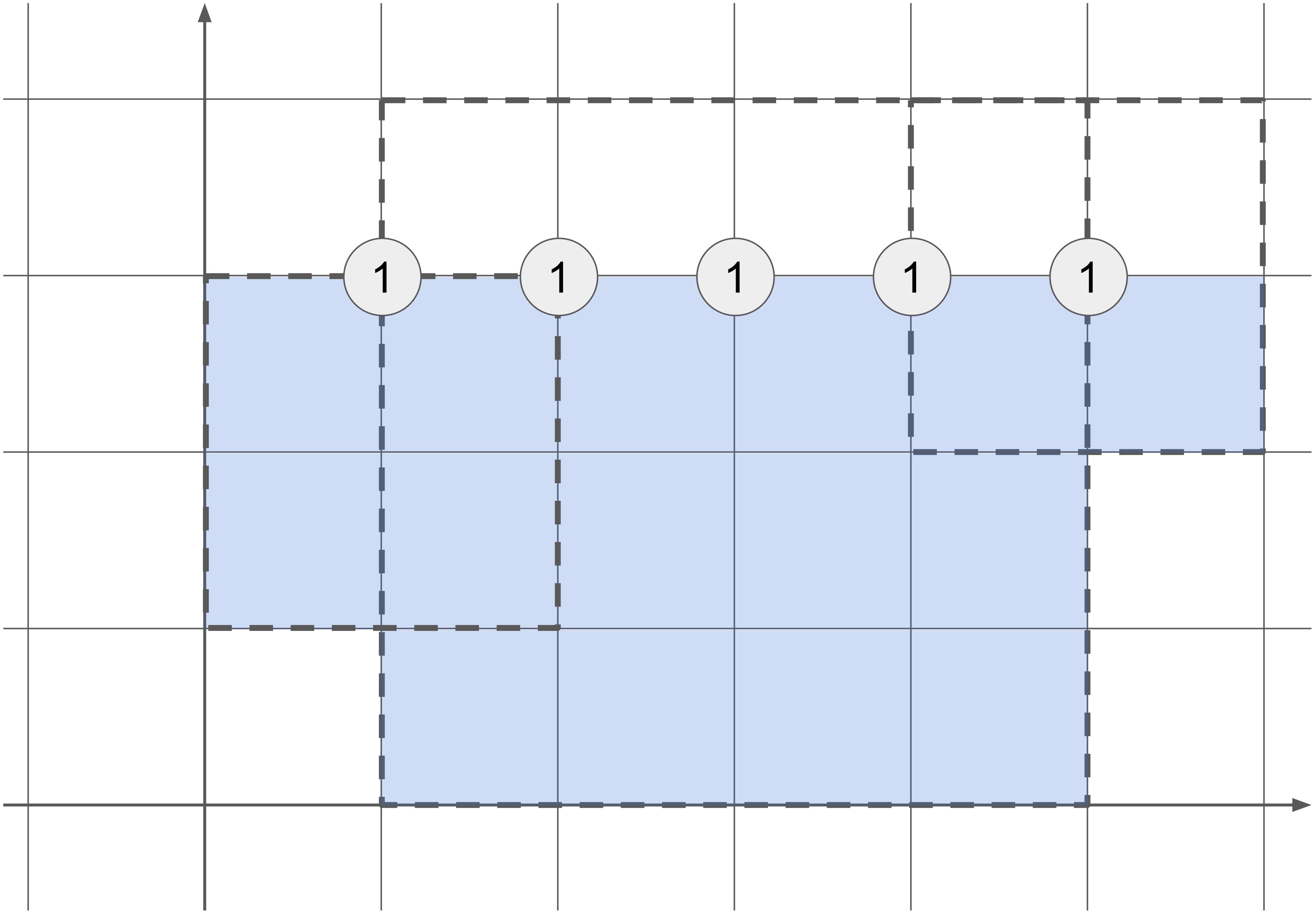

2
3
3
2
1
2

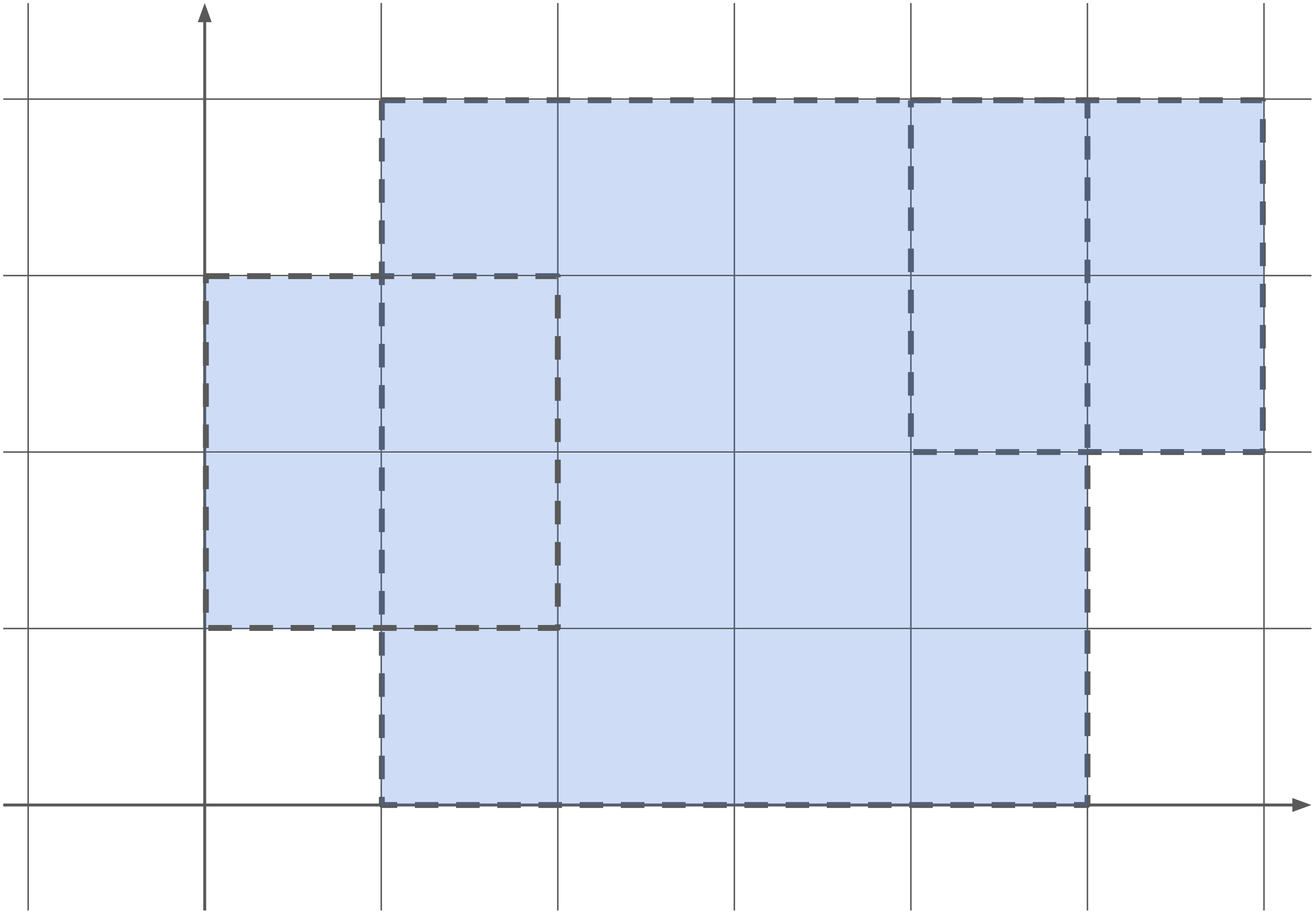

1
1
2
3
3
2
1

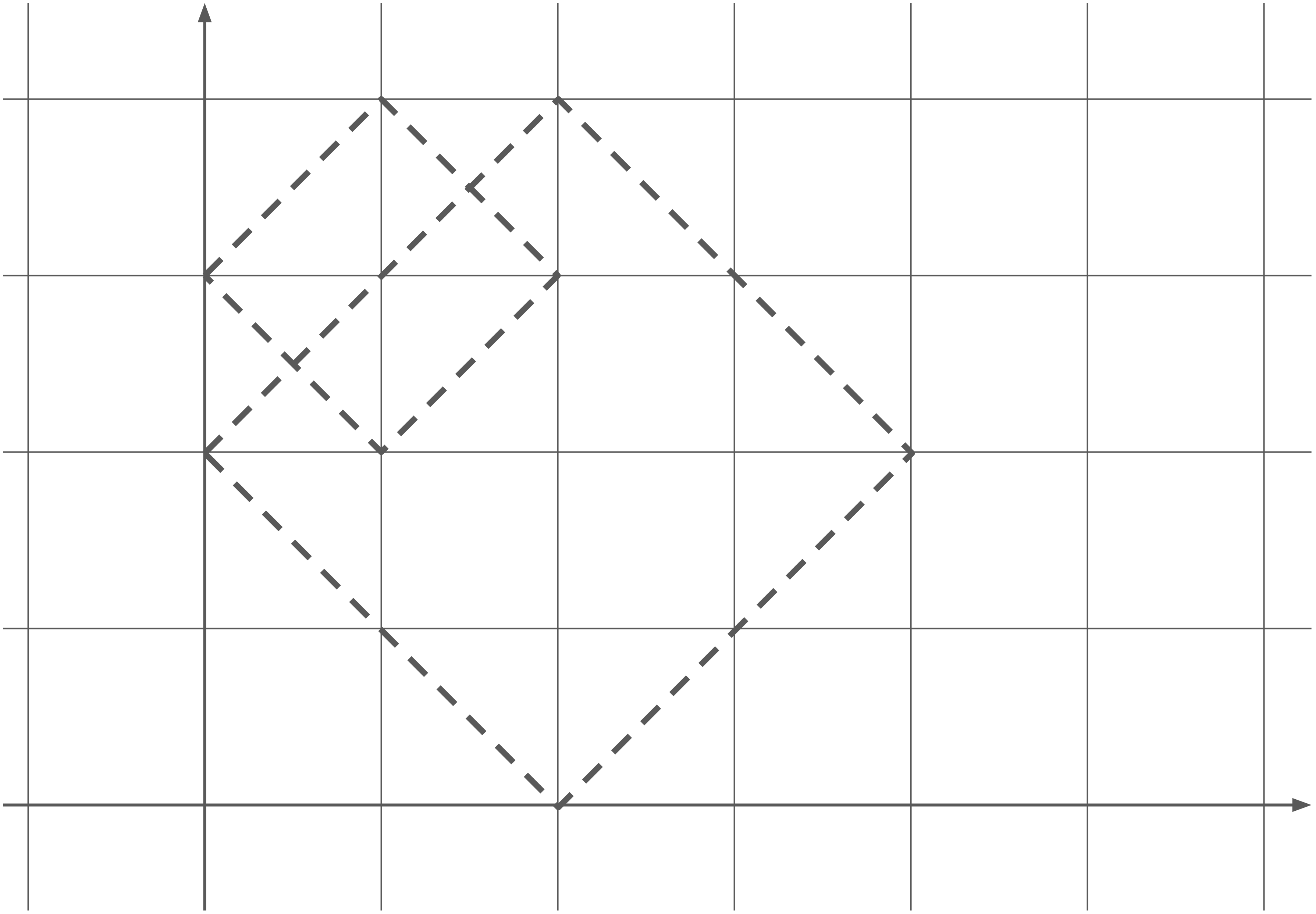

1
1
2
3
3
2
1

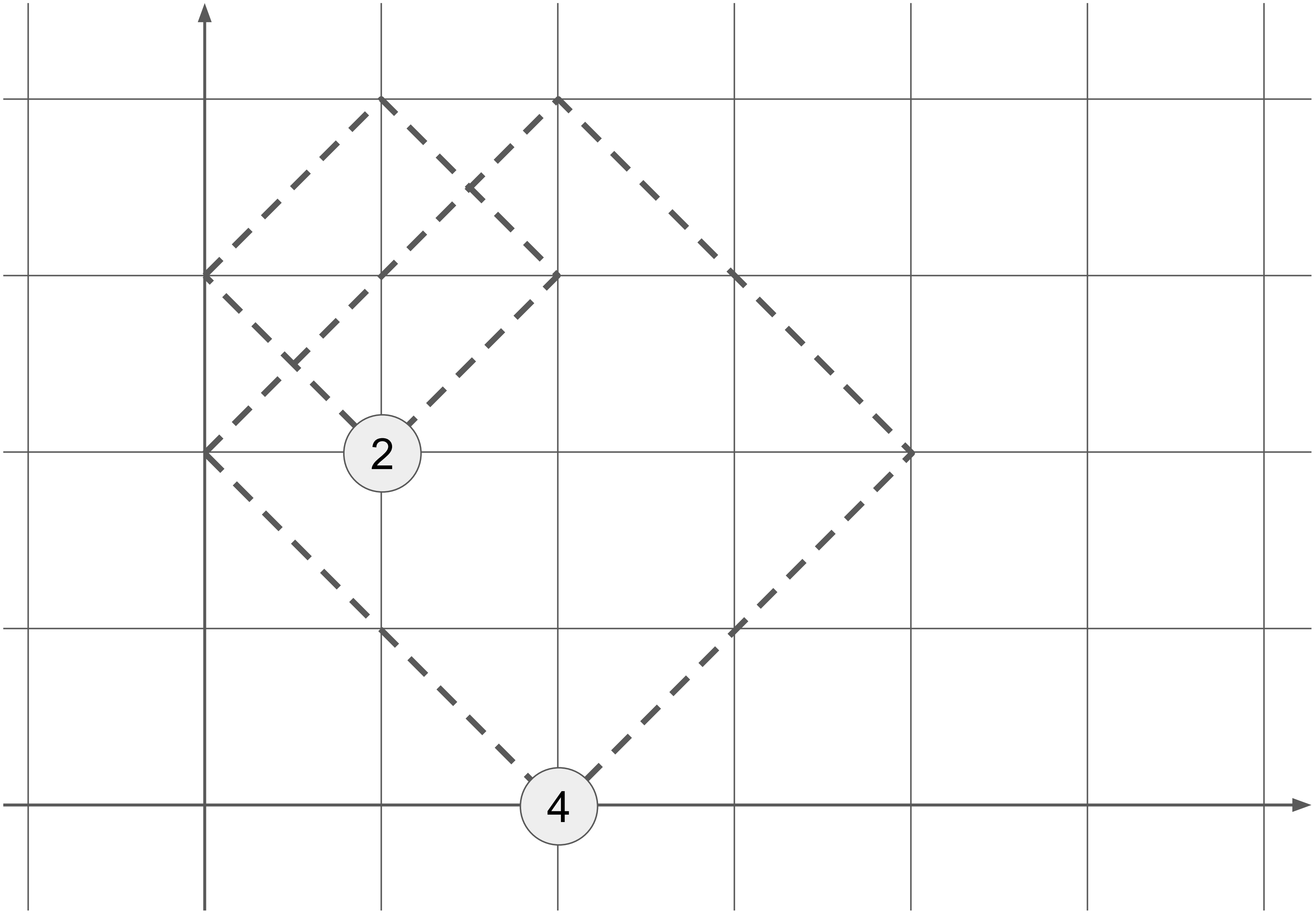

1
2
2
2
3
2
1

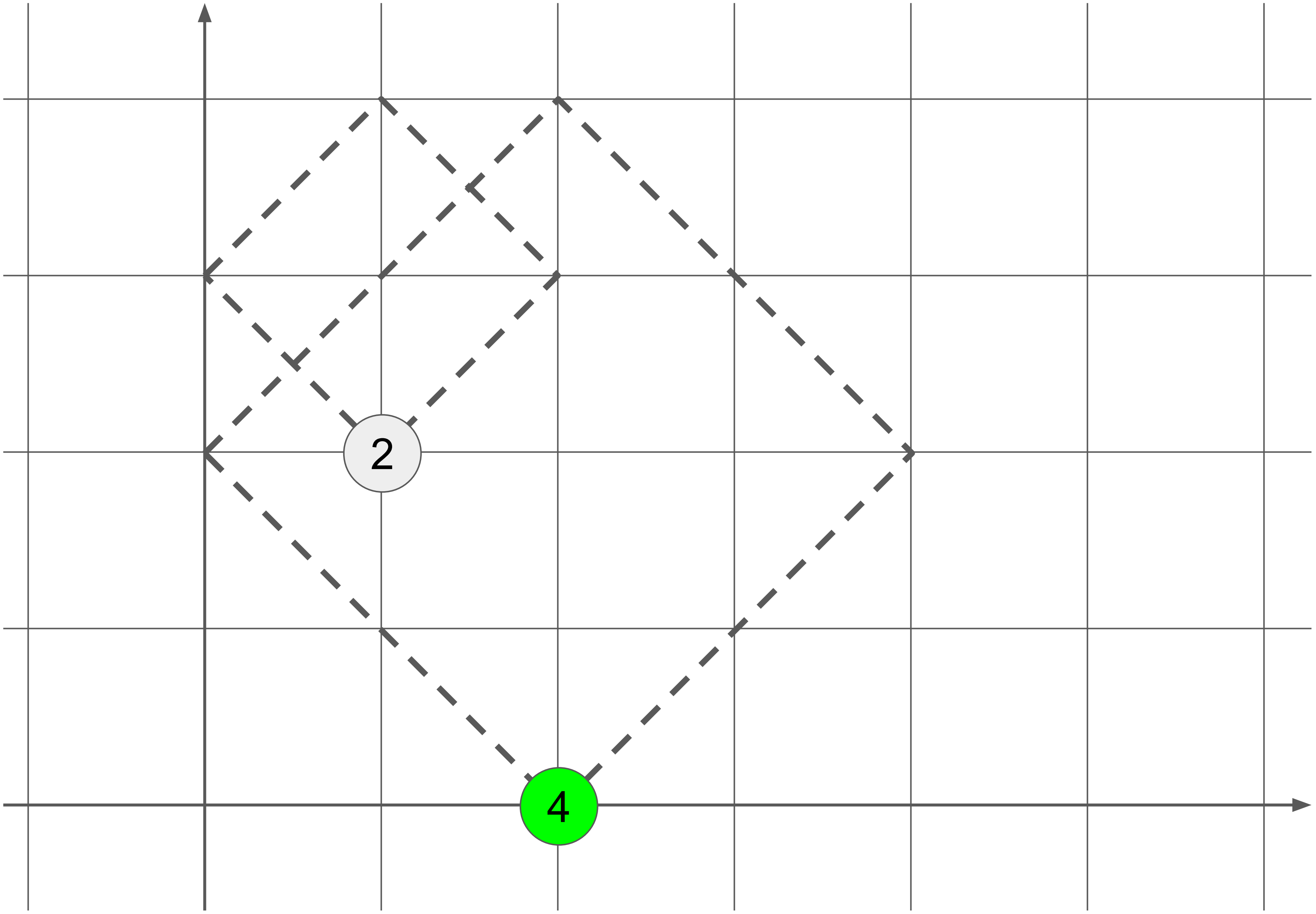

1
2
2
1
2

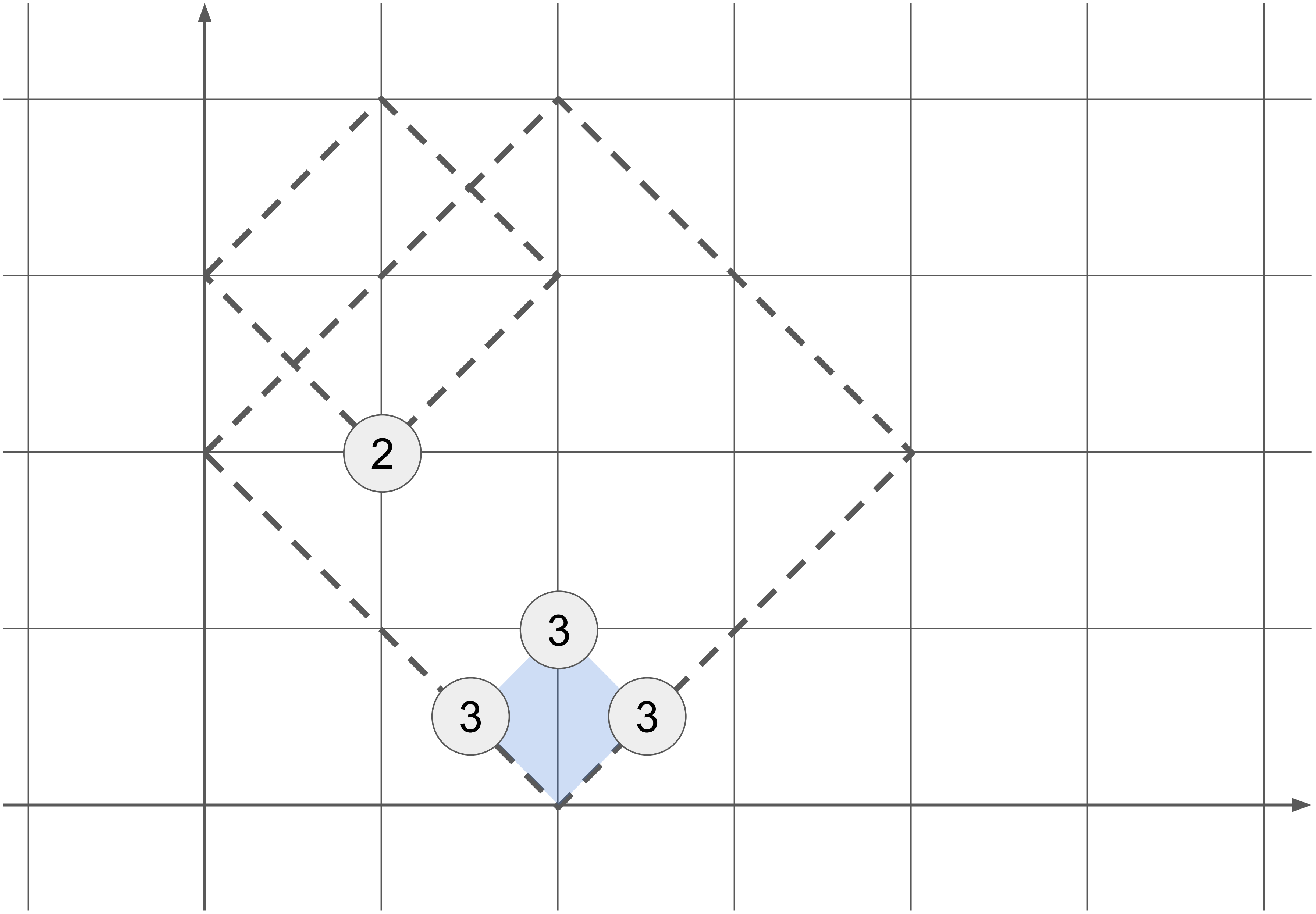

1
1
1
1
1

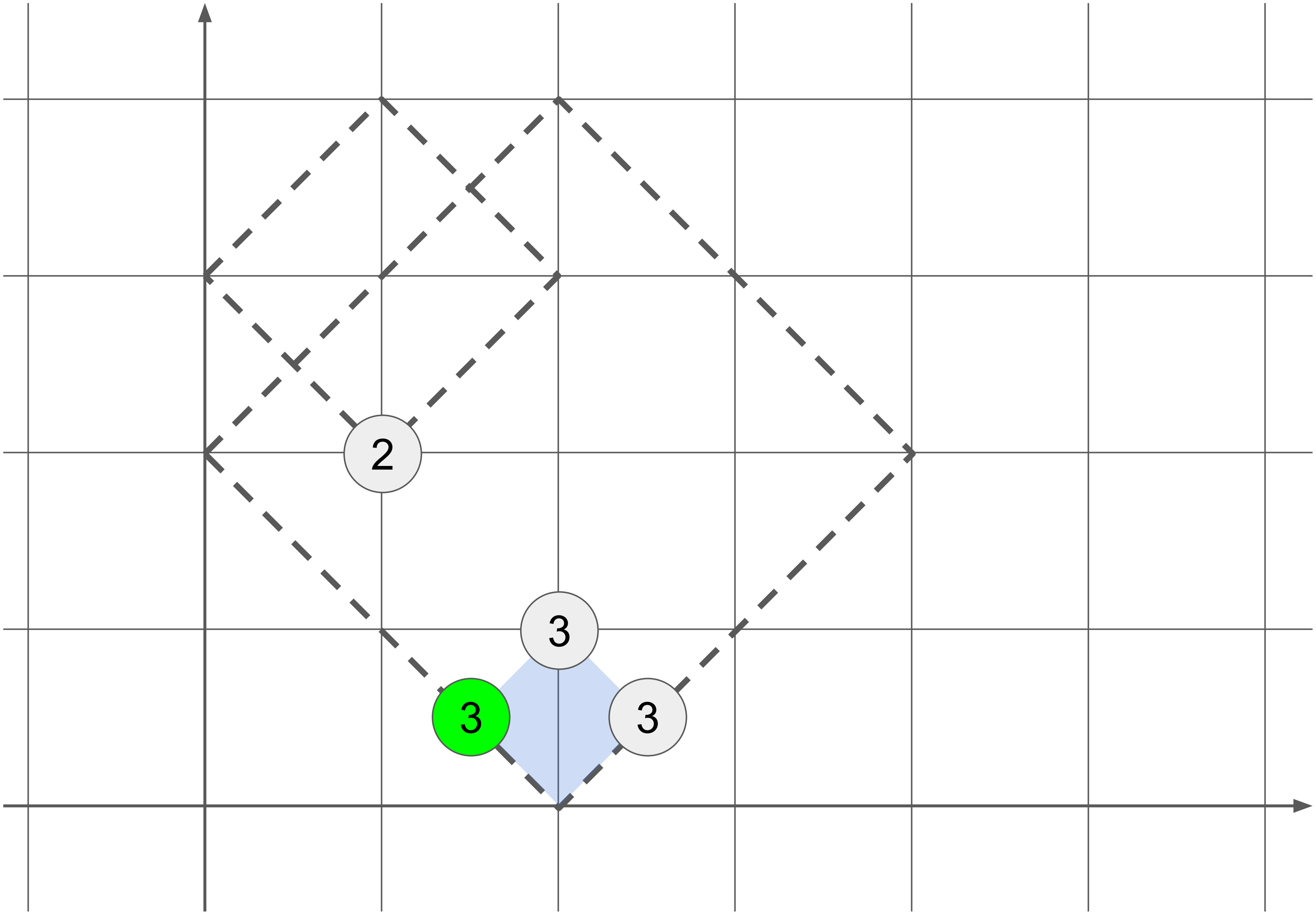


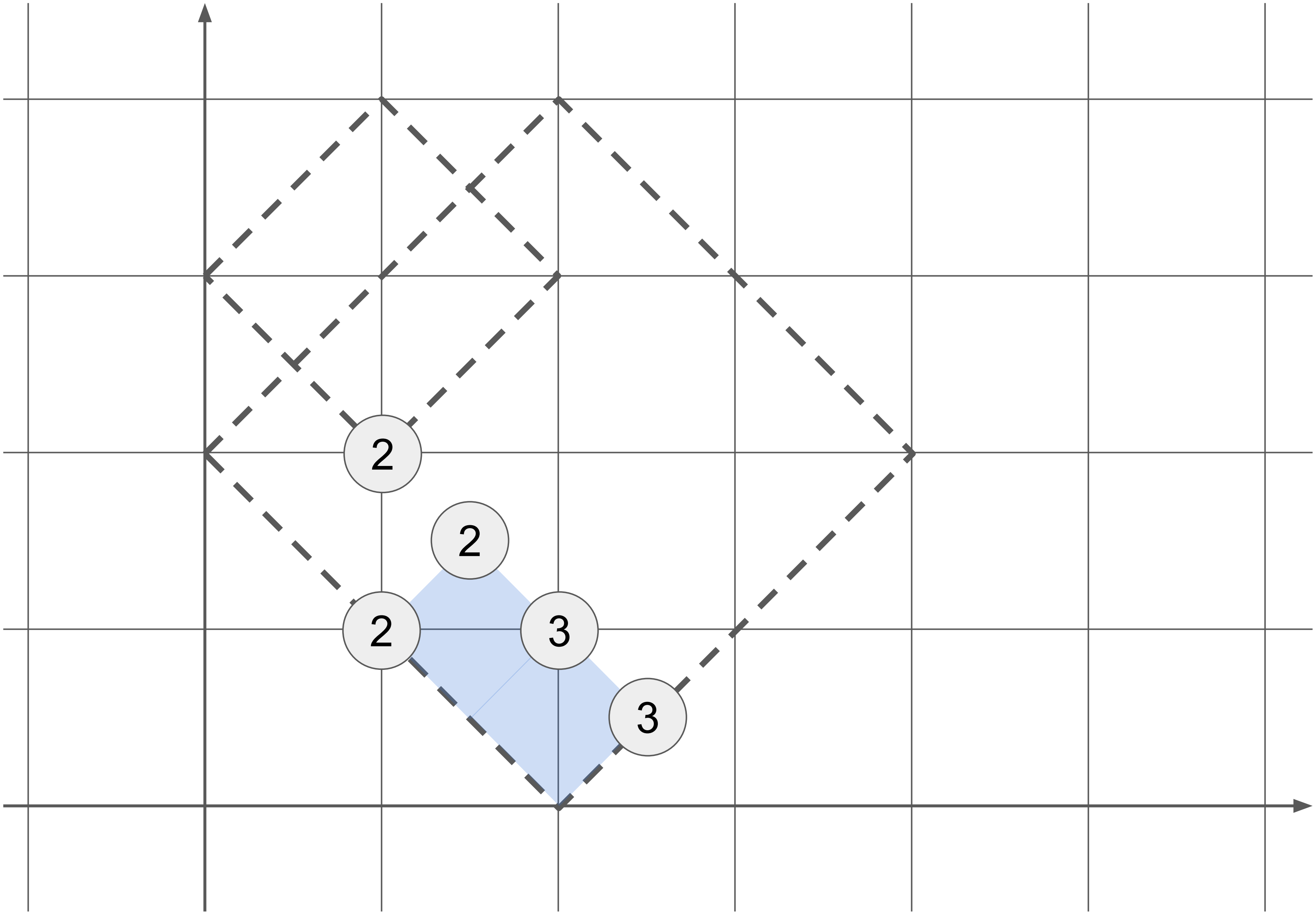


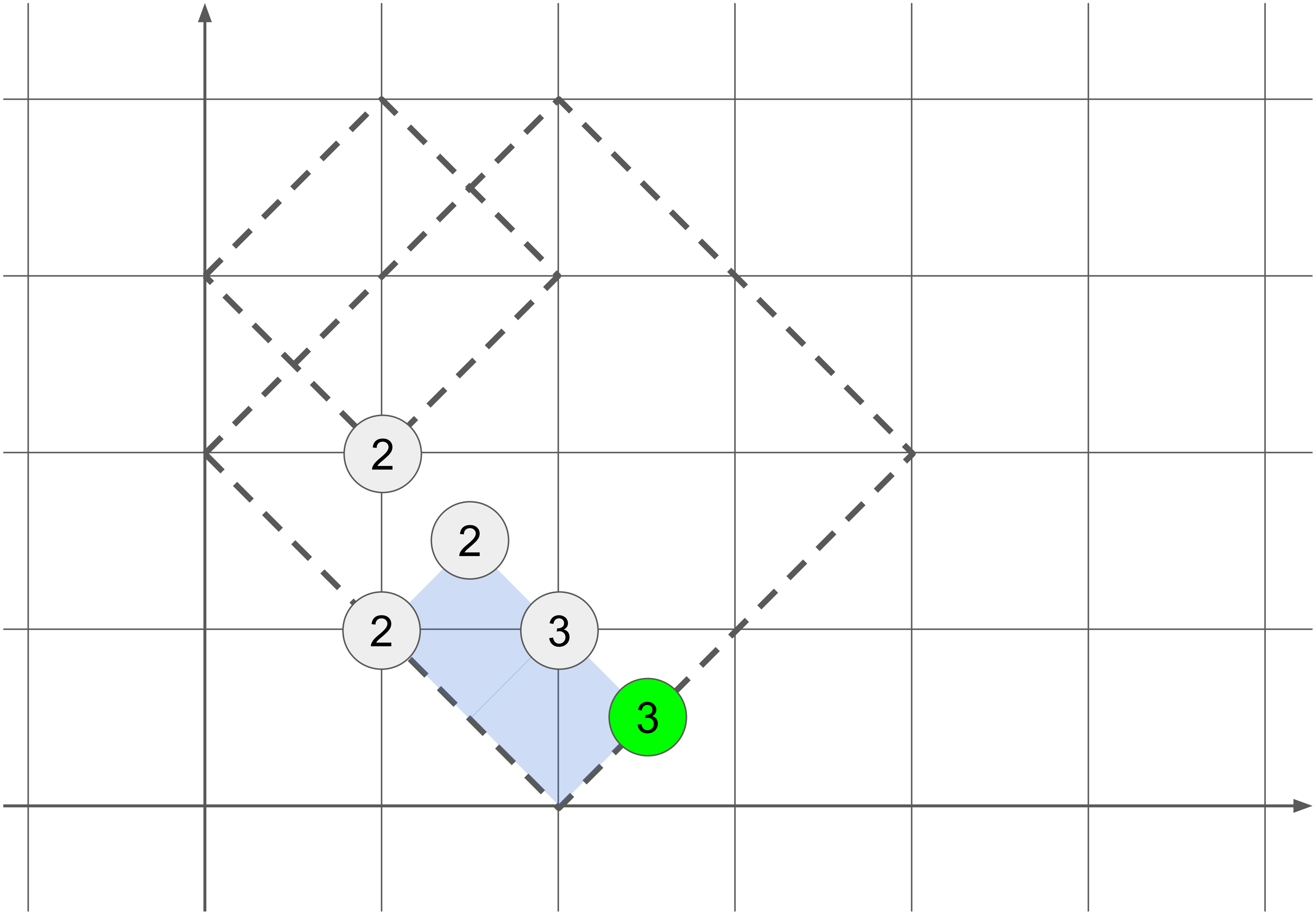

2
4

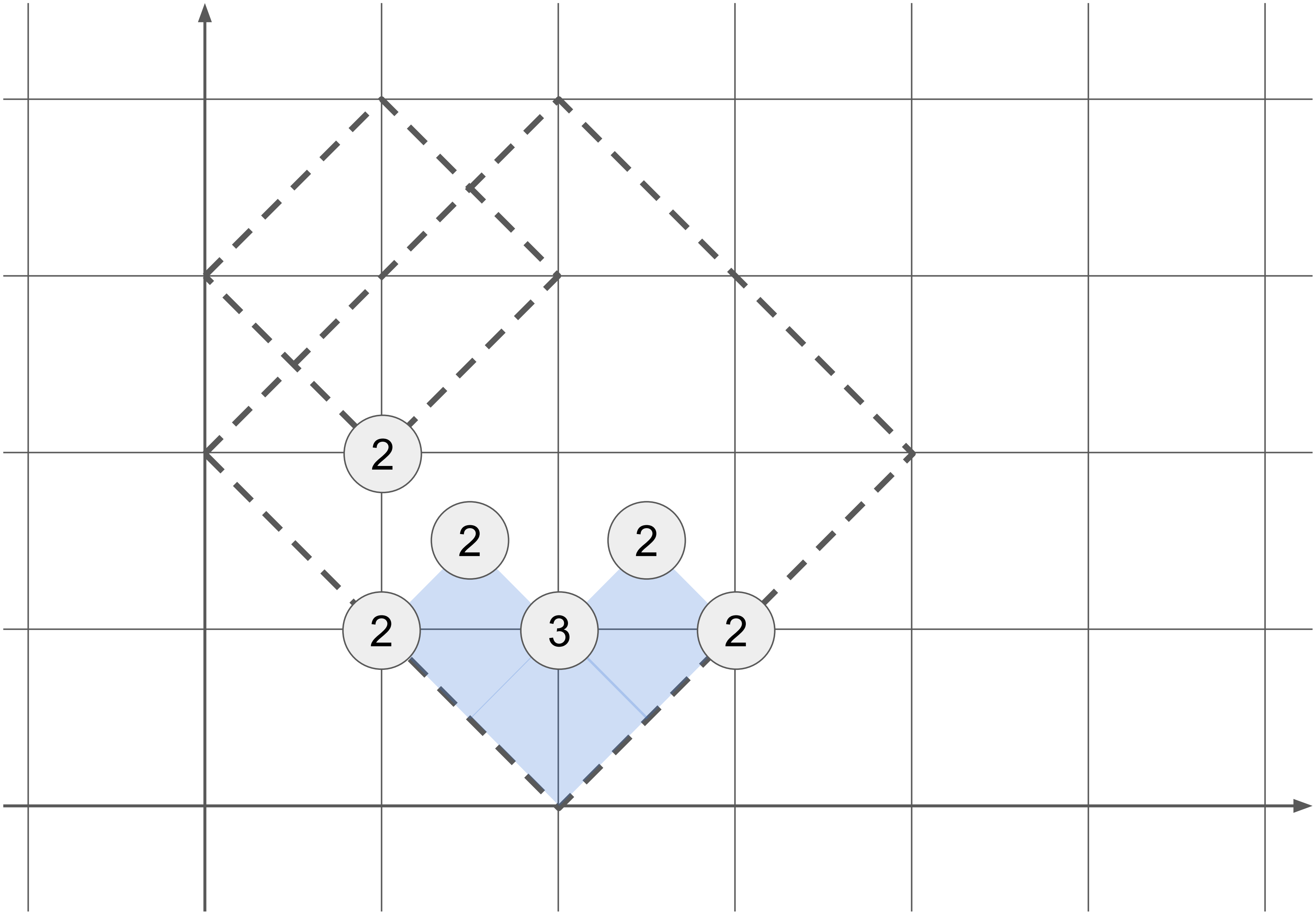

2
4

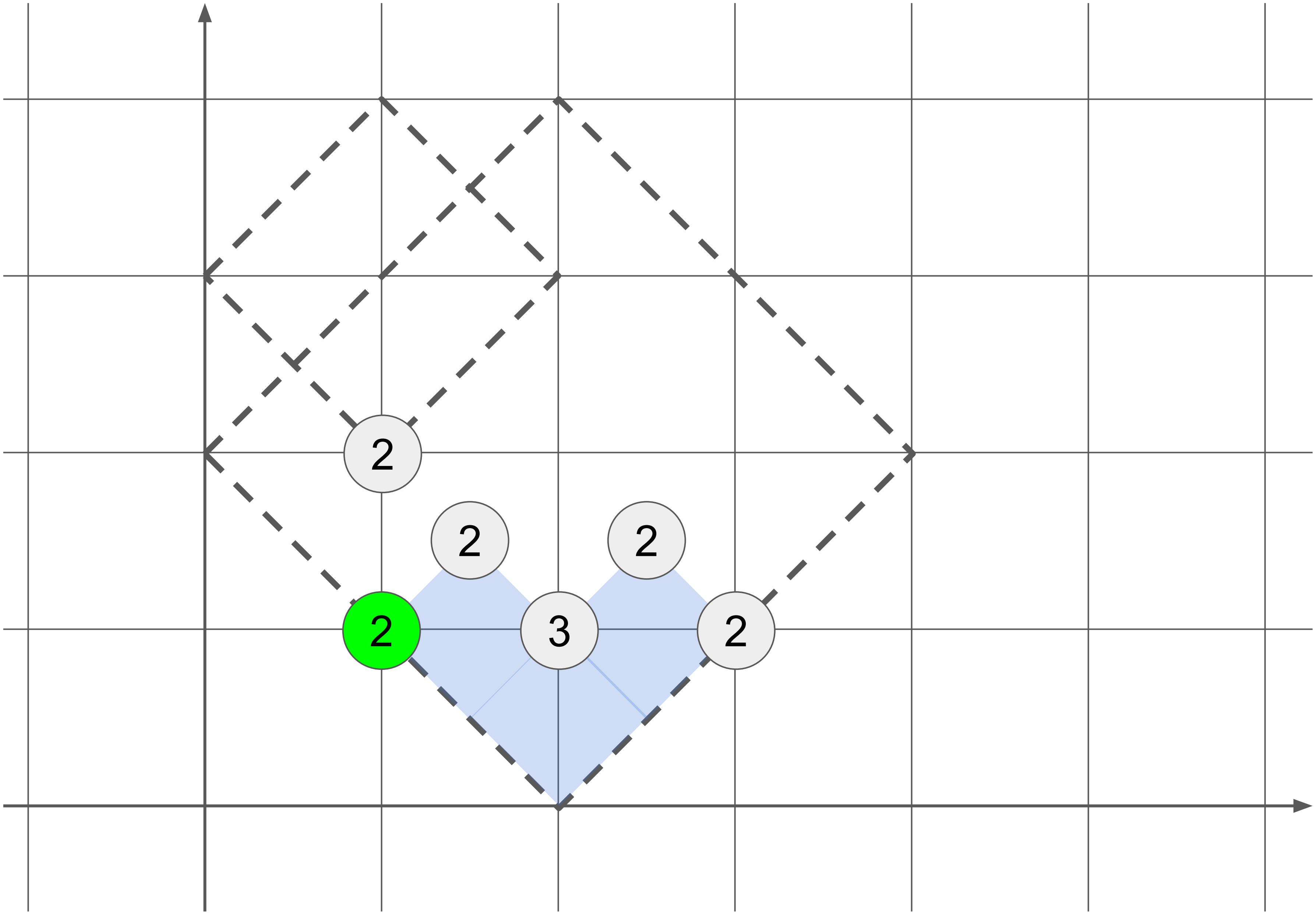

2
3
3
3

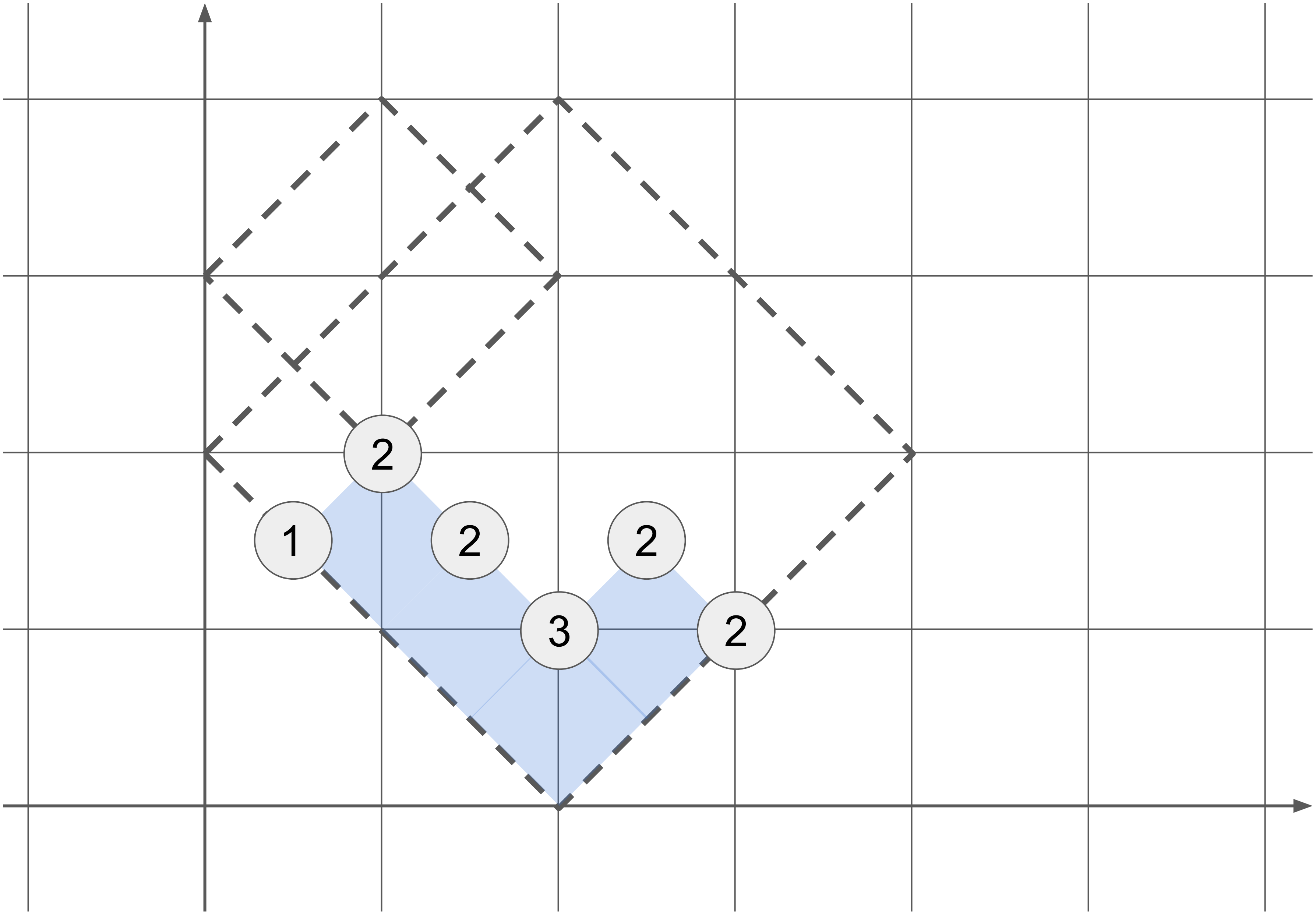

2
3
3
3

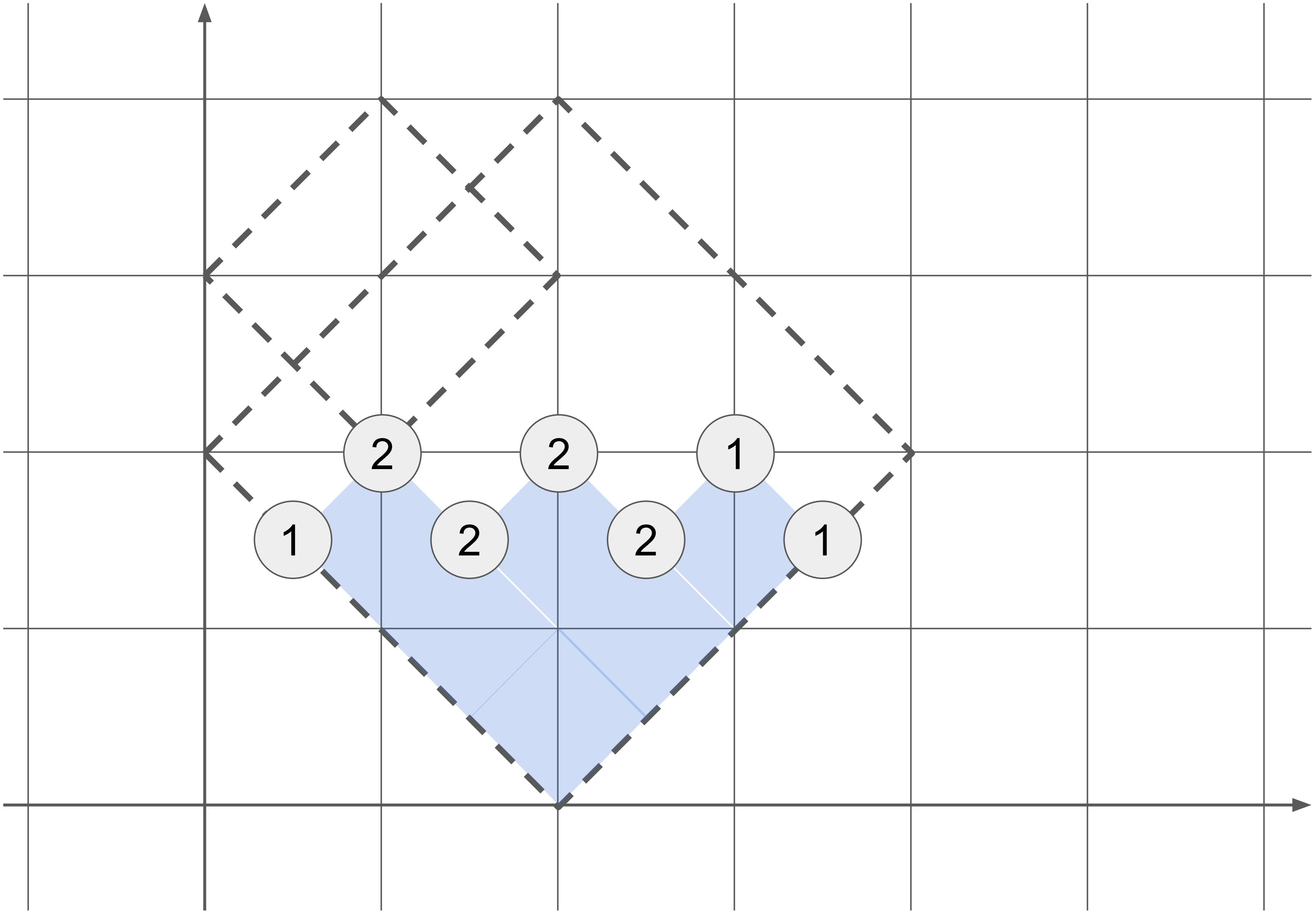

2
2
2
3
3

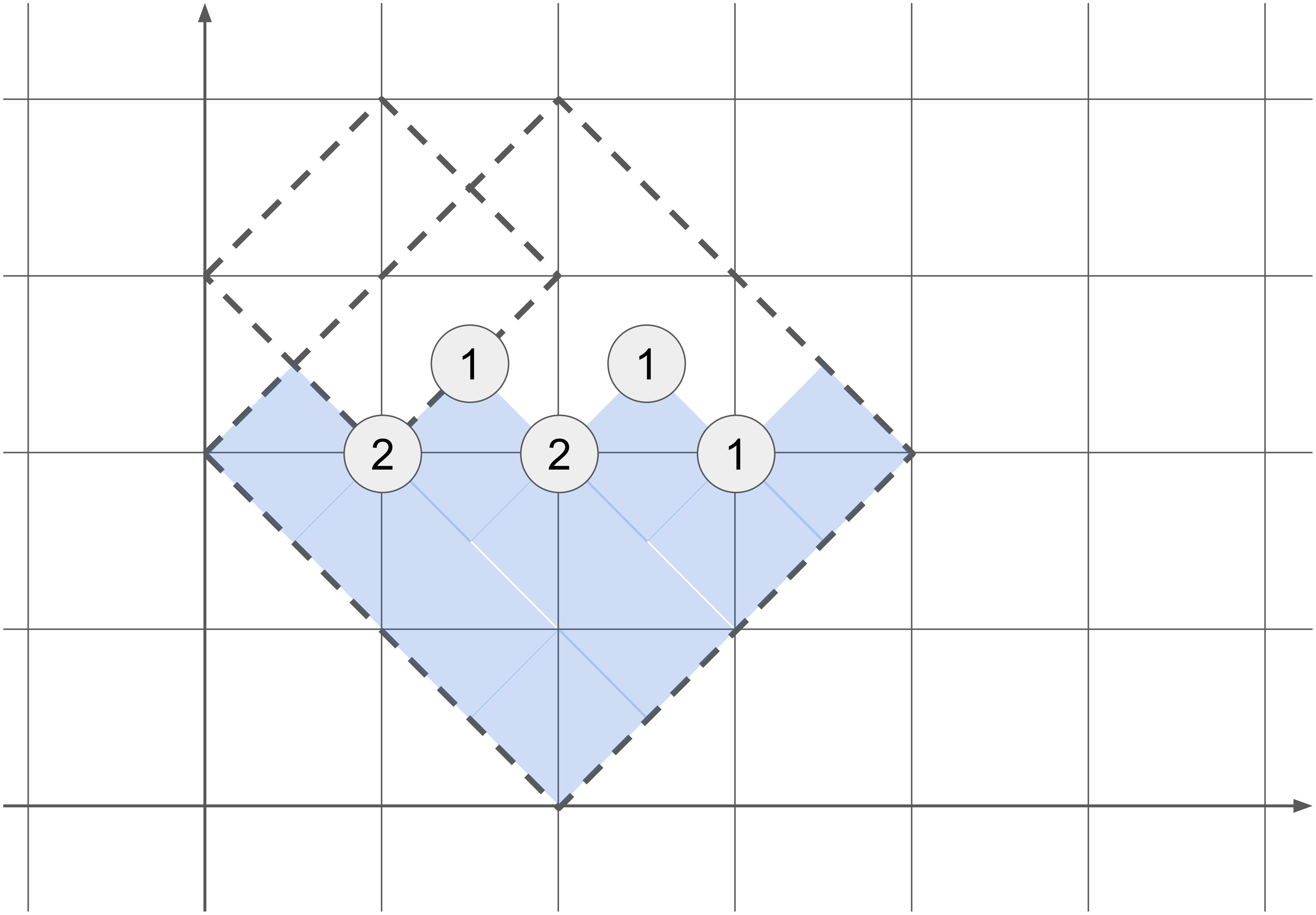

2
2
2
3
3

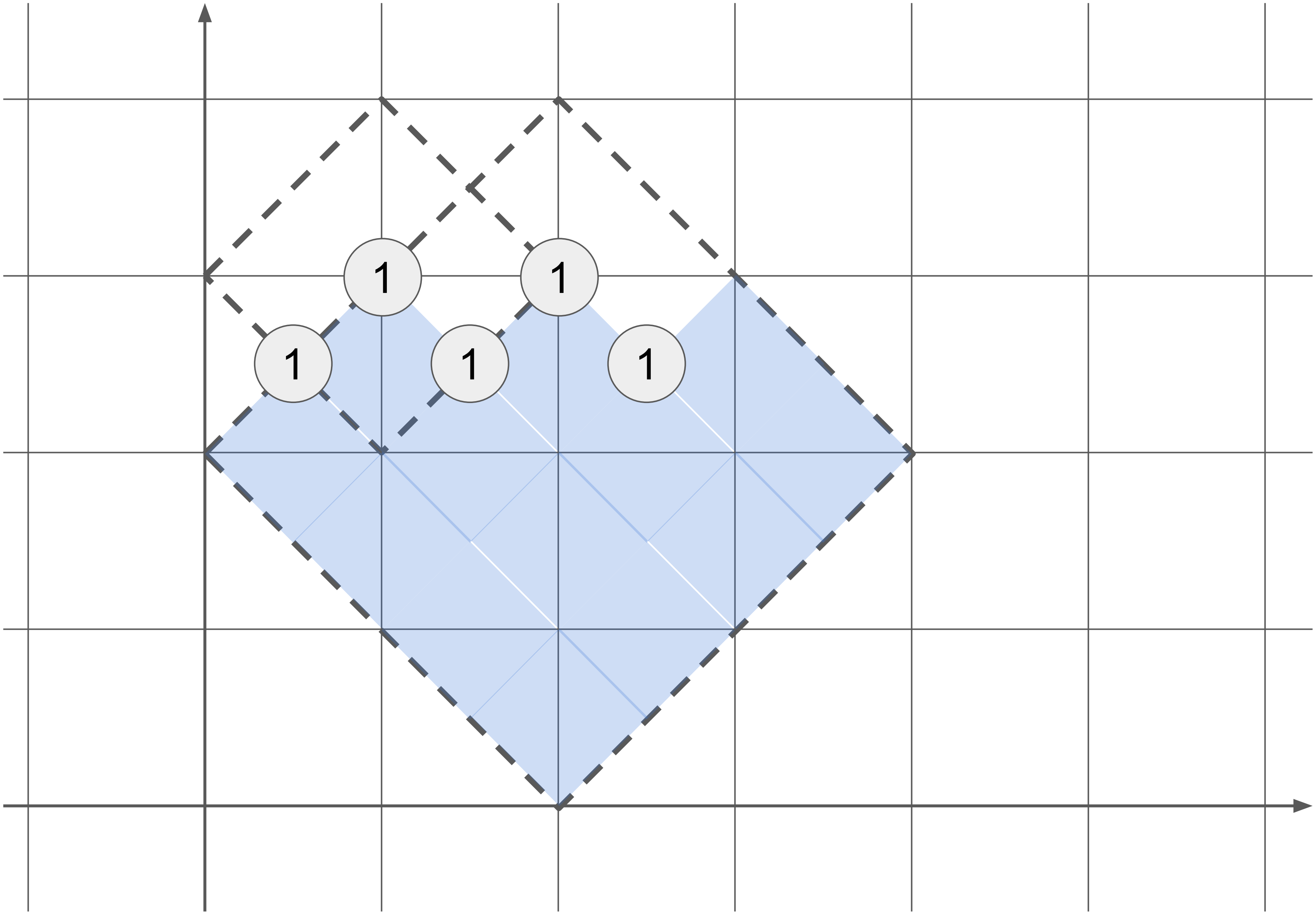

2
2
2
2
3
2

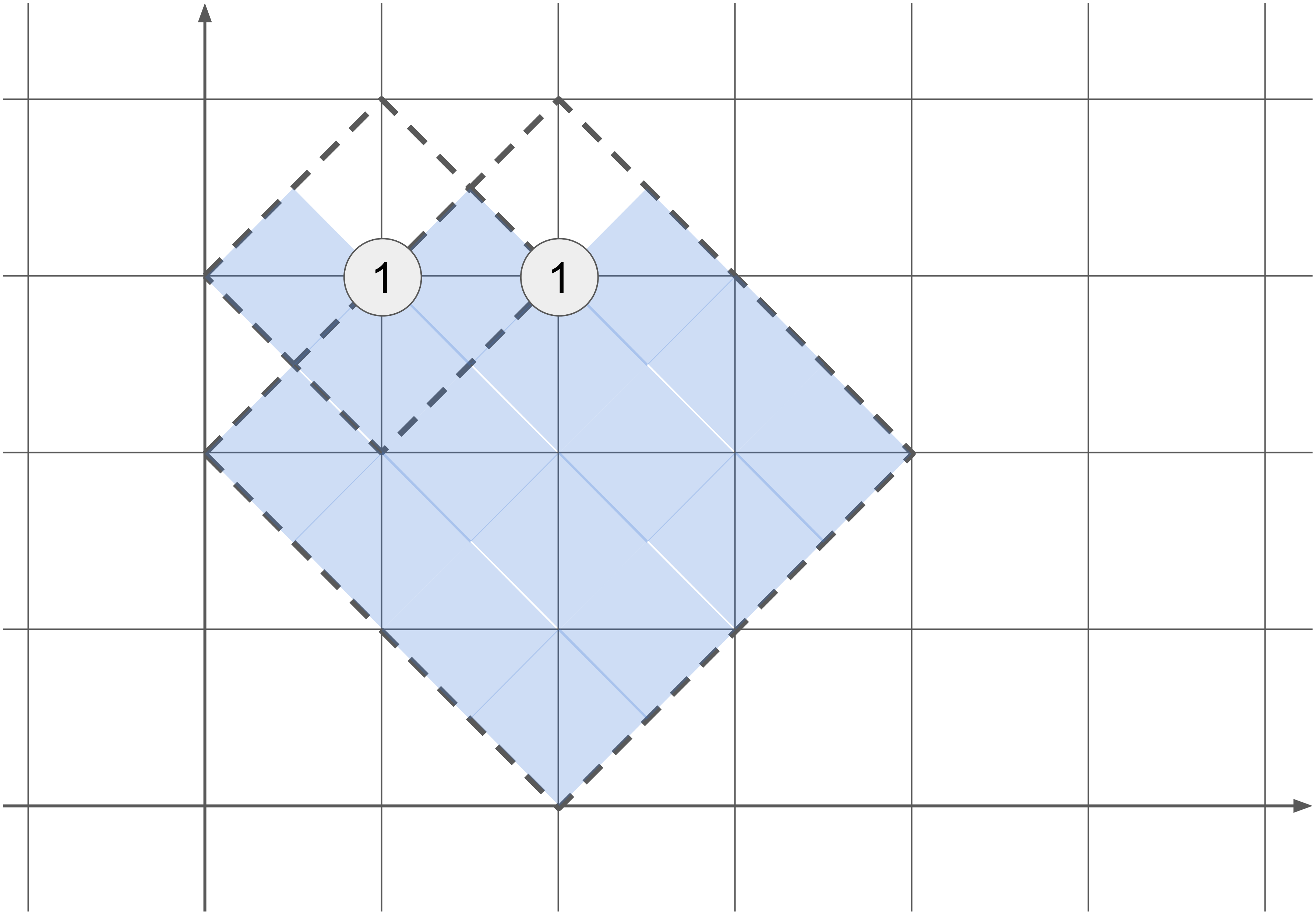

2
2
2
2
3
2

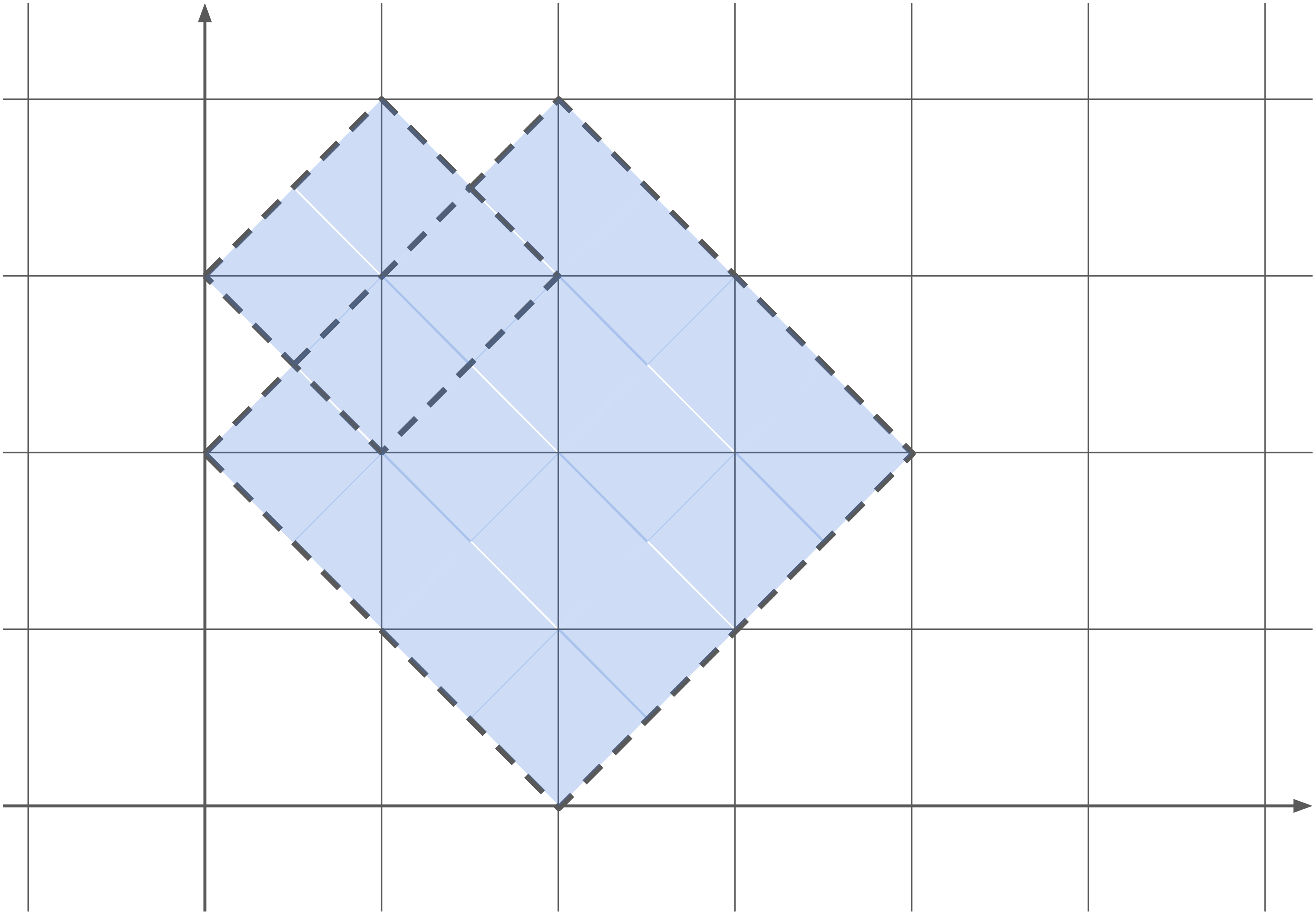

2
1
2
2
3
2

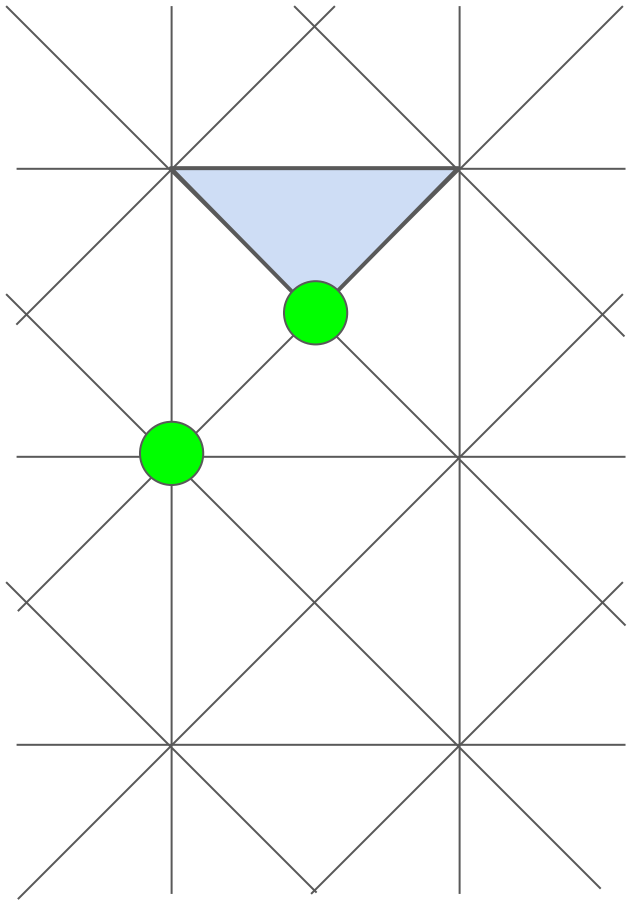

2
2
1
1
2
2
1


1
1
2
2
1

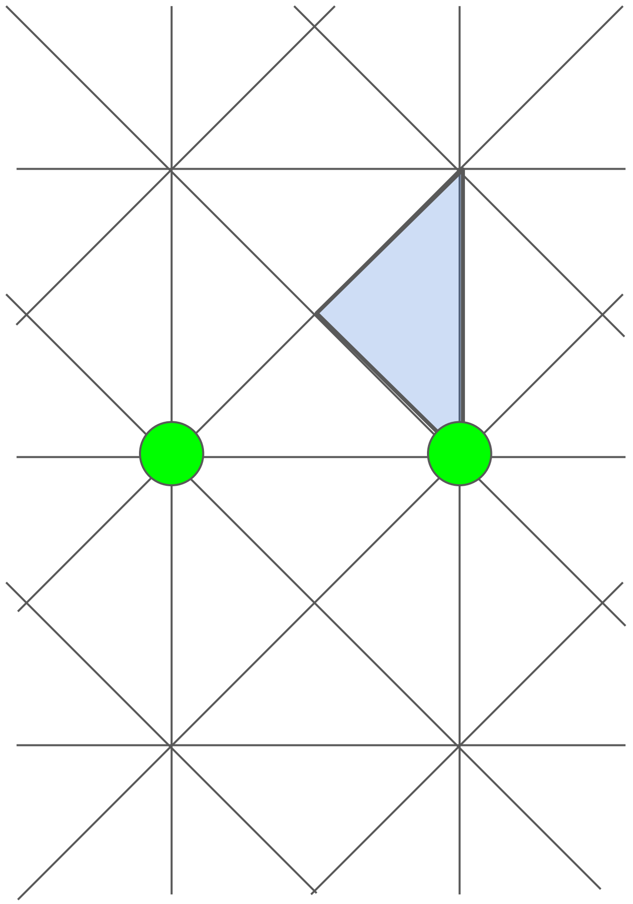

1
1
1
1
1


1
1

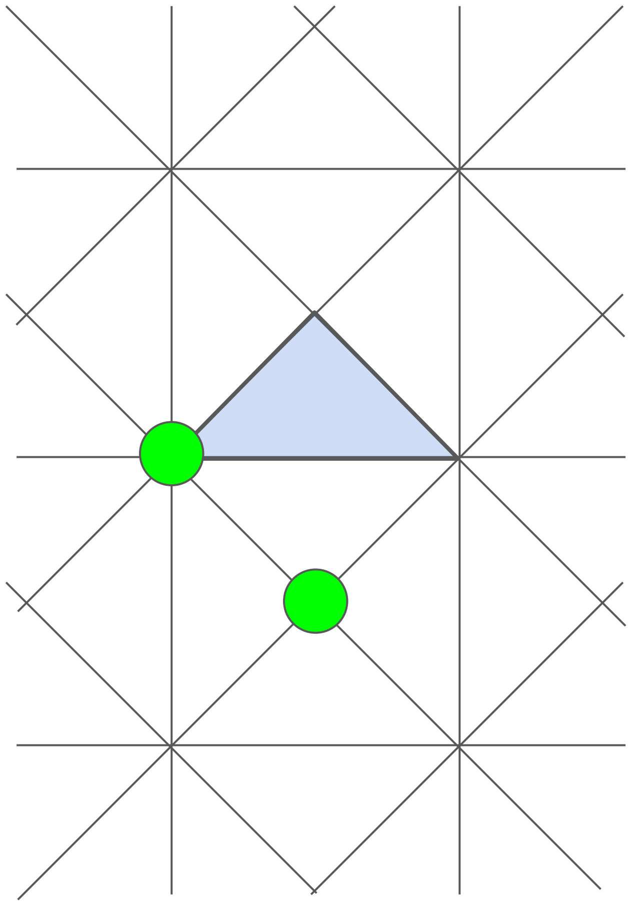




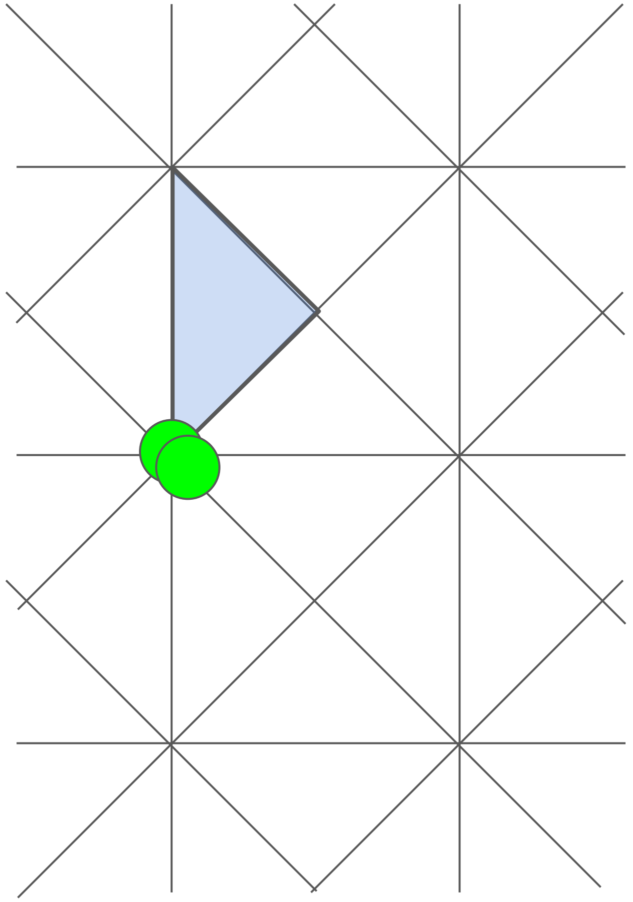
For each triangle we can deduce whether it was painted or not.
We need to check whether we ever had a type A bucket of paint in the lower left corner of the unit square or a type B bucket in the right position (depending on the triangle type).



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 · W)
Memory complexity: O(H · 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.

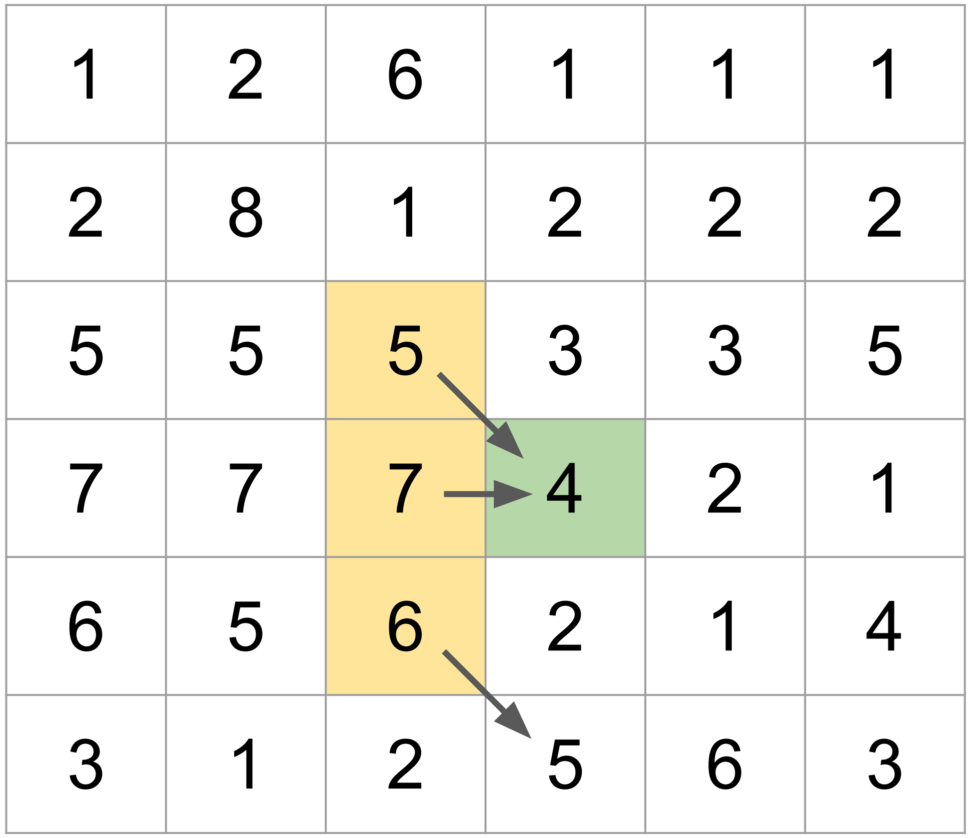
As soon as we reach the first column we can start making jumps of size C that stay in the first column until there are less C steps to make.

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.

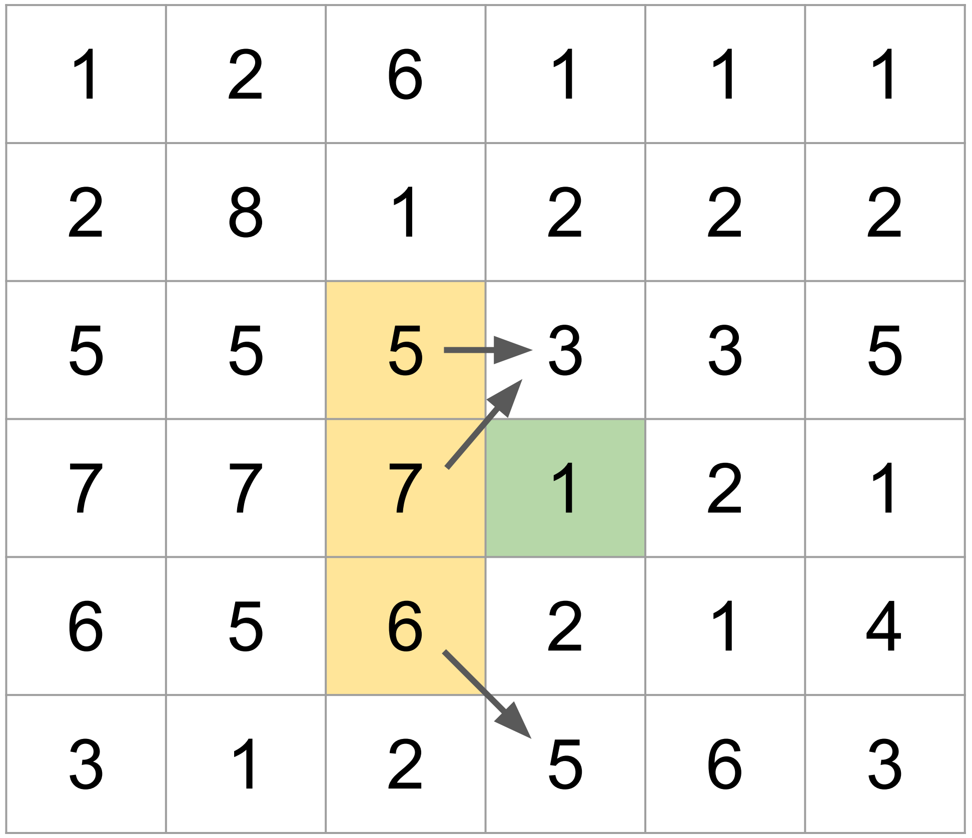
1	2	6	1	1	1
2	8	1	2	2	2
5	5	5	3	3	5
7	7	7	4	2	1
6	5	6	2	1	4
3	1	2	5	6	3

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



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

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

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

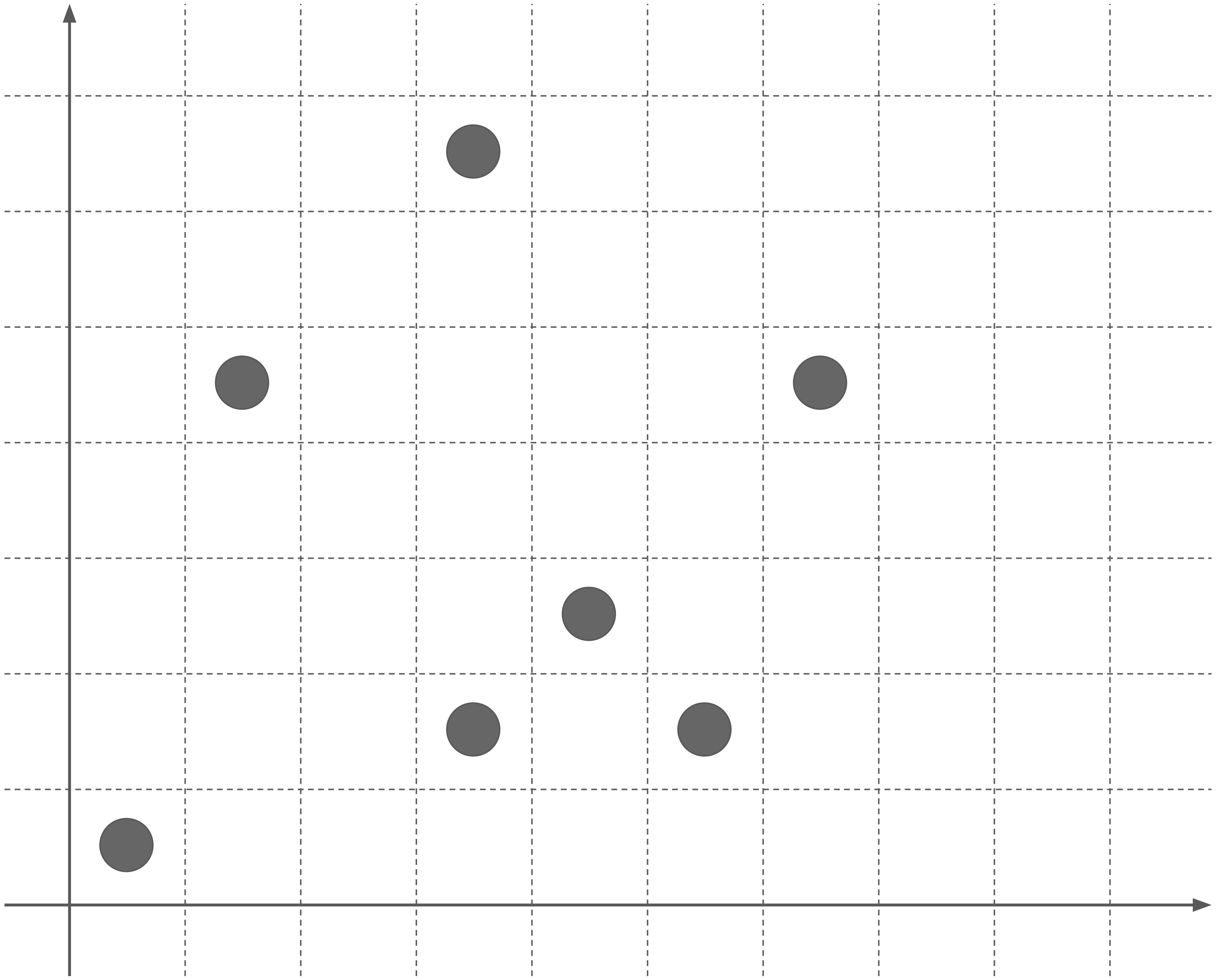
Interval bounds may only move by ±1 between neighbouring columns, so we can maintain the affected interval in O(1) per column as we backtrack. Overall update complexity is O(C).

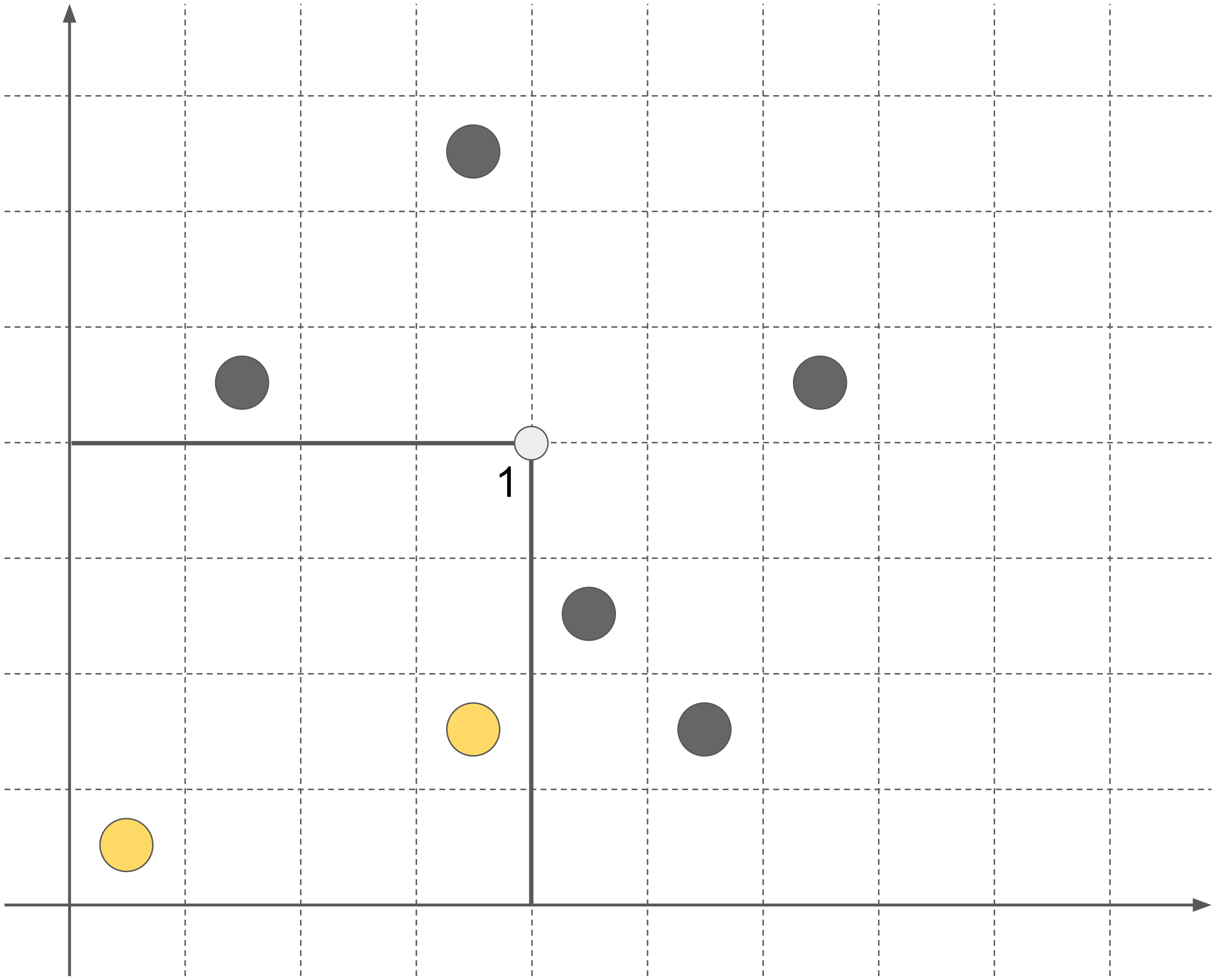
Problem B
Buffalo Barricades

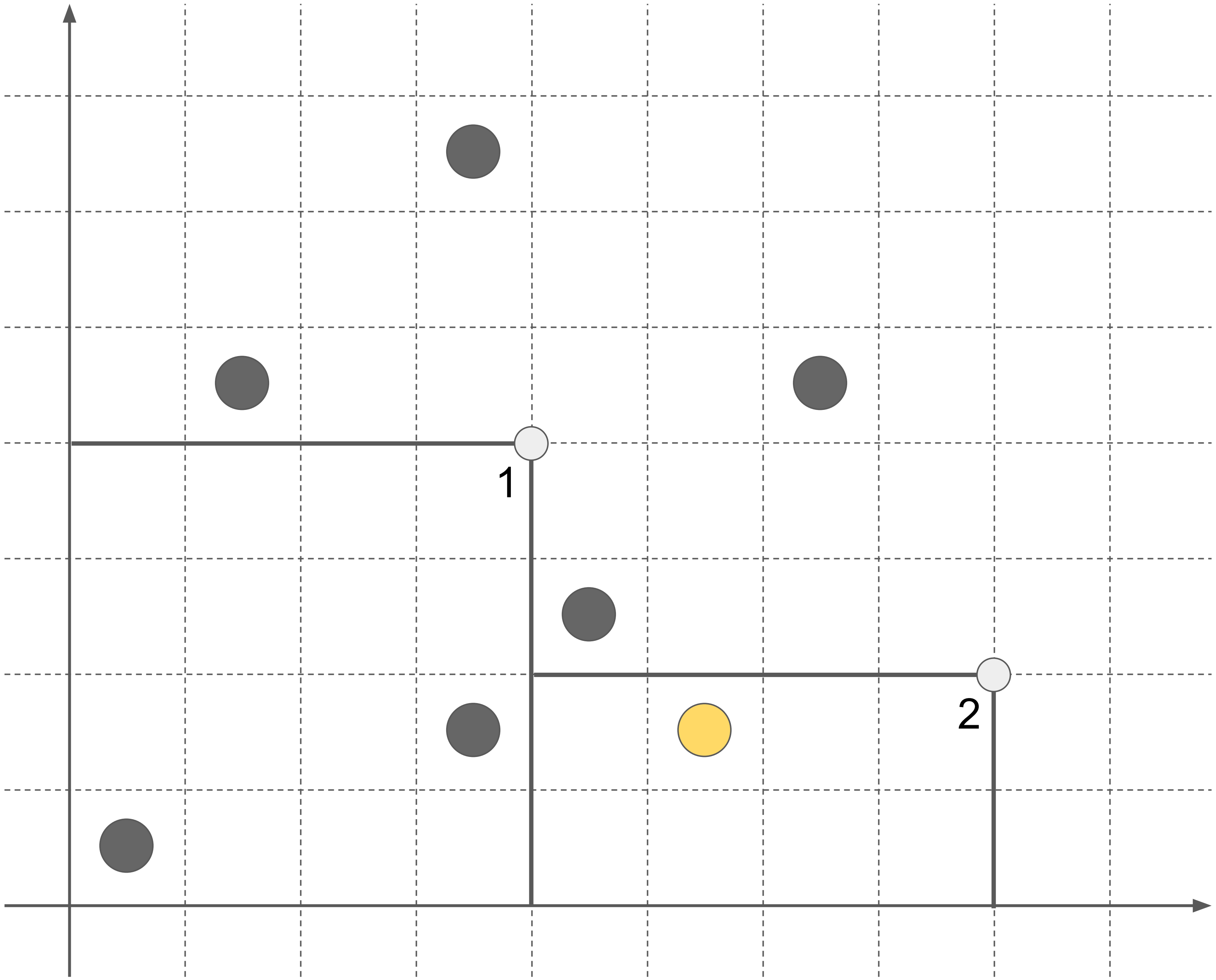
Submits: 17
Accepted: at least 1

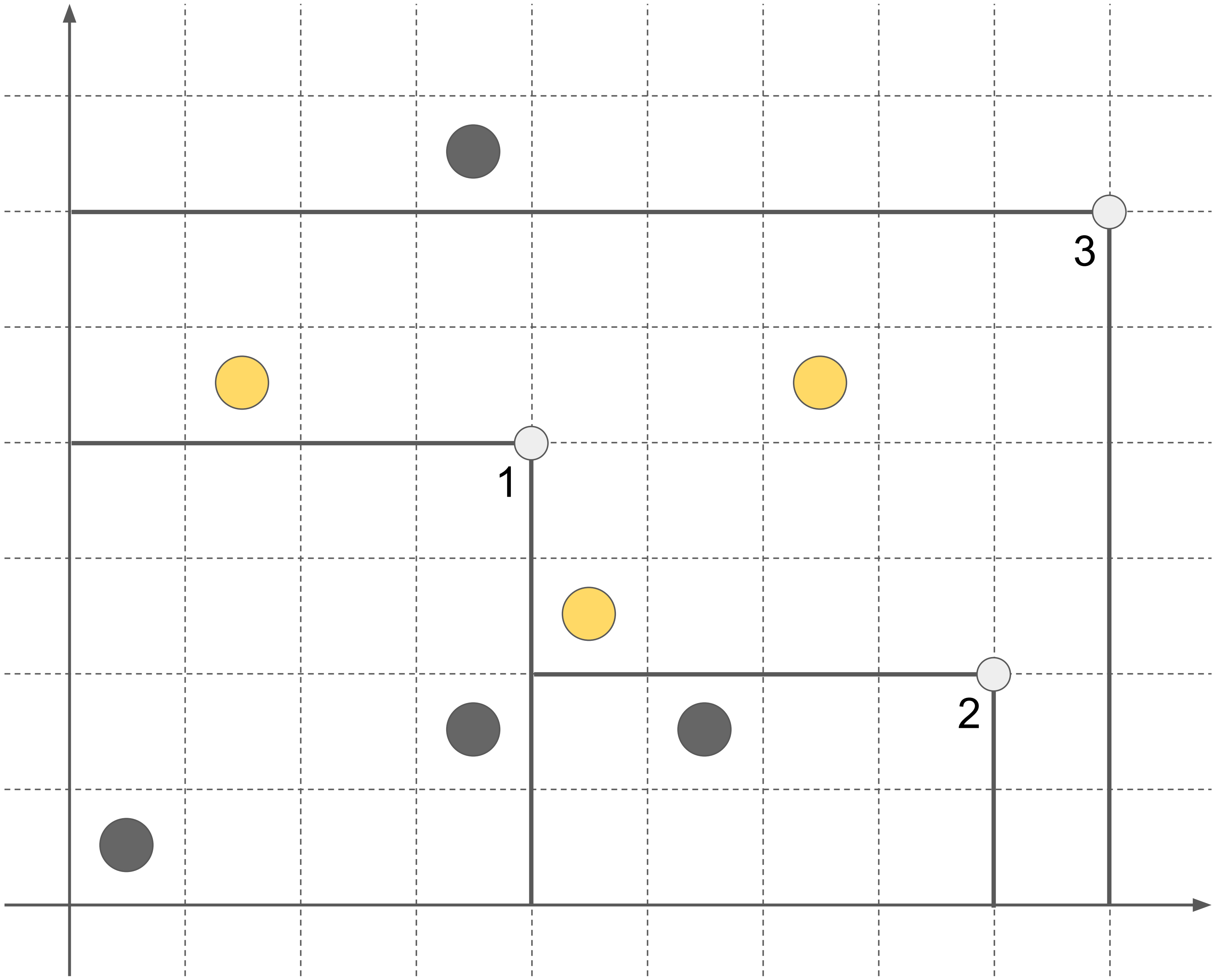
First solved by: UW1
University of Warsaw
(Dębowski, Radecki, Sommer)
02:40:43

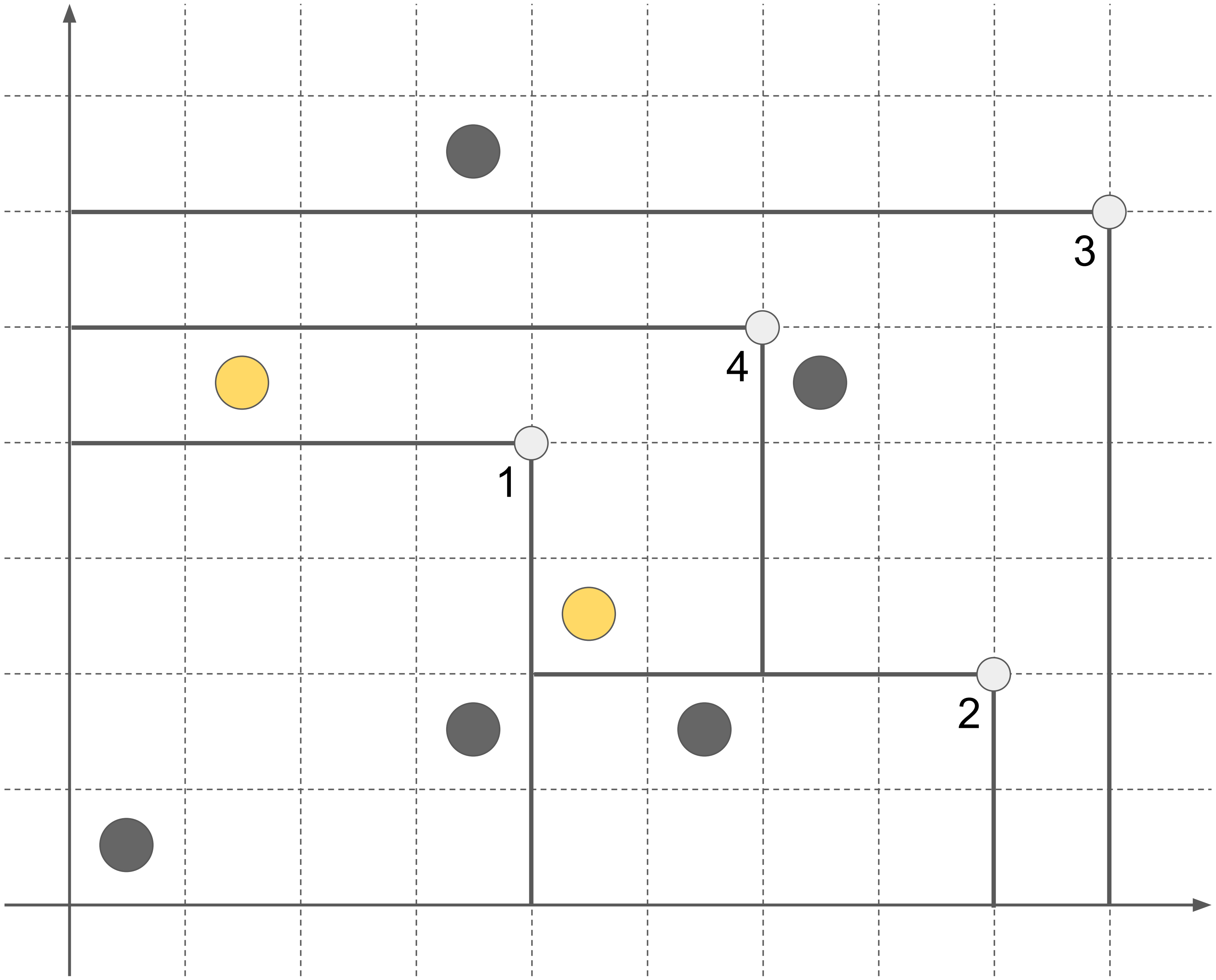
Author: Luka Kalinovčić




1


1
2


1
2
3


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.

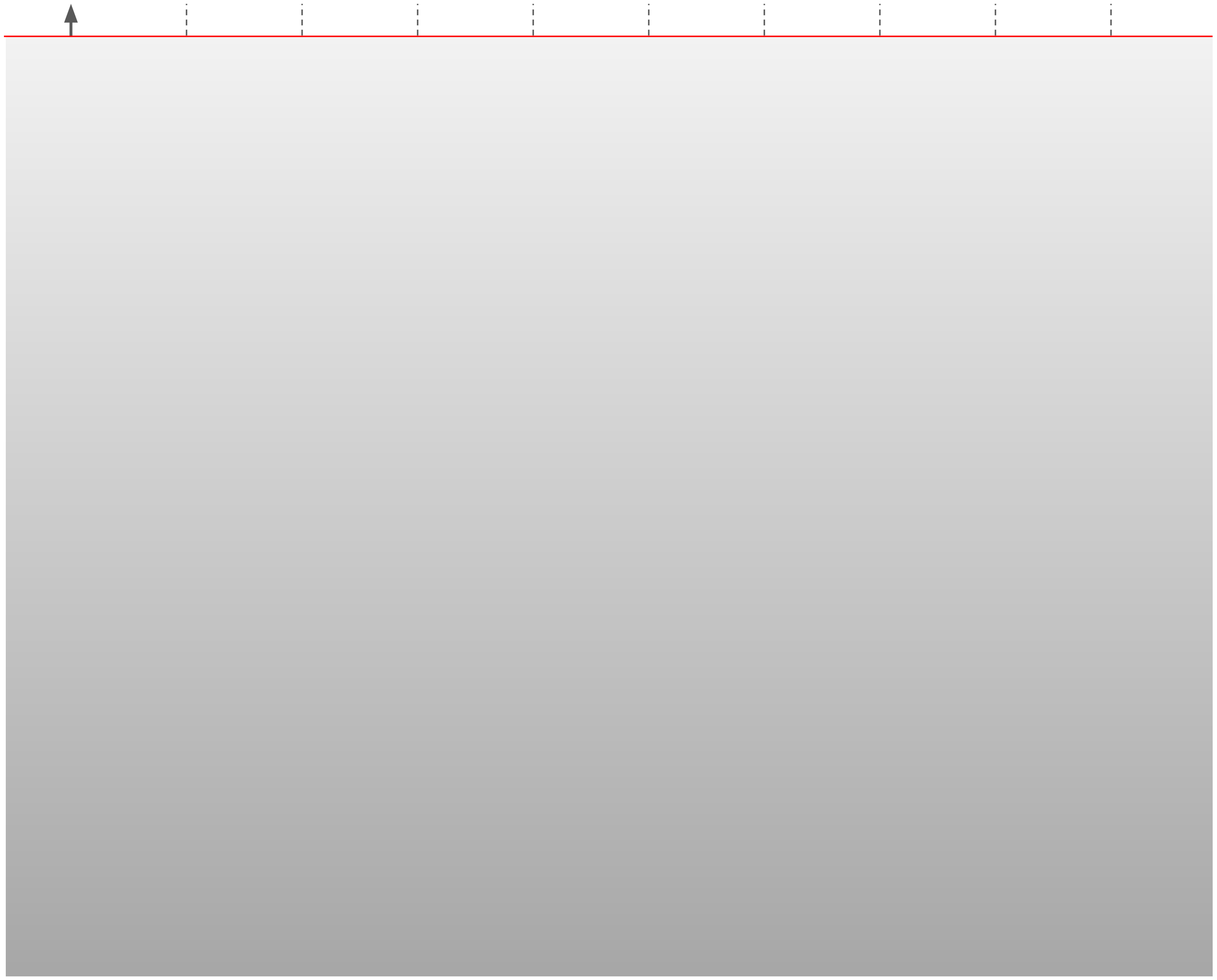
b) When we encounter a fence, we find the neighboring region that it will get merged with when the fence is removed the same way: it's the first active fence to the right.

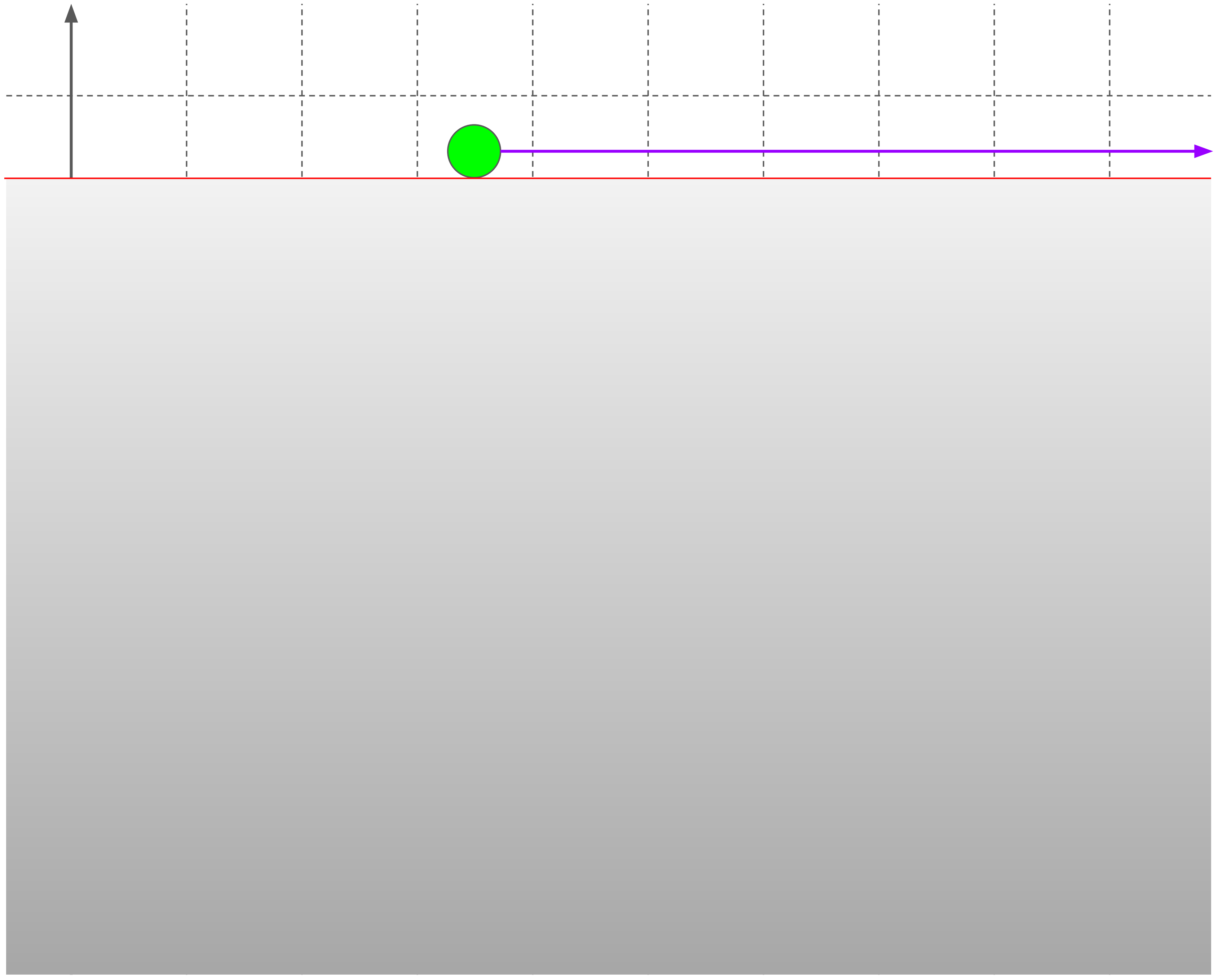
Sweep-line algorithm overview:

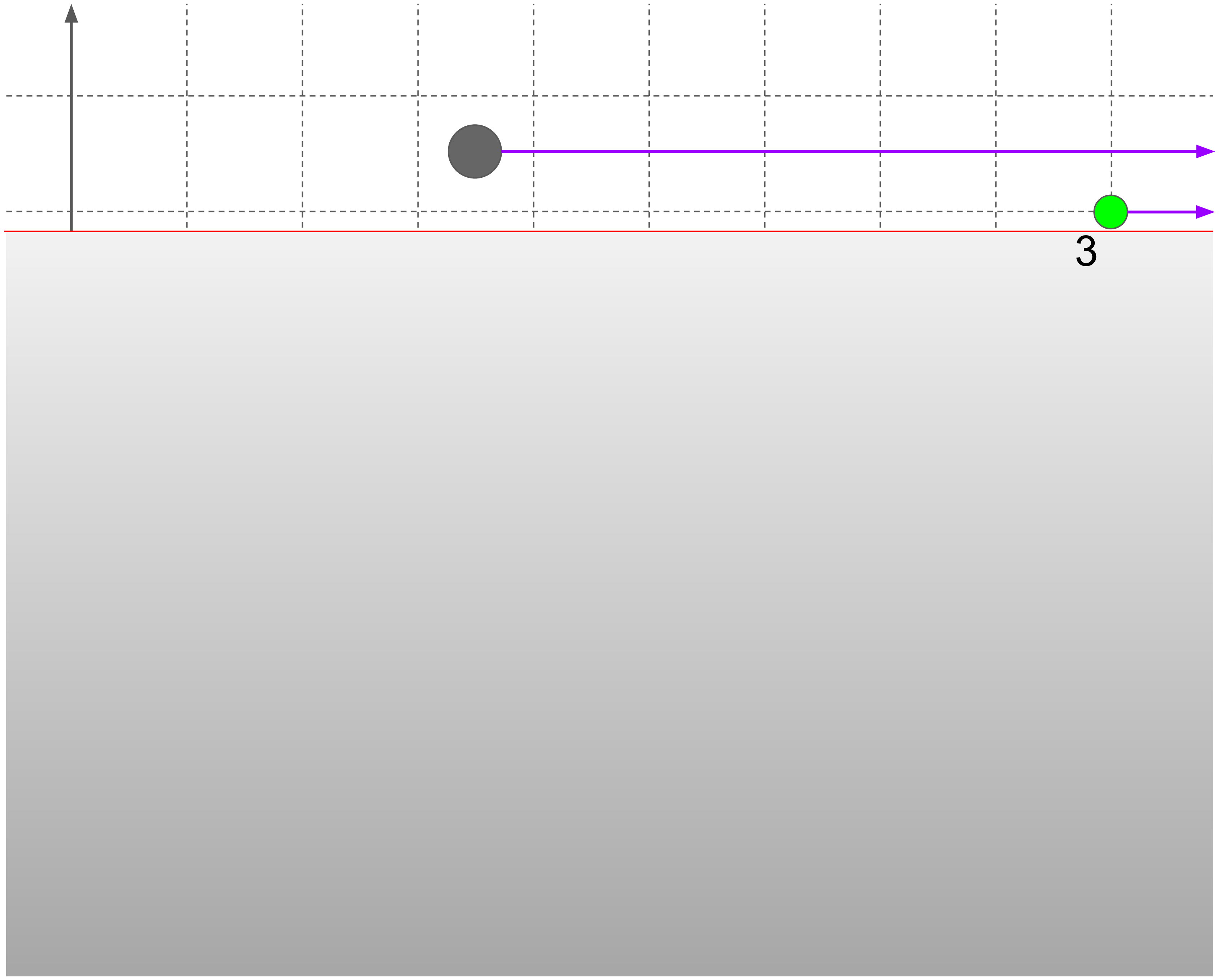
We process fence posts and buffalos in order of decreasing y coordinate.

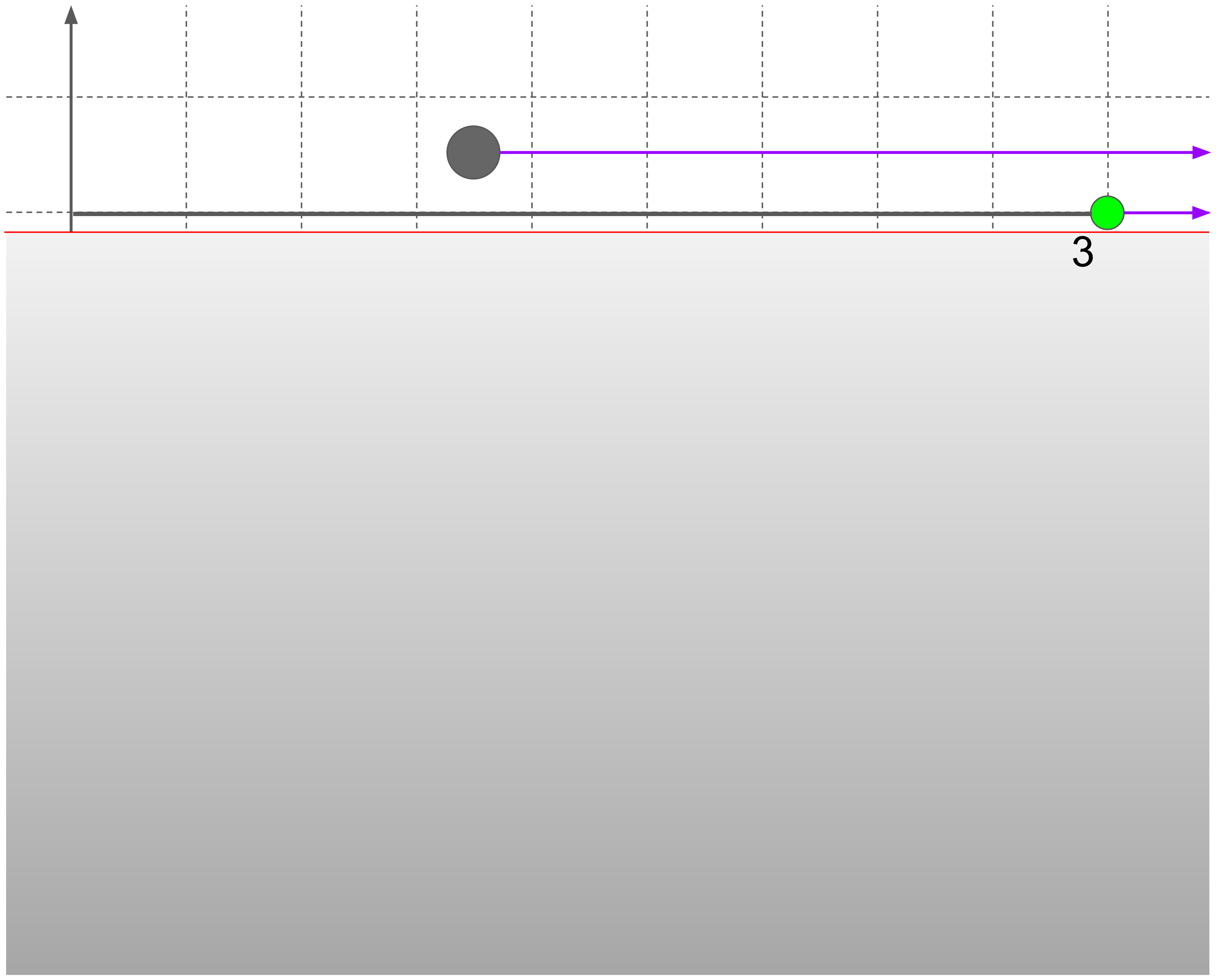
At each step we maintain a set of "active" vertical fences that have not yet hit another horizontal fence.

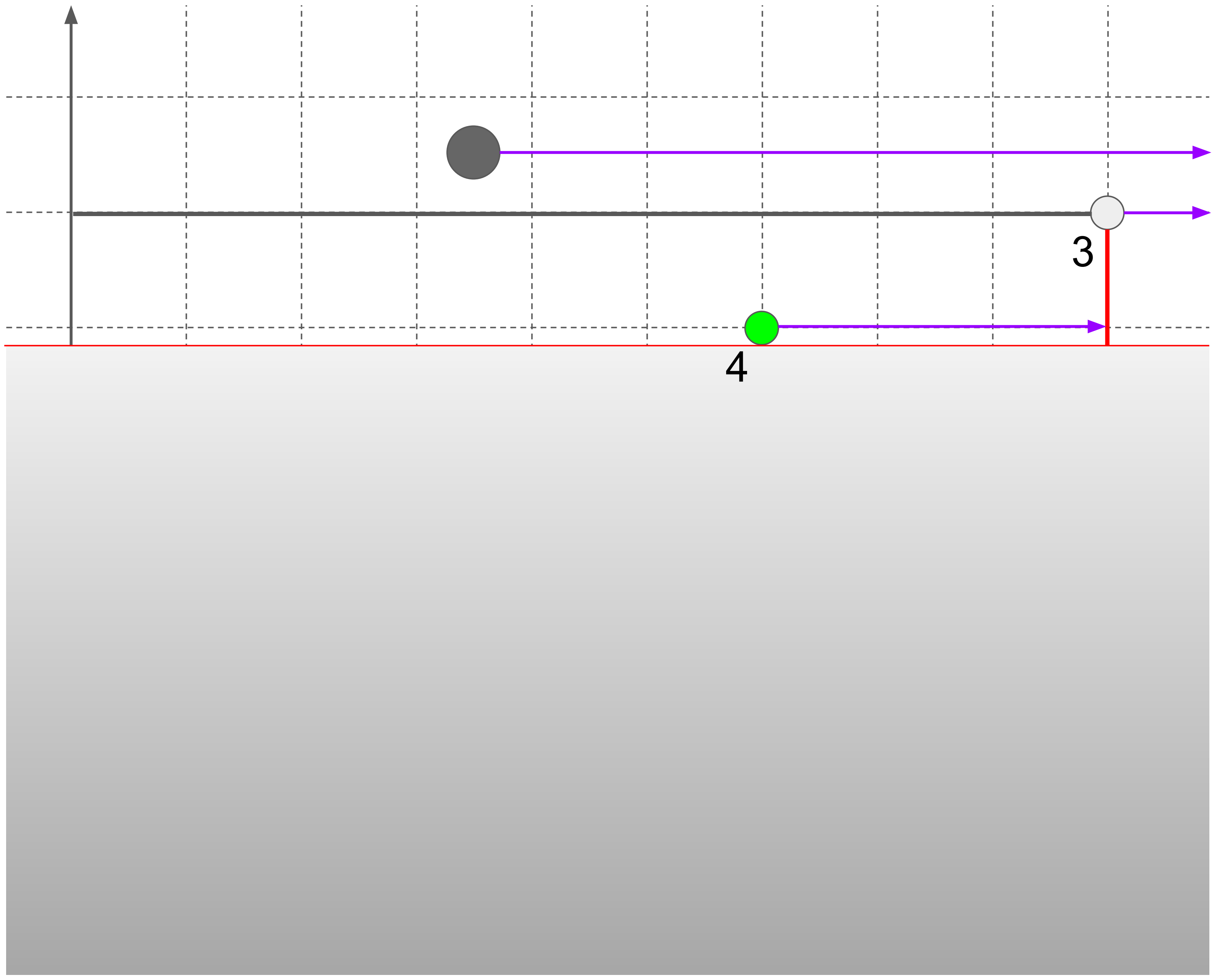
c) We also erect the horizontal fence starting from the fence post going to the left. Our fence will hit the first active fence to the left that has a smaller index (i.e. was erected prior to this fence). Other vertical fences we encounter along the way will, in turn, hit the horizontal fence we are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

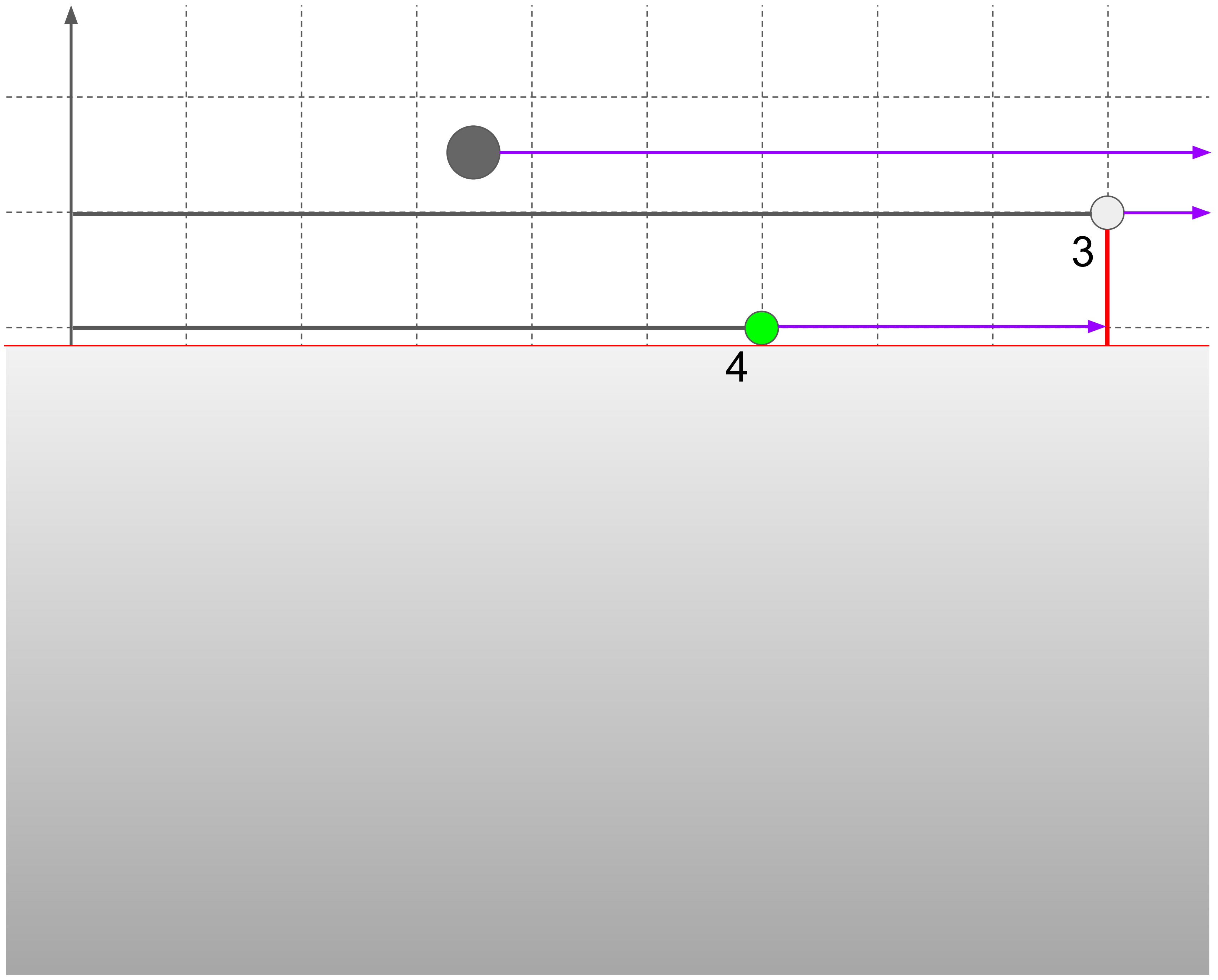


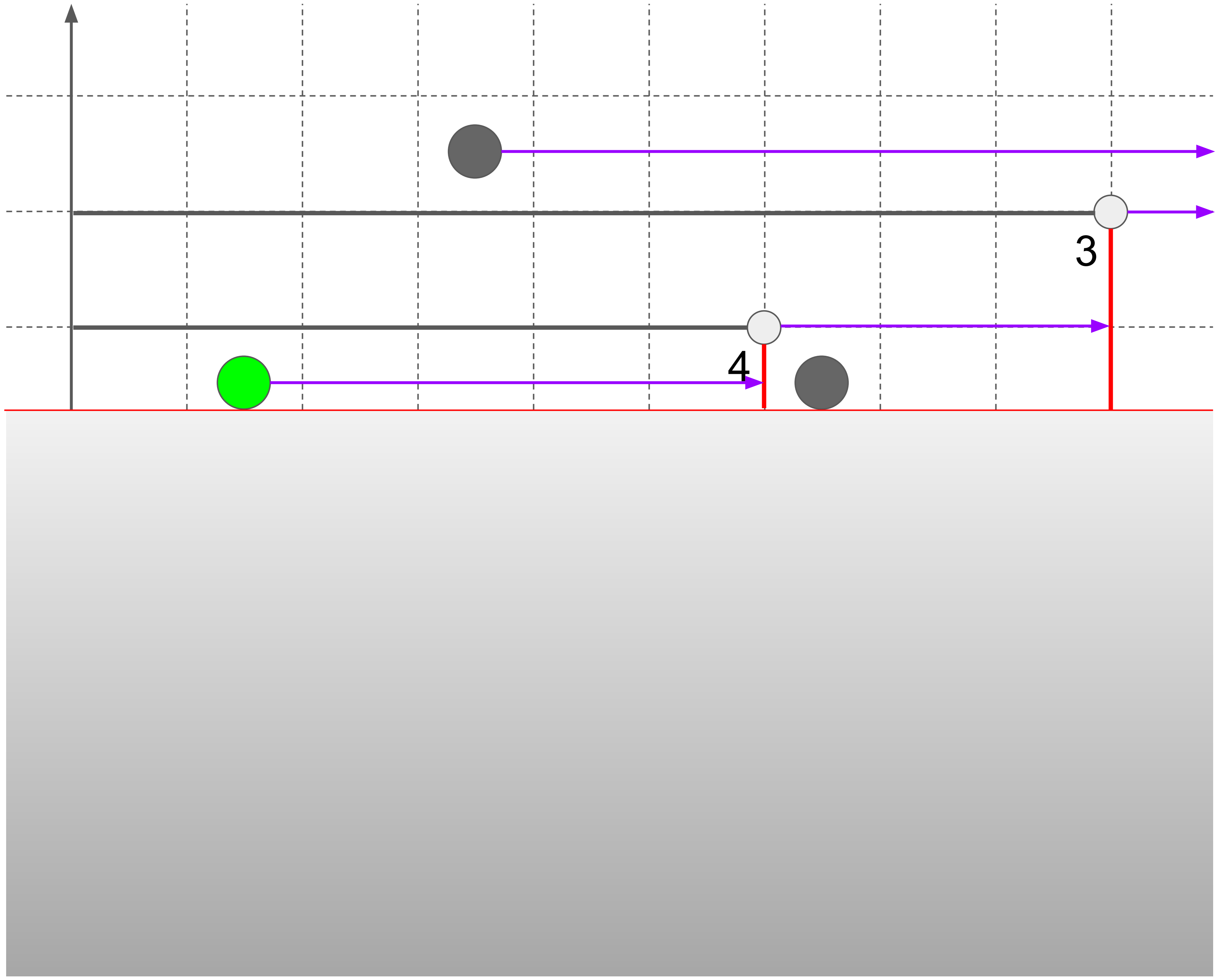


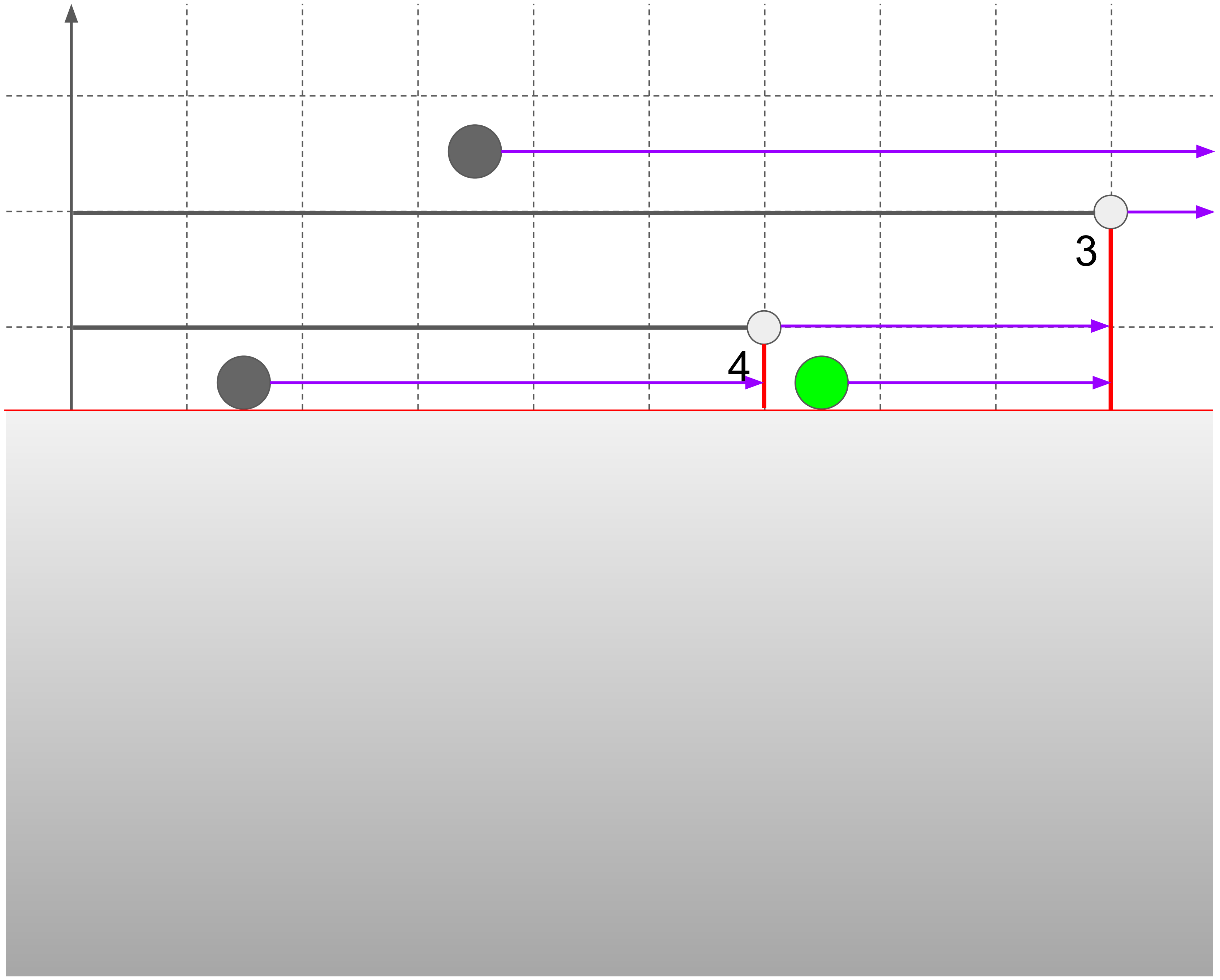

3

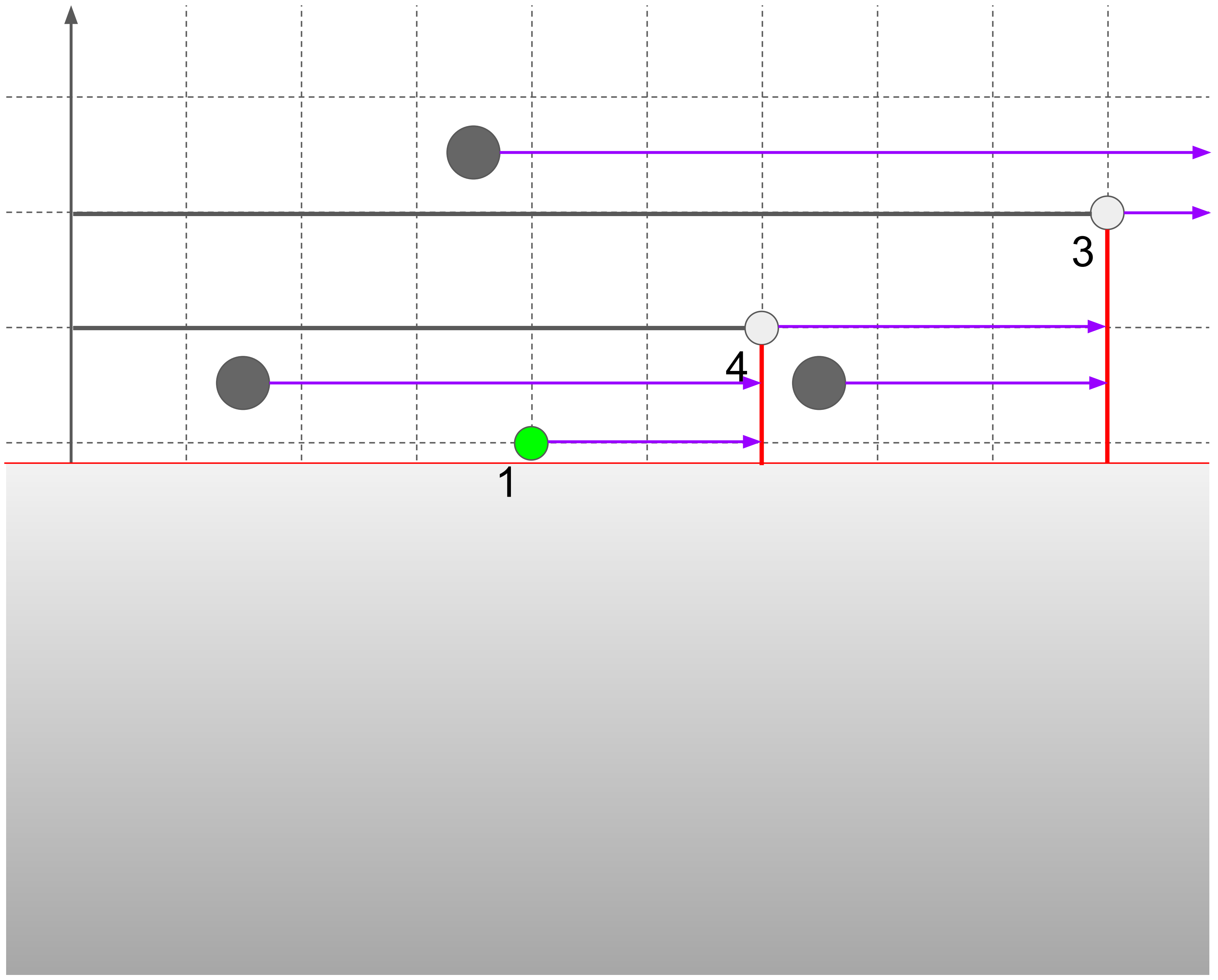

3

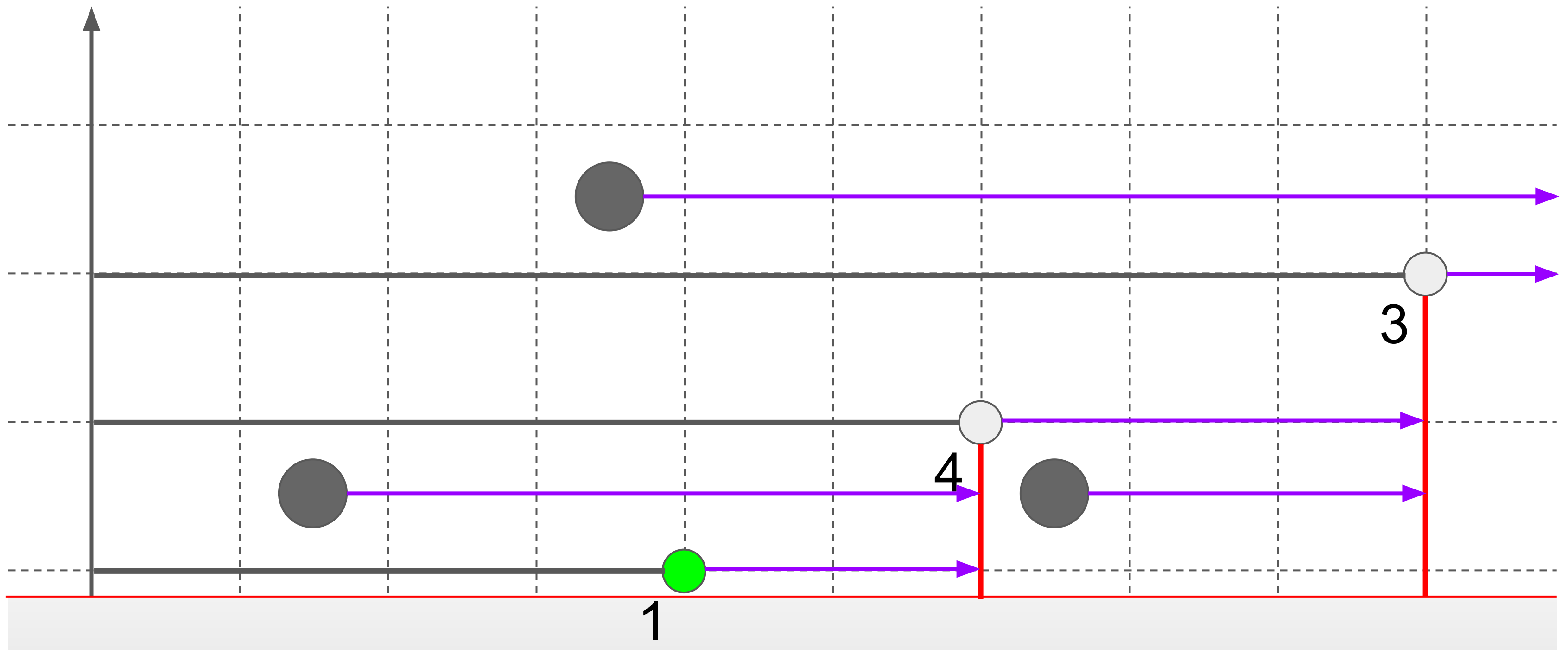

3
4

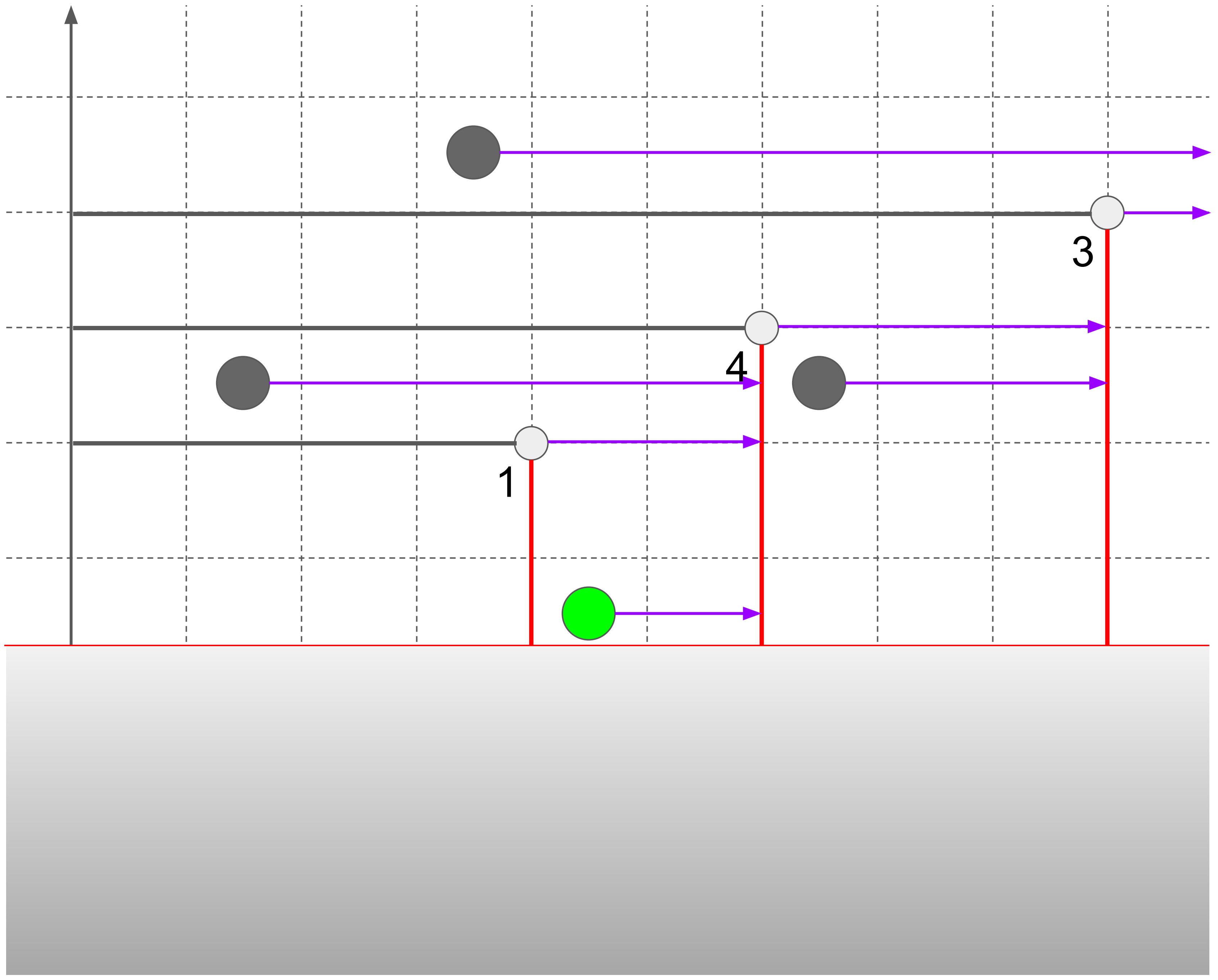

3
4

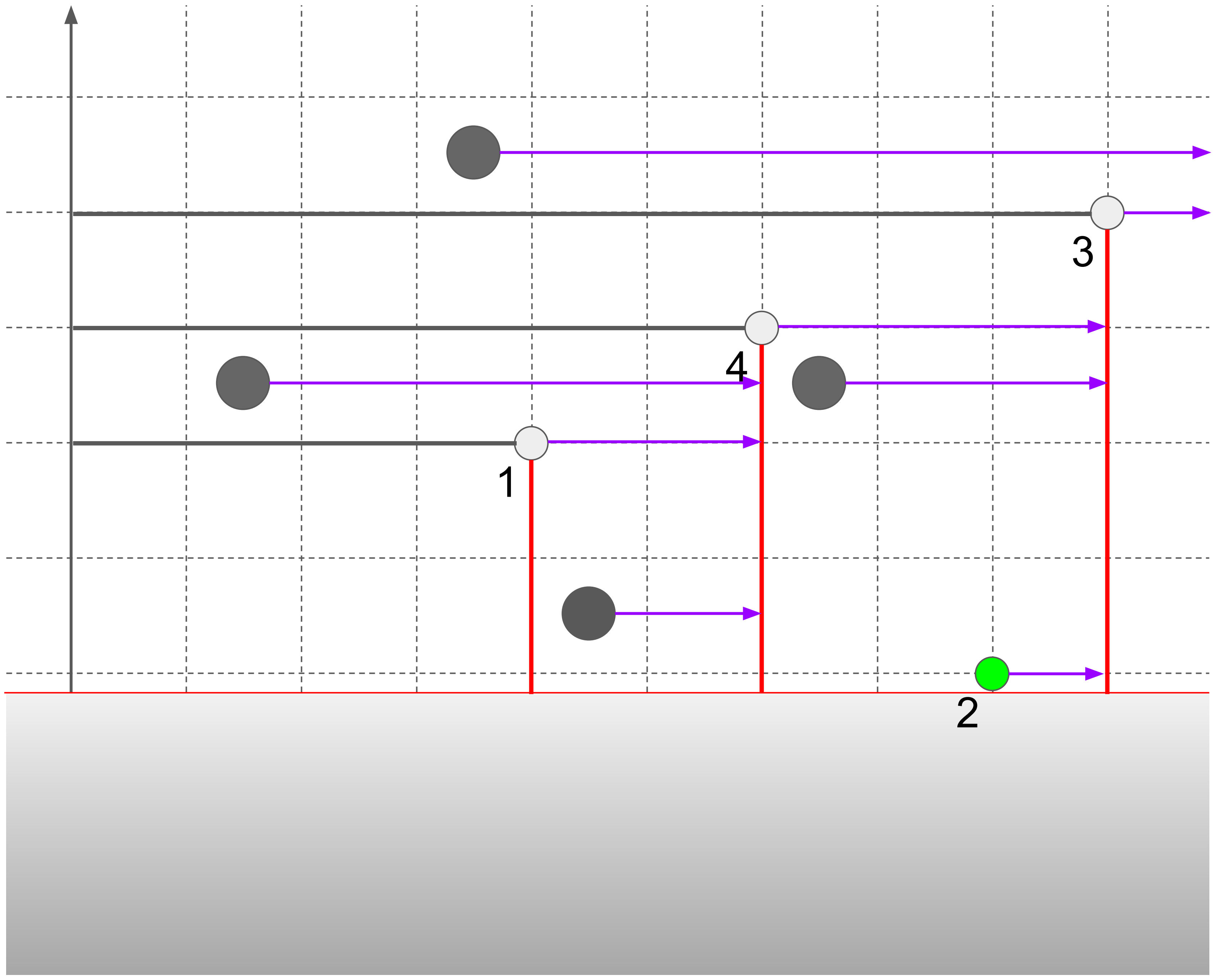

3
4

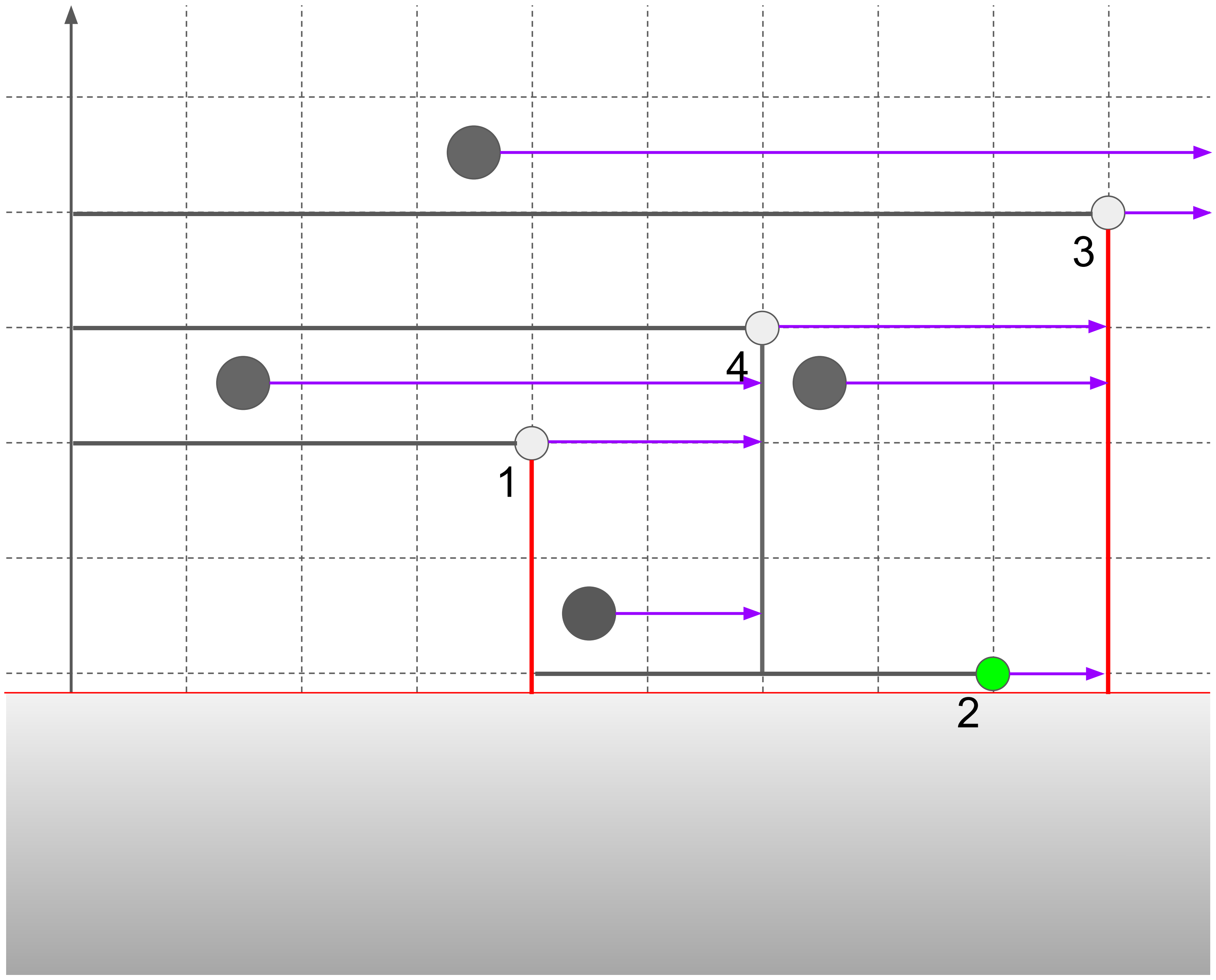

3
4

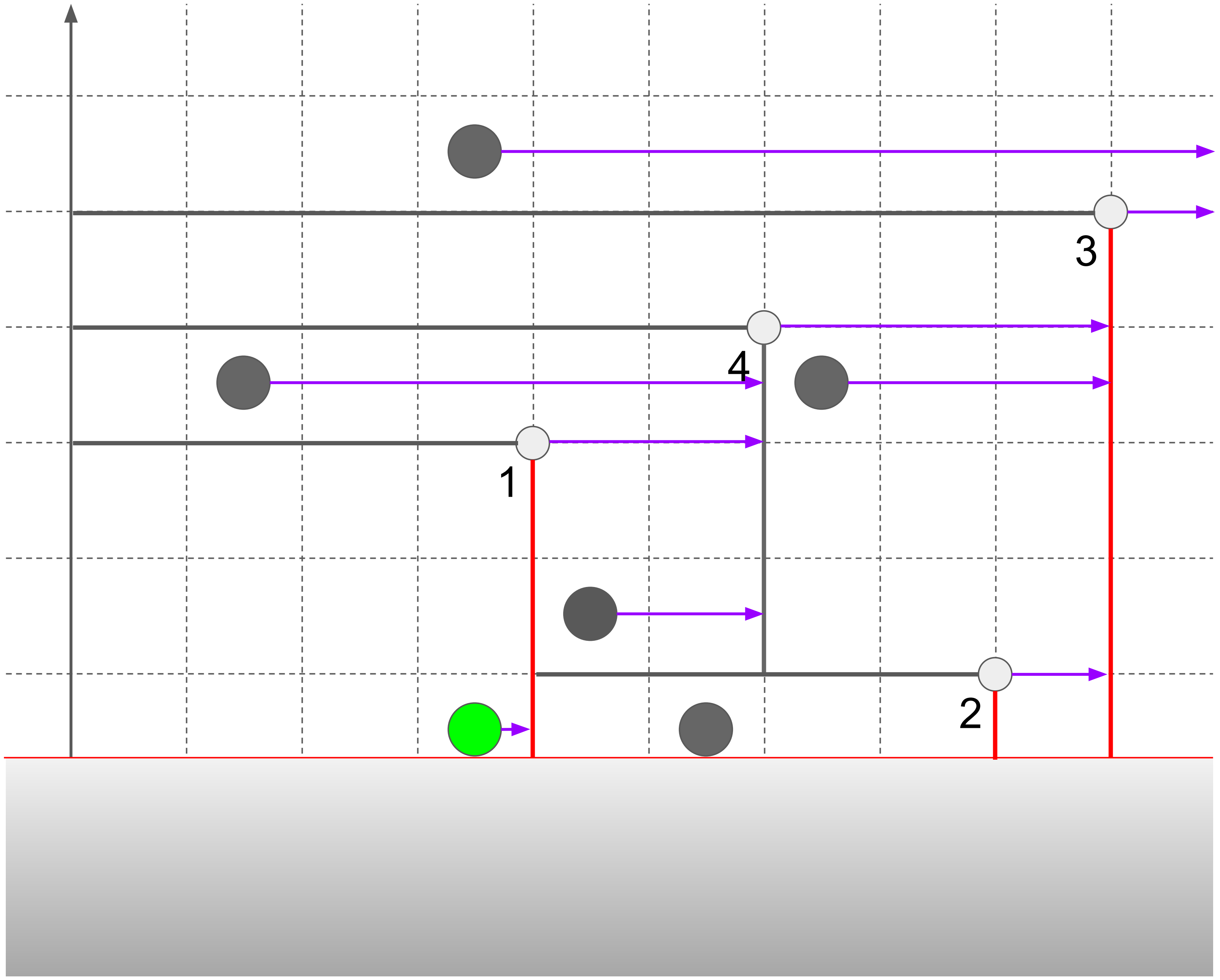

3
4
1

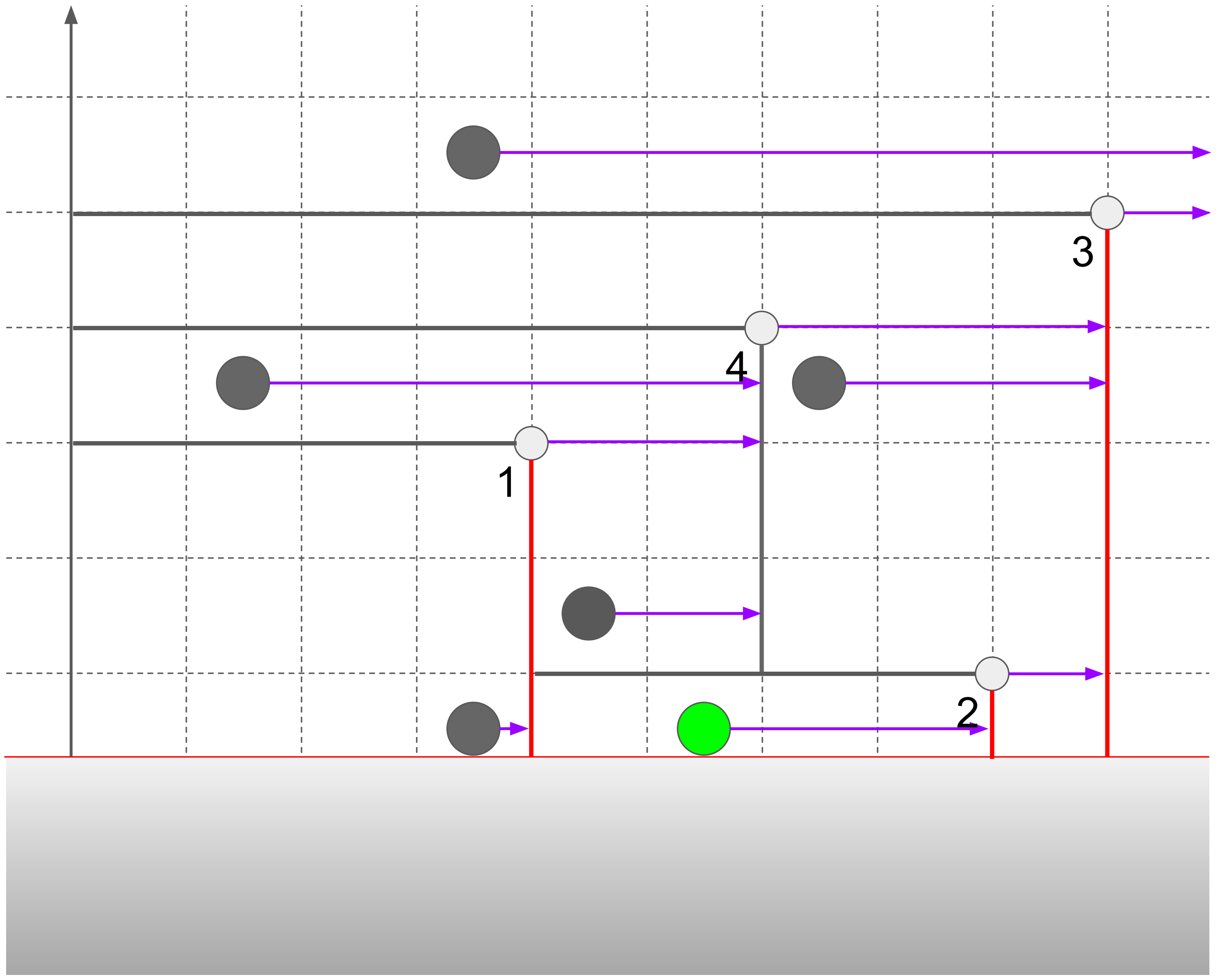

3
4
1

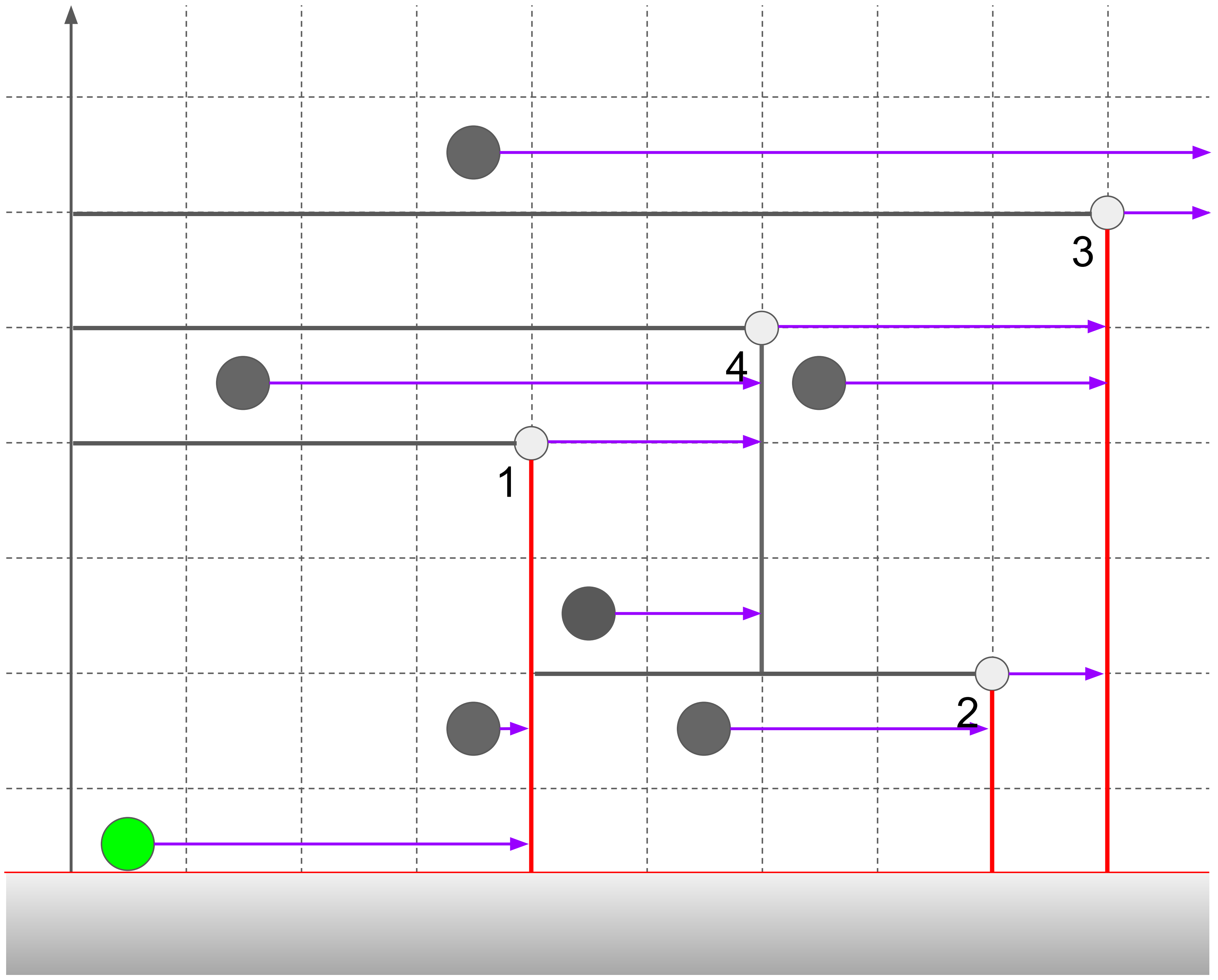

3
4
1

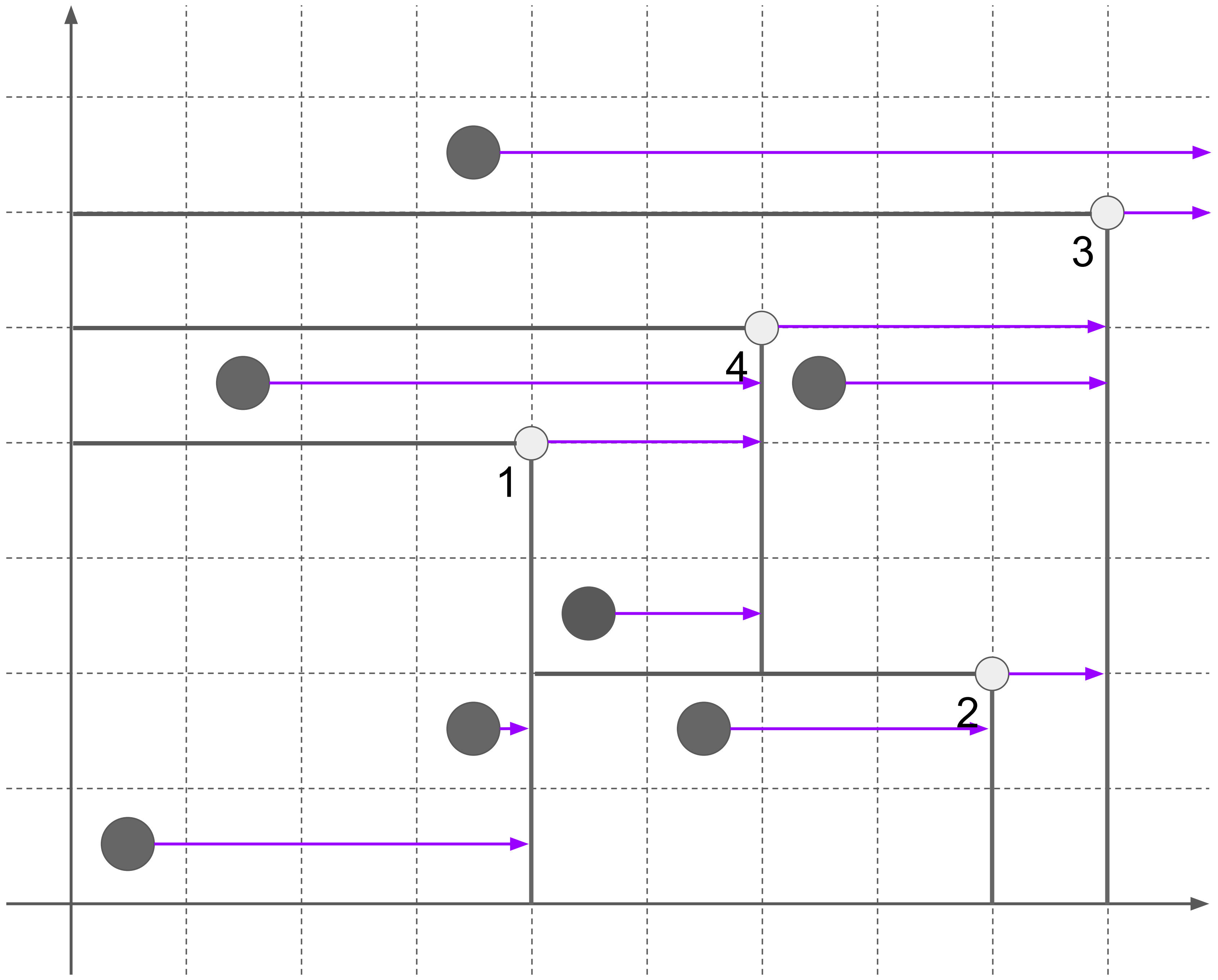

3
4
1
2


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))

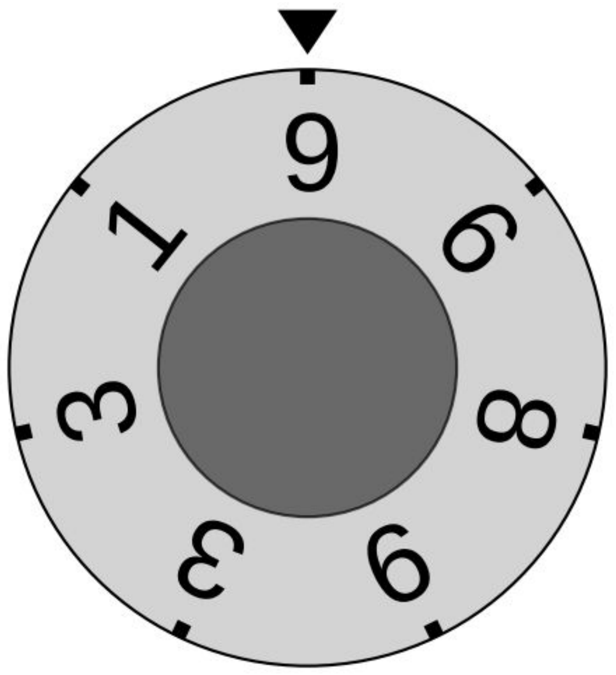
Problem K
Kitchen Knobs

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.

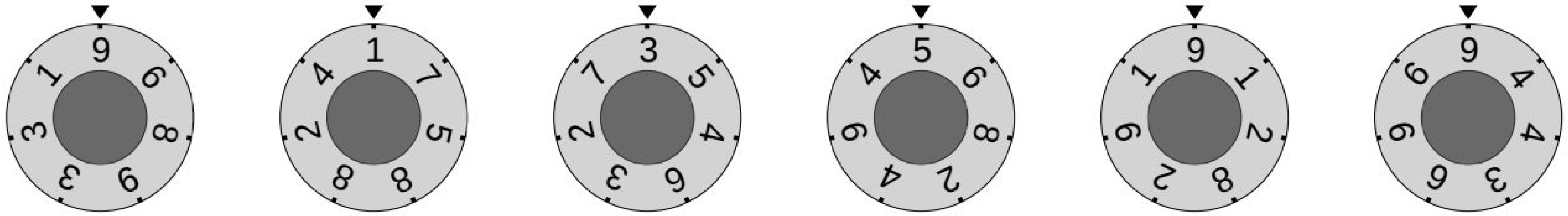
Because we have exactly 7 digits on each knob, every element either has:

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

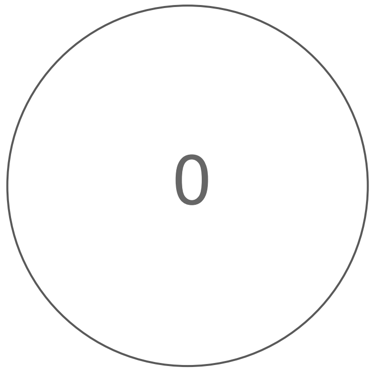
Given a set B with elements from [0, 6], group them into as many groups as possible such that the sum of each group is 0 (modulo 7).

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).

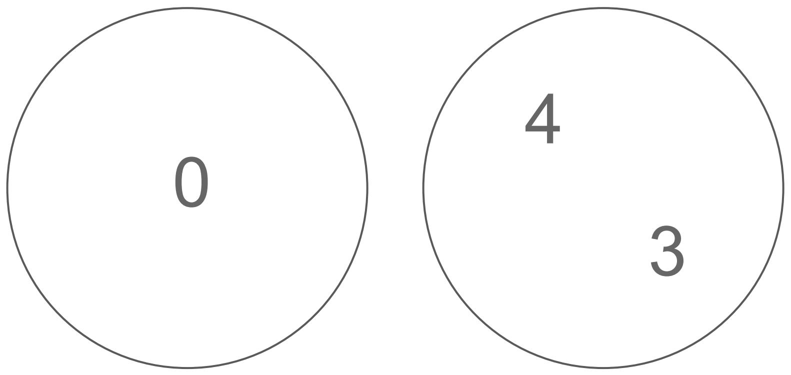
B: 1 4 1 3 5 5 4 1 4



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).

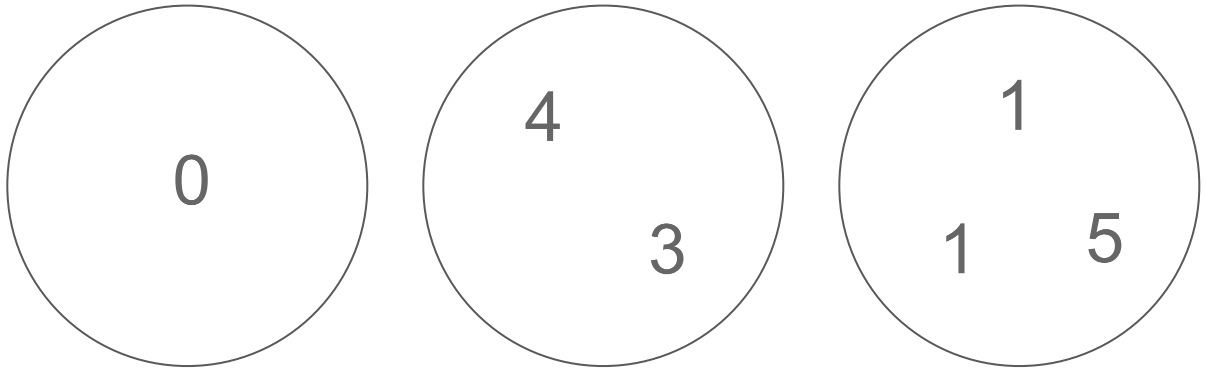
B: 1 1 5 5 4 1 4



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).

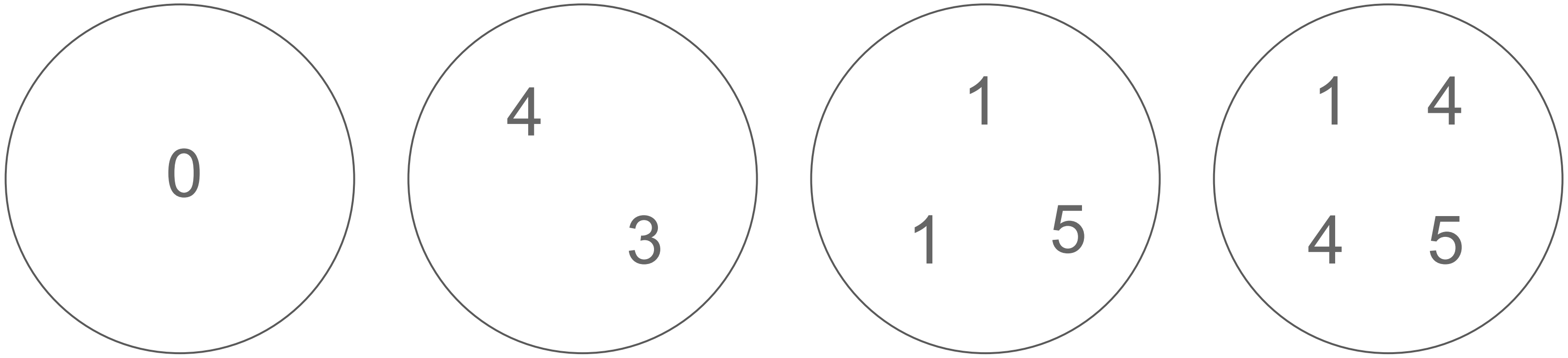
B: 1 5 4 4



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.

$$\boxed{2\ 3\ 1}\ 6\ 4\ 7\ 5\ 8$$

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

$$2\ 3\ 1\ \boxed{6\ 4\ 7\ 5}\ 8$$

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

$$2\ \ 3\ \ 1\ \boxed{6\ \ 4\ \ 7\ \ 5\ \ 8}$$

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

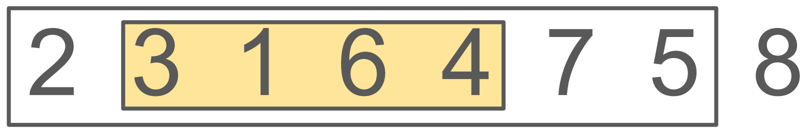
$$2 \quad 3 \quad 1 \quad 6 \quad \boxed{4} \quad 7 \quad 5 \quad 8$$

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

$$\boxed{2\ \ 3\ \ 1\ \ 6\ \ 4\ \ 7\ \ 5\ \ 8}$$

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

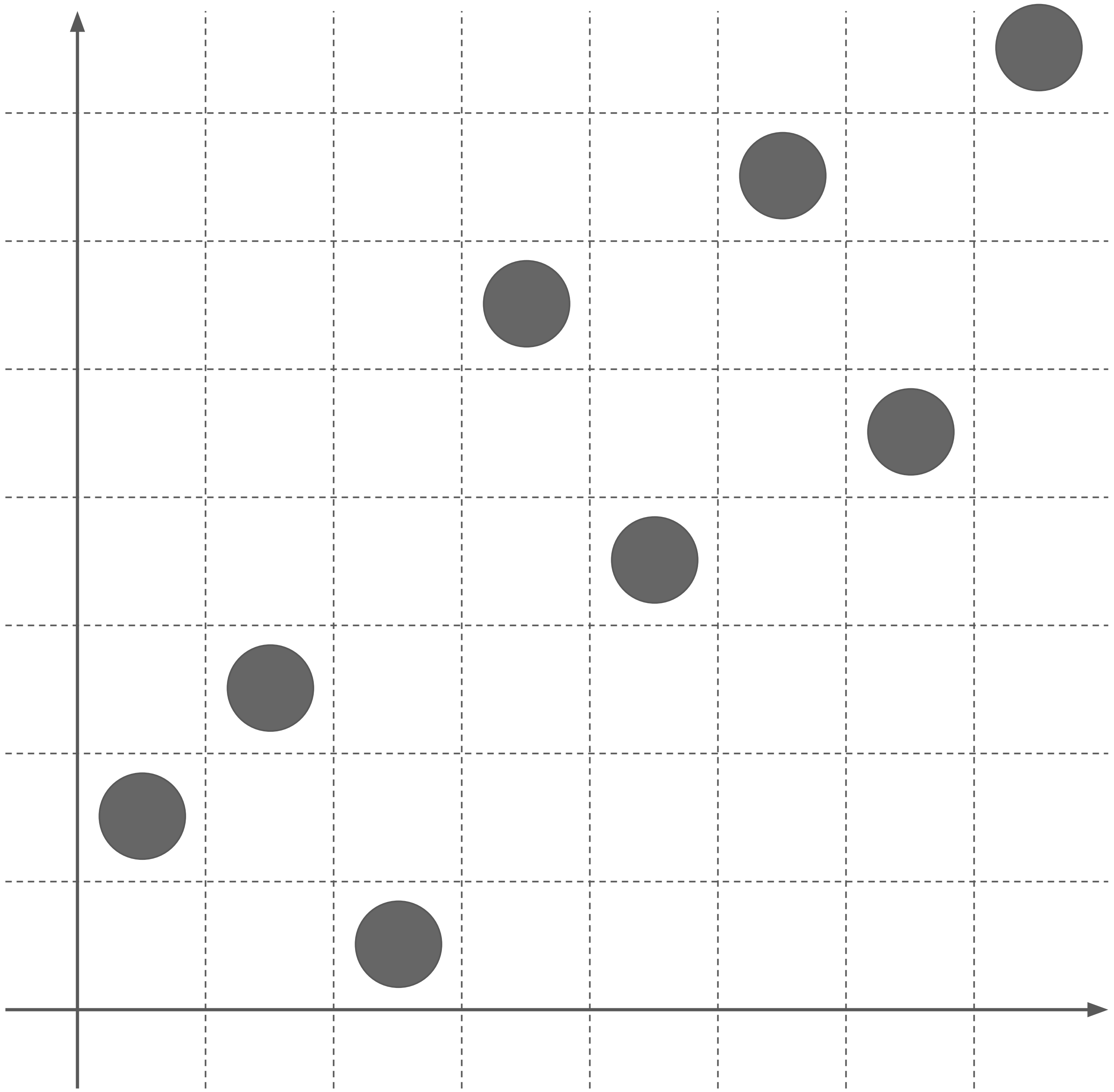
For a given subsequence we need to find the shortest enclosing interval.

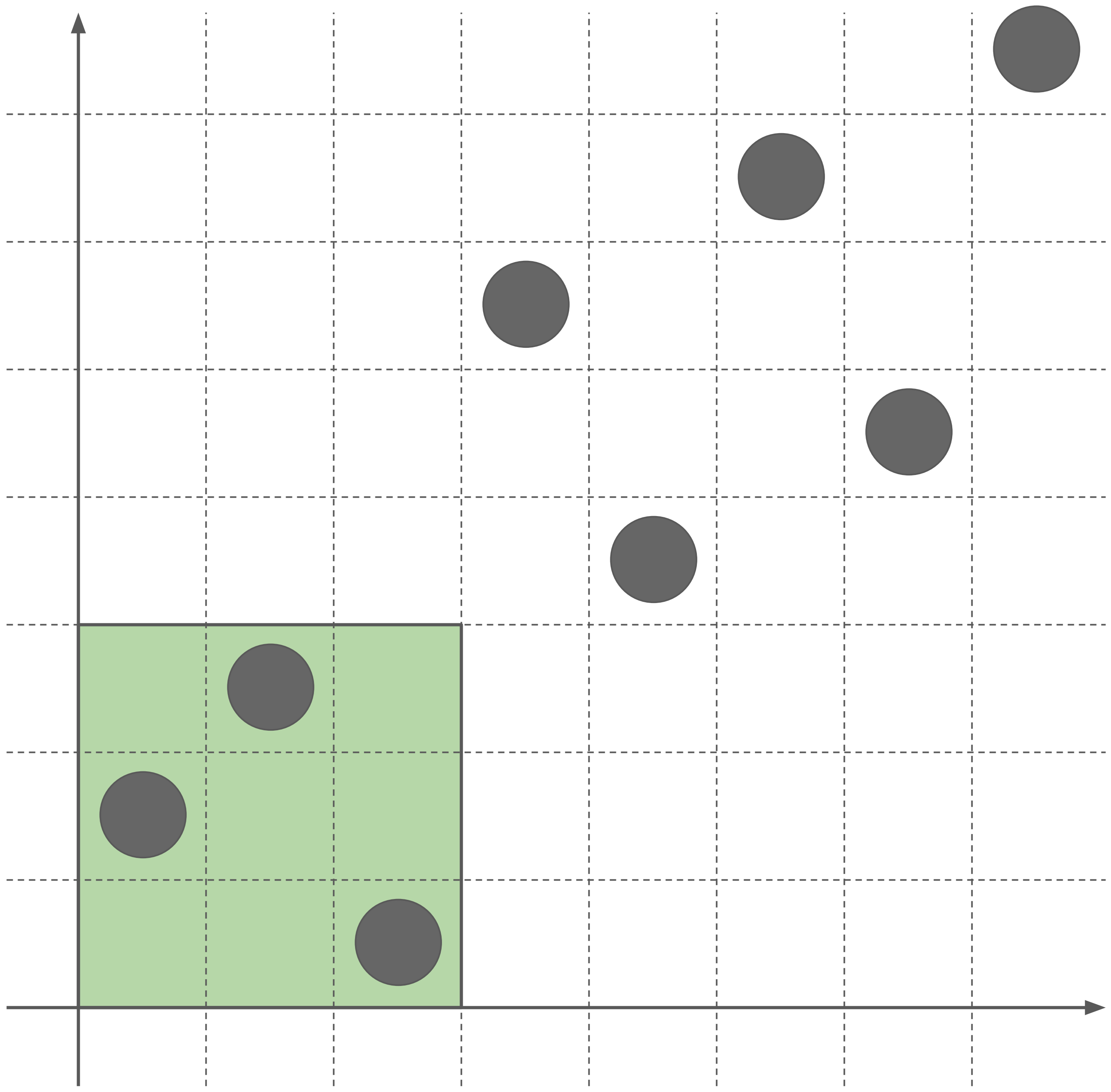


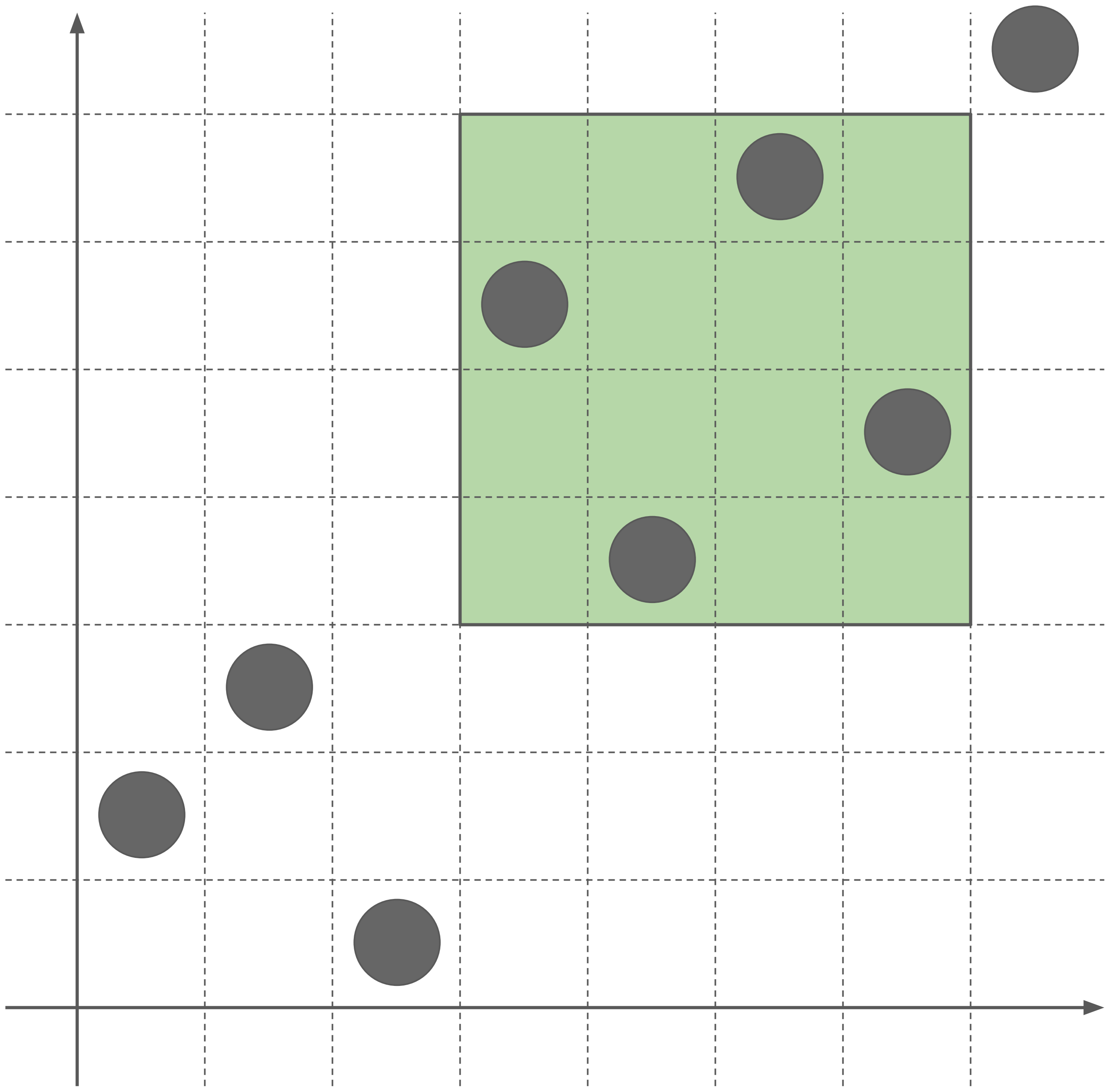
An interval of the permutation is a consecutive subsequence consisting of consecutive numbers.

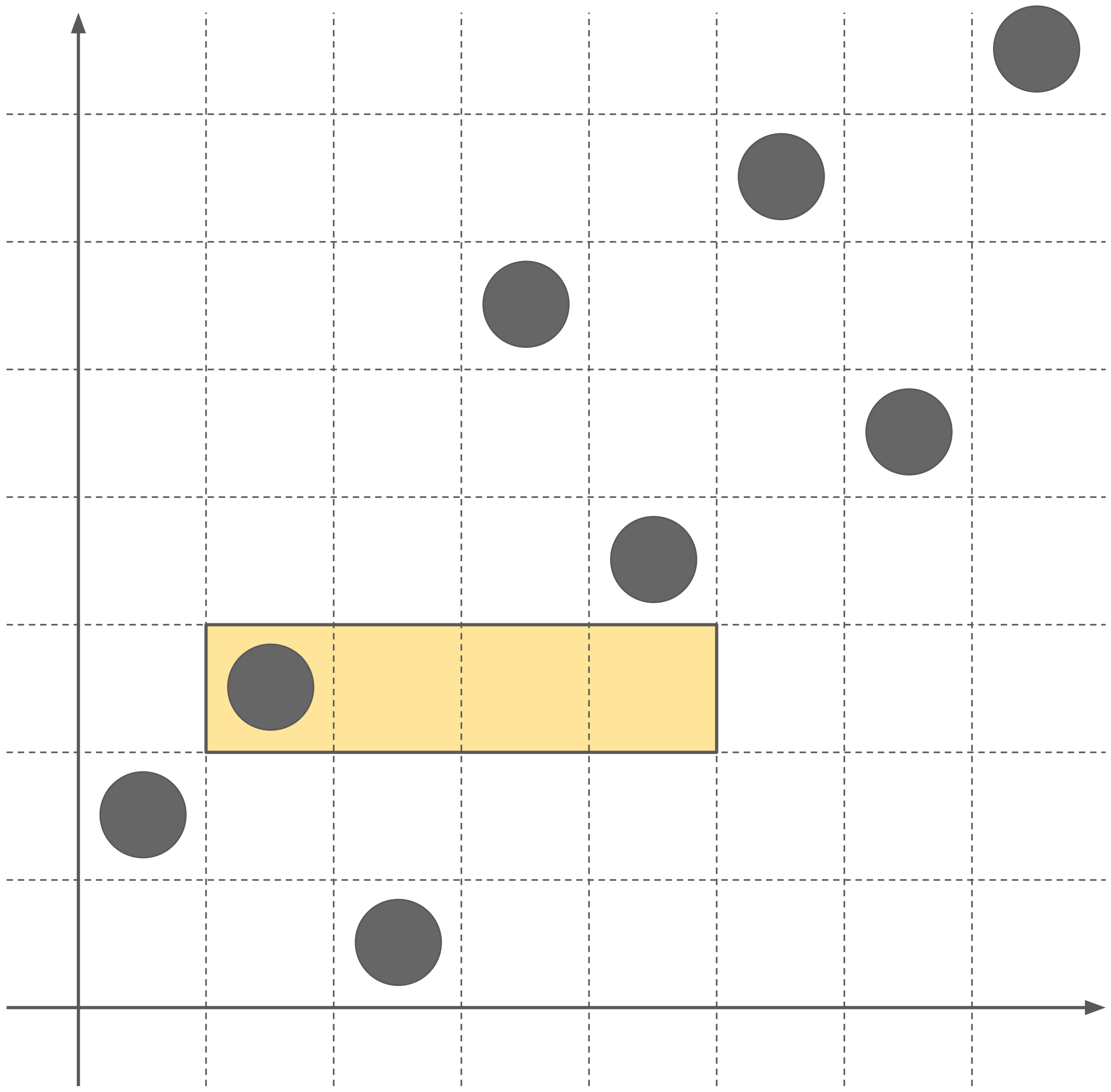
For a given subsequence we need to find the shortest enclosing interval.

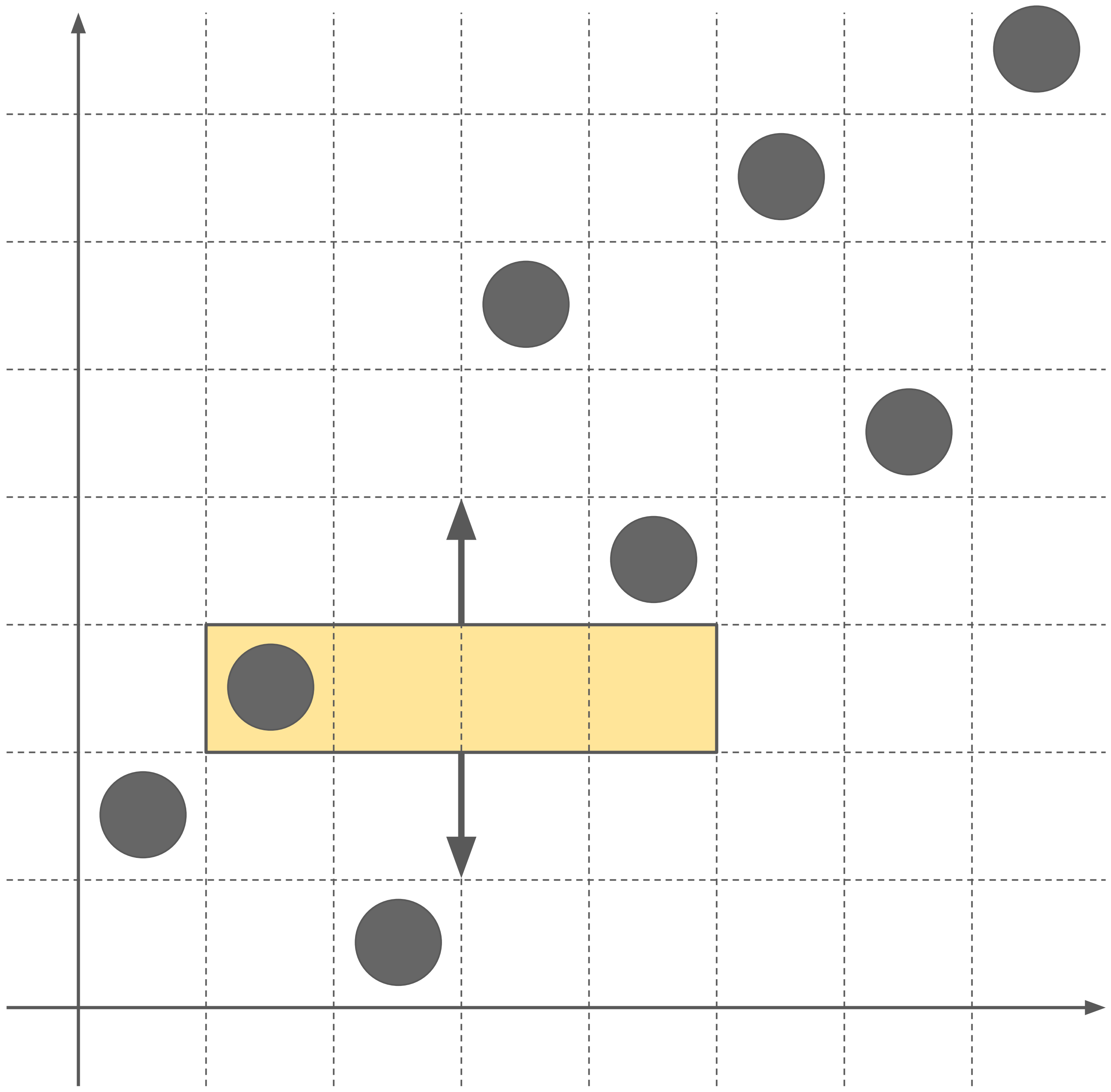
To see how we could expand the subsequence into the shortest enclosing interval, let's visualize the permutation in two dimensions.

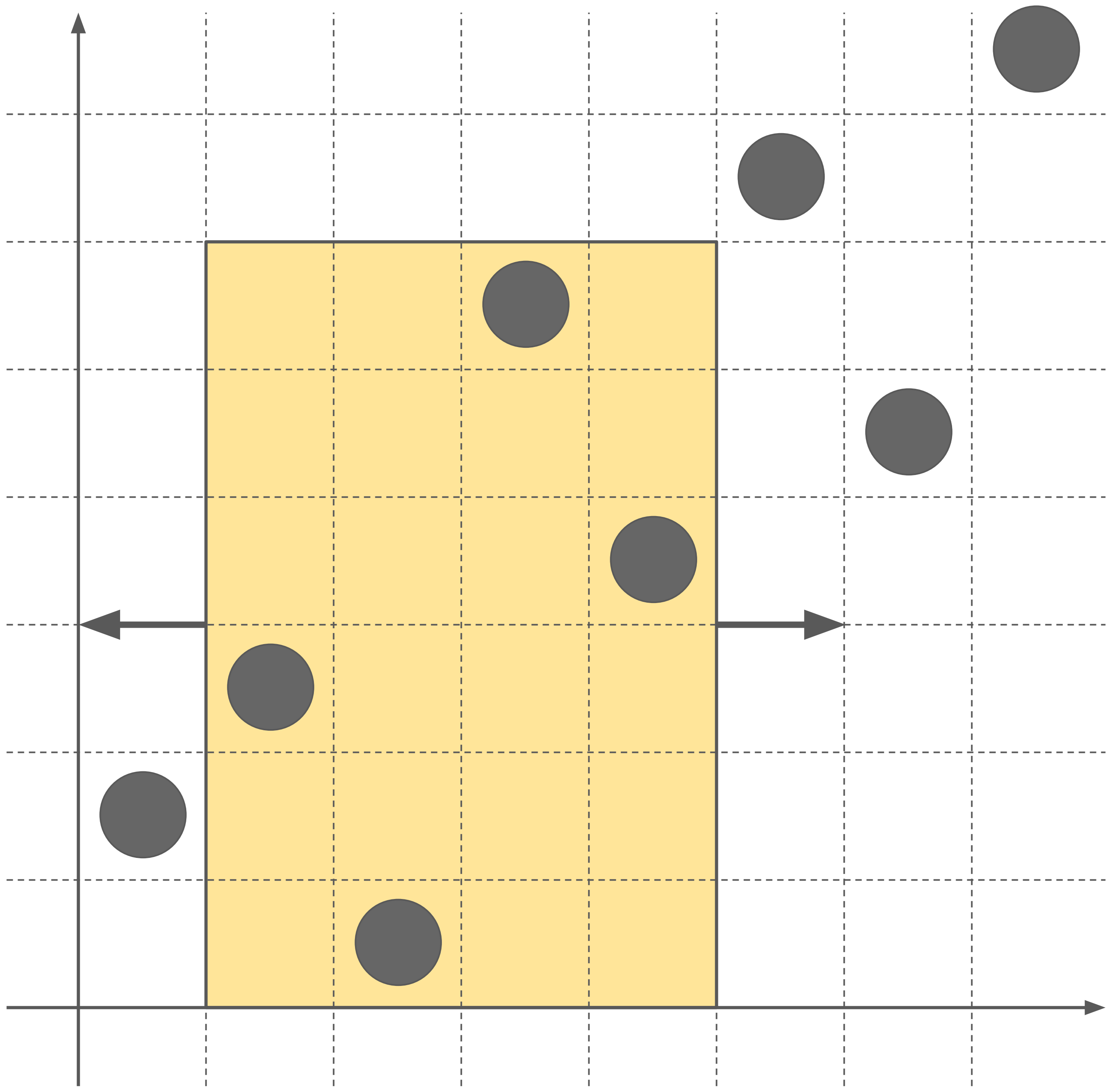


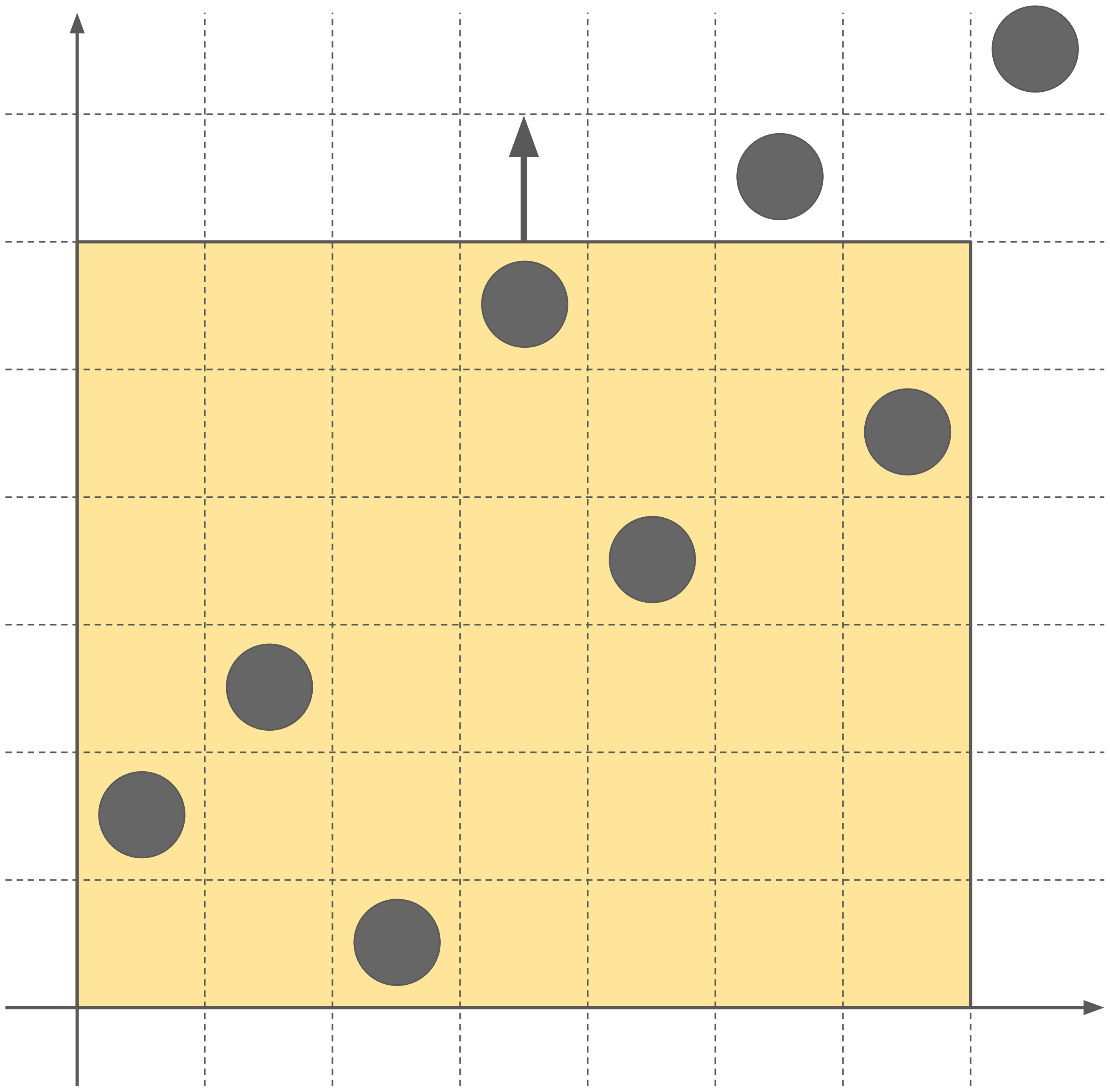


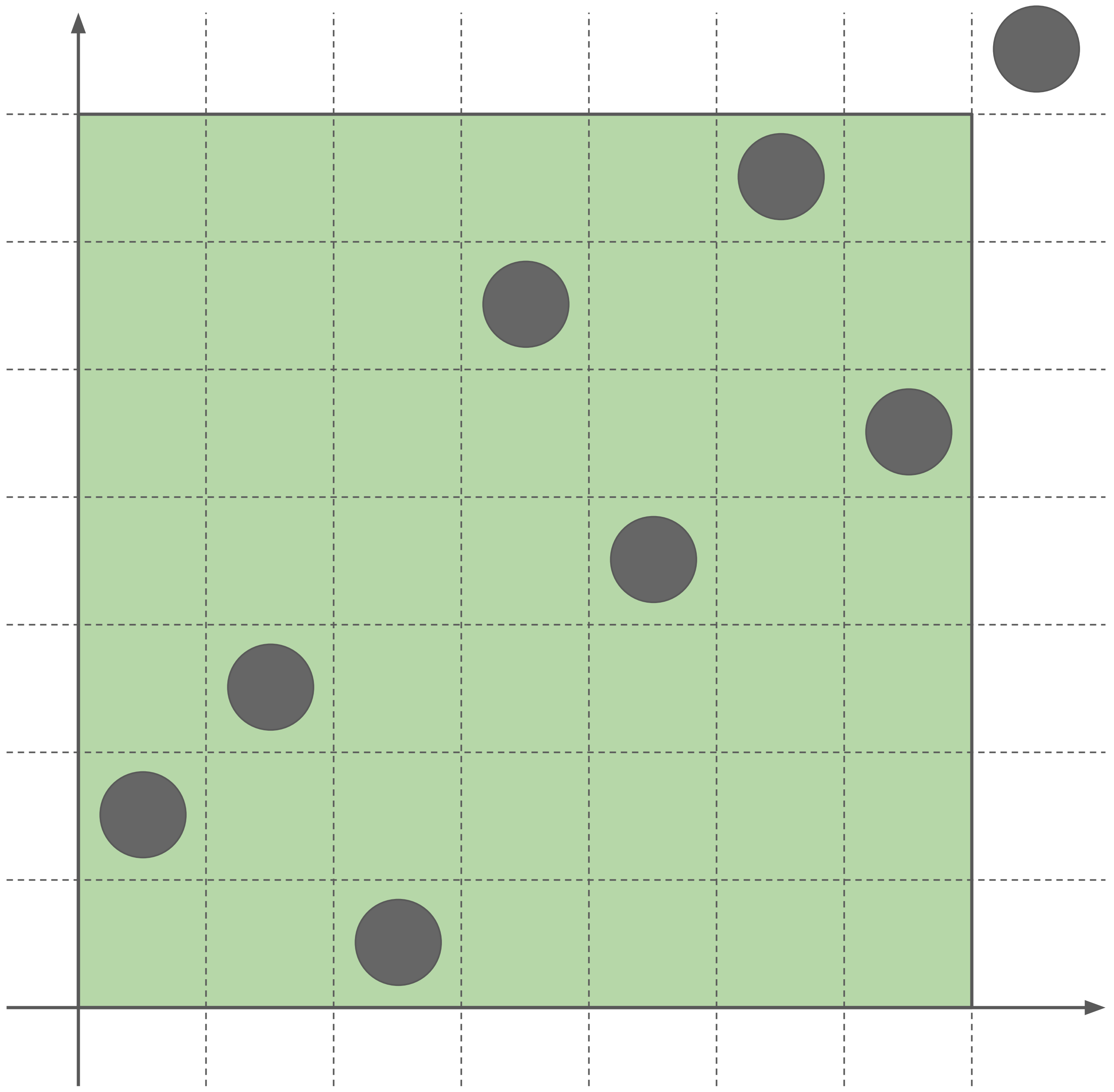












With careful implementation of the algorithm, it is possible to expand a subsequence [a, b] to an enclosing interval [x, y] in O(|y - x| - |b - a|).

However, that's too slow for this problem.

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

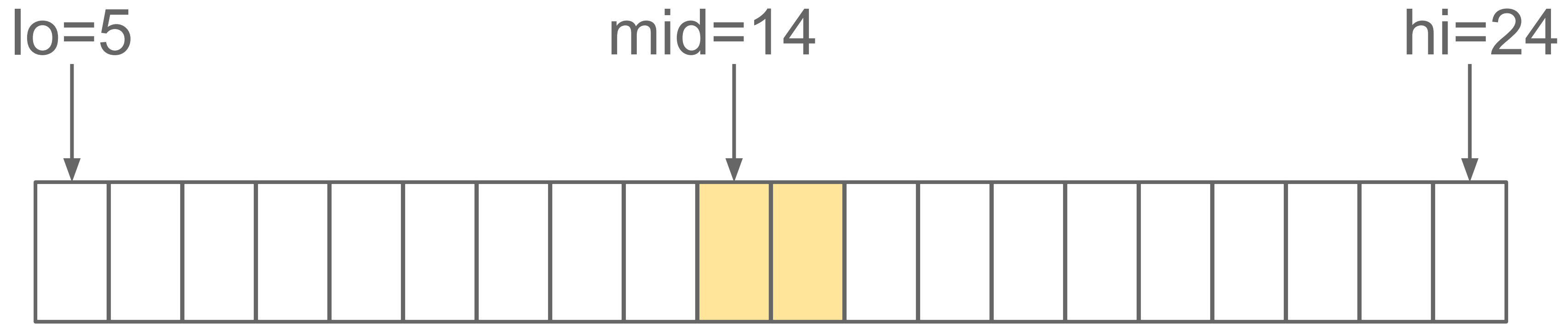
We initialize the result for each query with interval [1, n] and then we'll try to improve it.

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

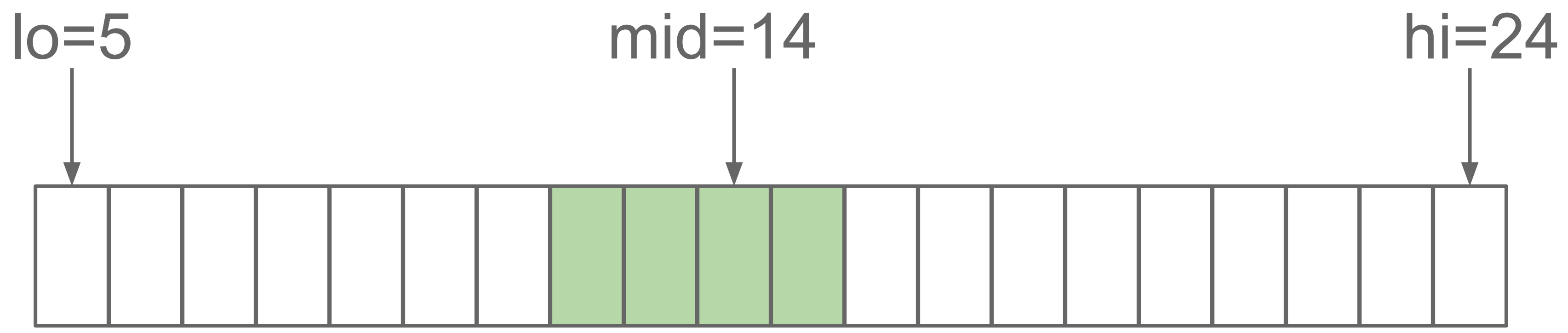
```
Improve(queries, lo, hi):
  if lo == hi: return
  mid = (lo + hi) / 2
  Improve([q in queries where q.b <= mid], lo, mid)
  Improve([q in queries where q.a > mid], mid + 1, hi)

  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.

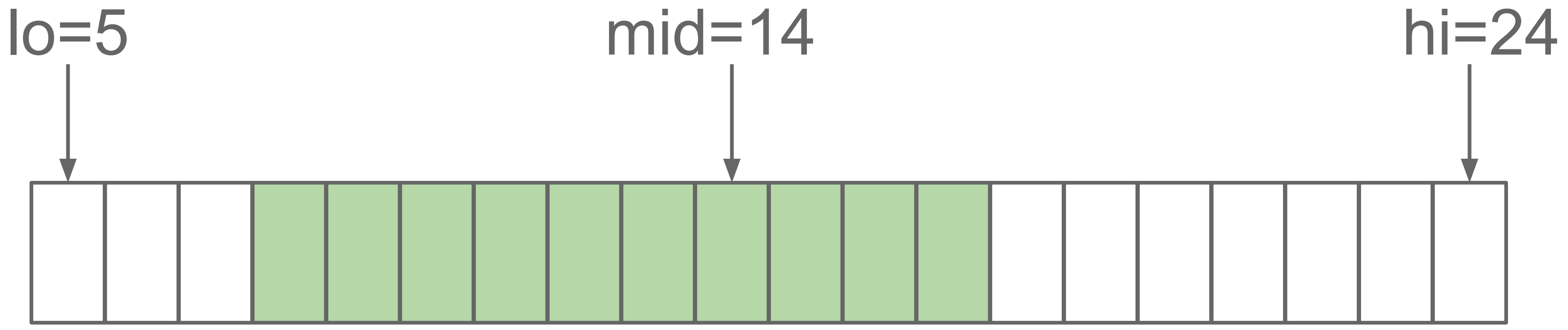


Starting from subsequence [mid, mid + 1], we expand it to the left, storing all intervals we encounter until we exit the [lo, hi] window.



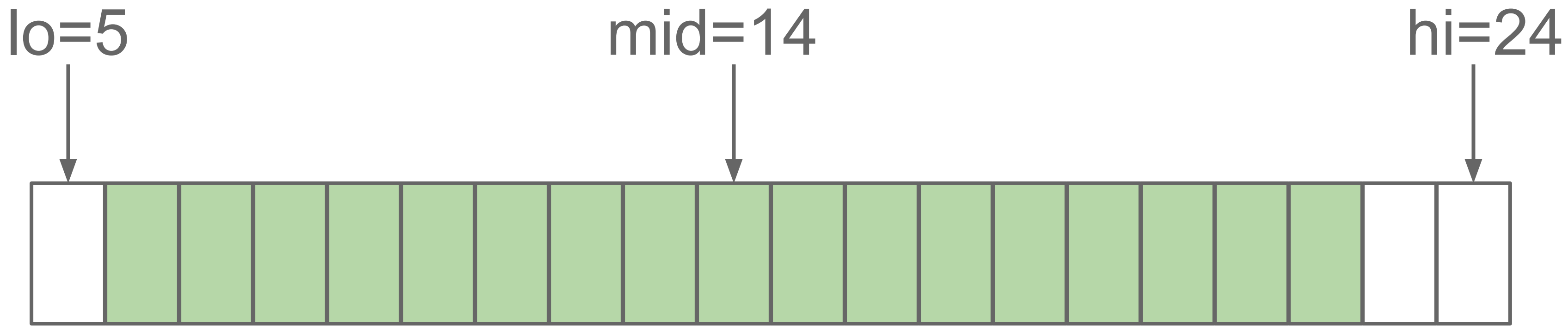
Left intervals: [12, 15]

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



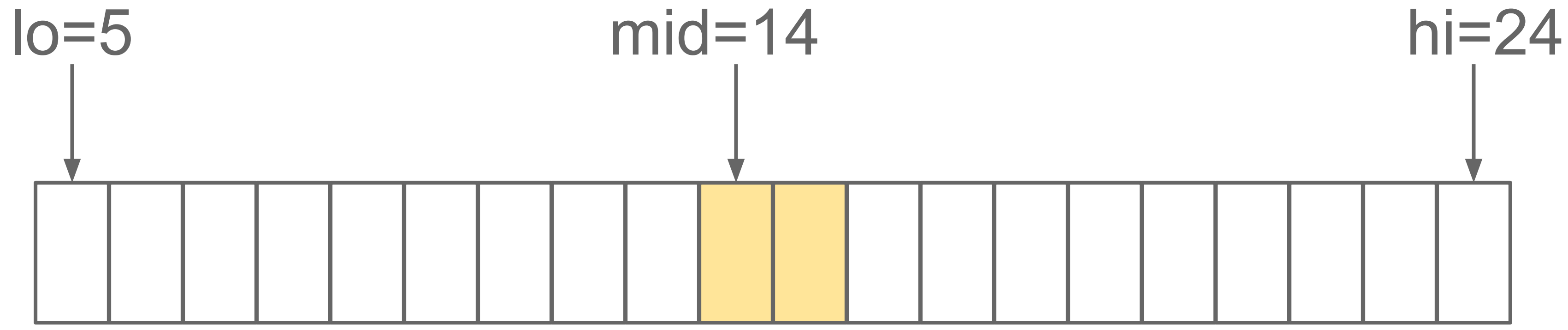
Left intervals: [12, 15], [8, 17]

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



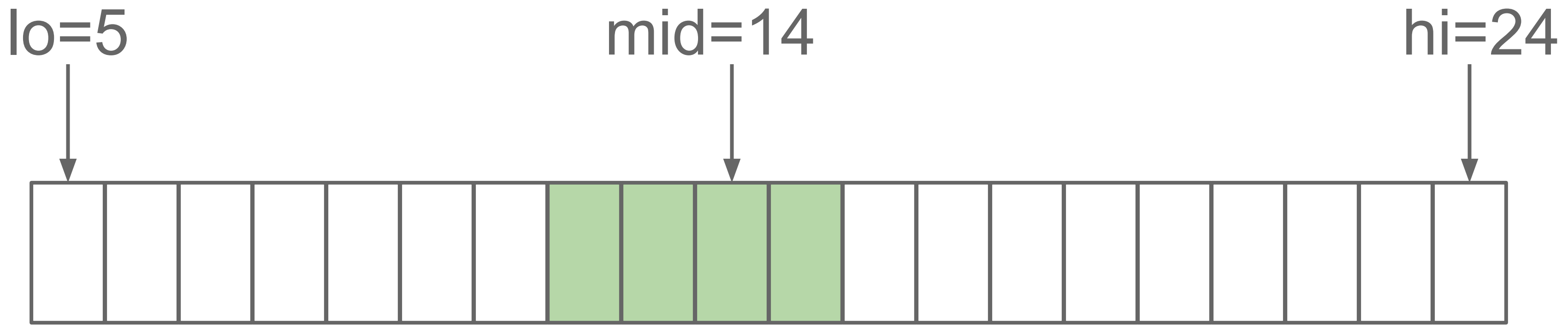
Left intervals: [12, 15], [8, 17], [6, 22]

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]

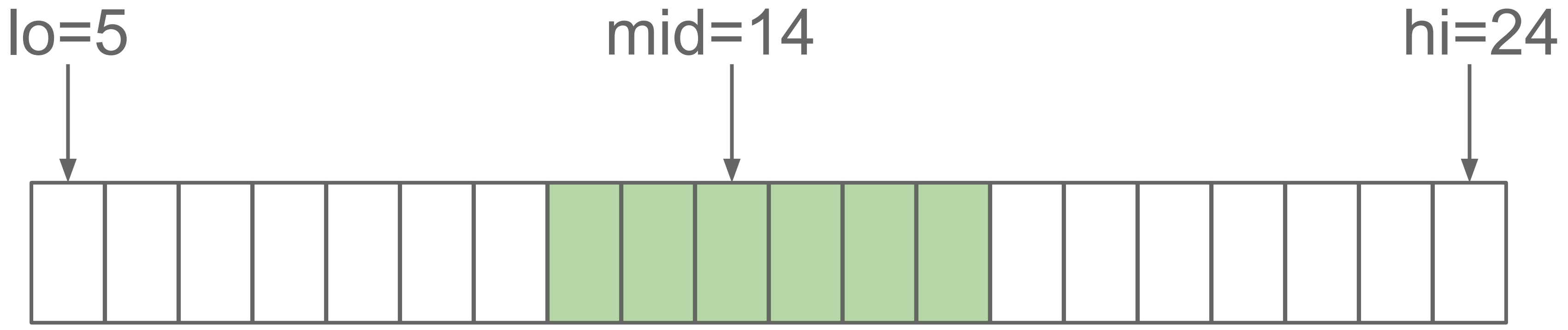
Again, starting from subsequence [mid, mid + 1], we expand it to the right, storing all intervals we encounter until we exit the [lo, hi] window.



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

Right intervals: [12, 15]

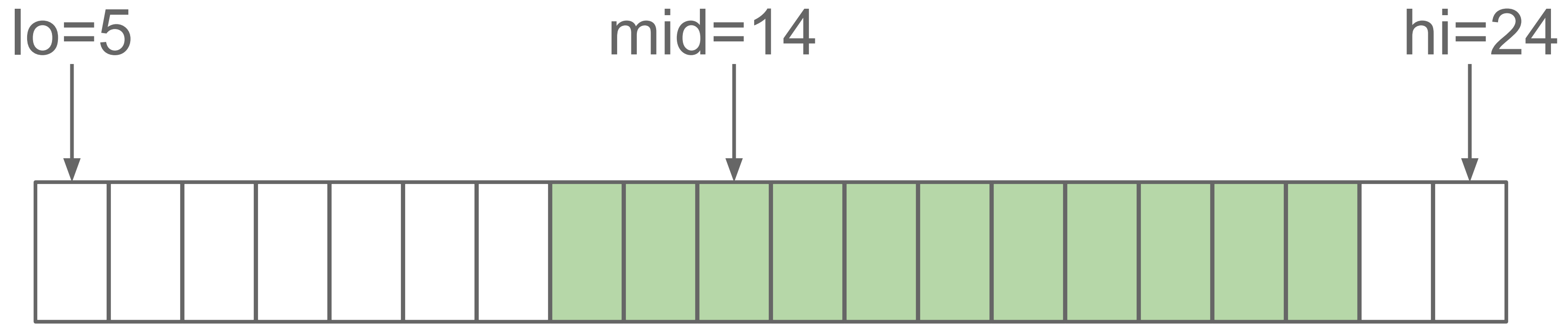
Again, starting from subsequence [mid, mid + 1], we expand it to the right, storing all intervals we encounter until we exit the [lo, hi] window.



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

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

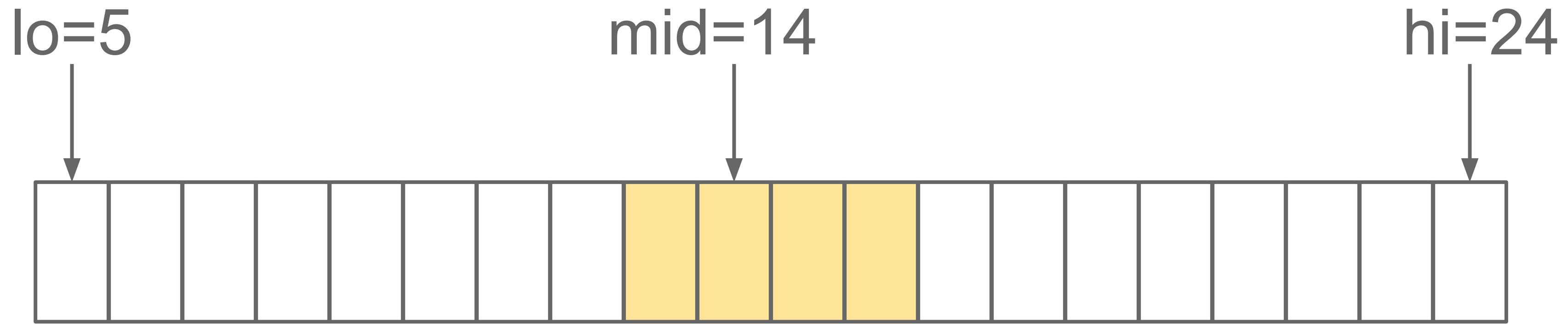
Again, starting from subsequence [mid, mid + 1], we expand it to the right, storing all intervals we encounter until we exit the [lo, hi] window.



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

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

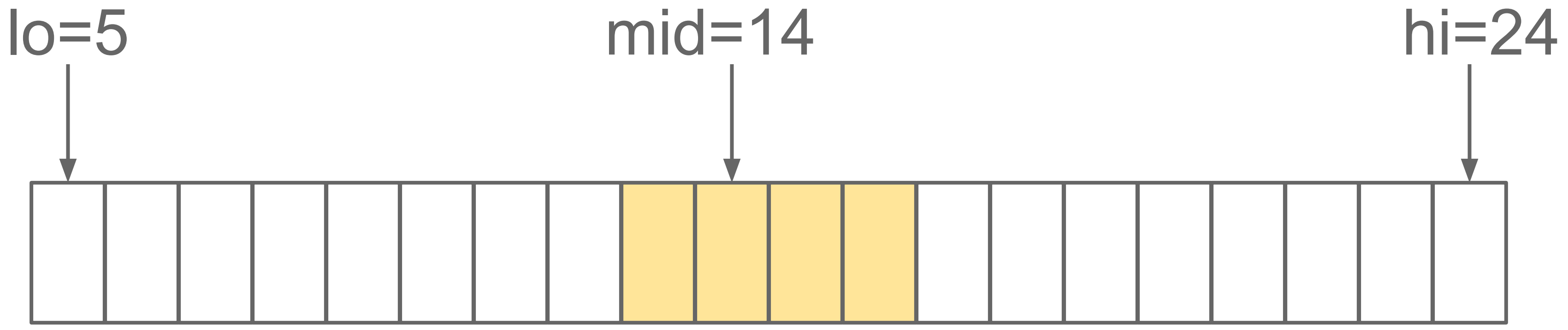
Again, starting from subsequence [mid, mid + 1], we expand it to the right, storing all intervals we encounter until we exit the [lo, hi] window.



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]

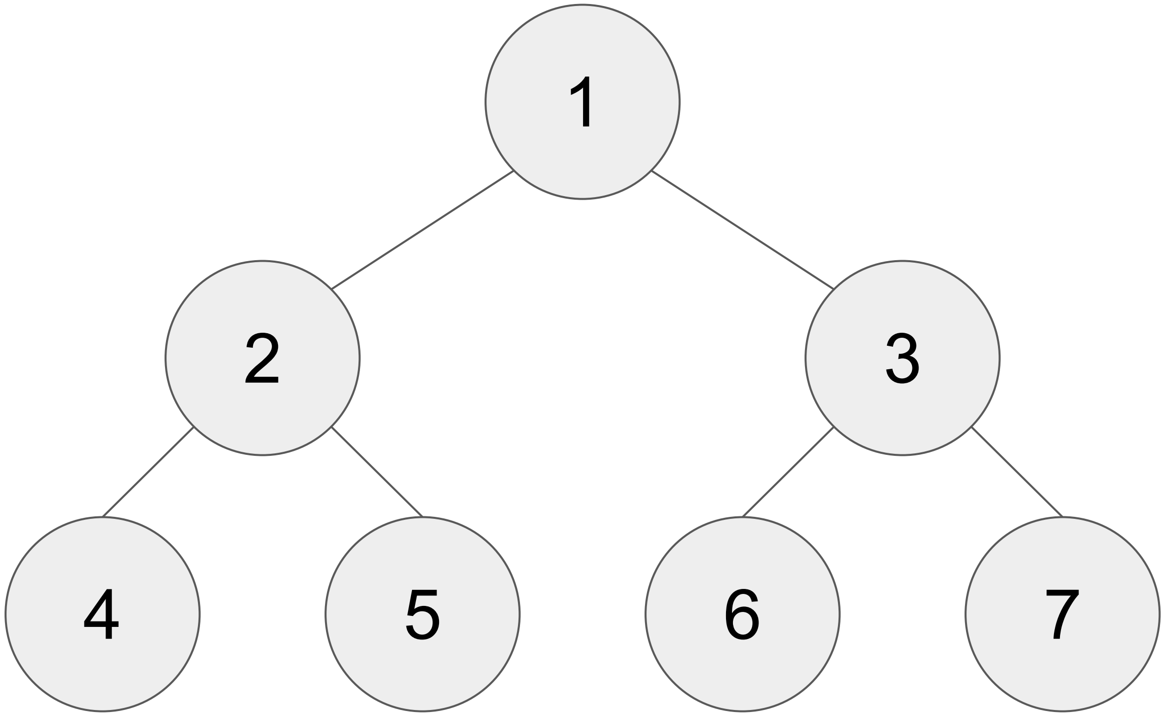
Right intervals: [12, 15], **[12, 17]**, [12, 22]

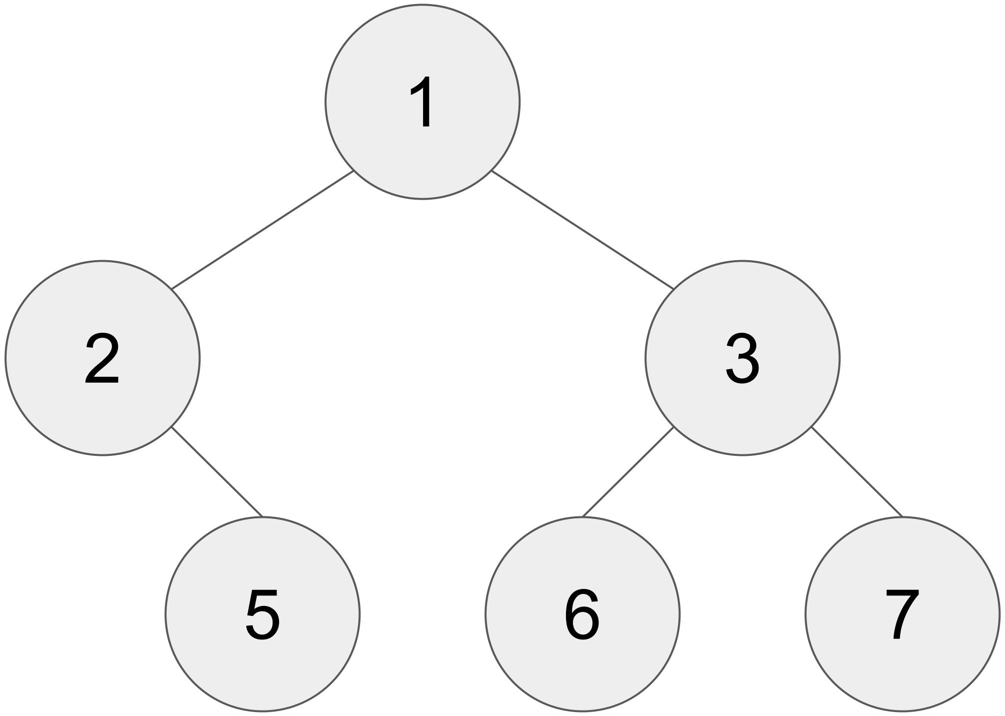
We can implement ImproveViaMid(queries, lo, mid, hi) in O(|hi - lo| + queries.size()), for overall complexity of O((N + Q) log N).

Problem C
Cumulative Code

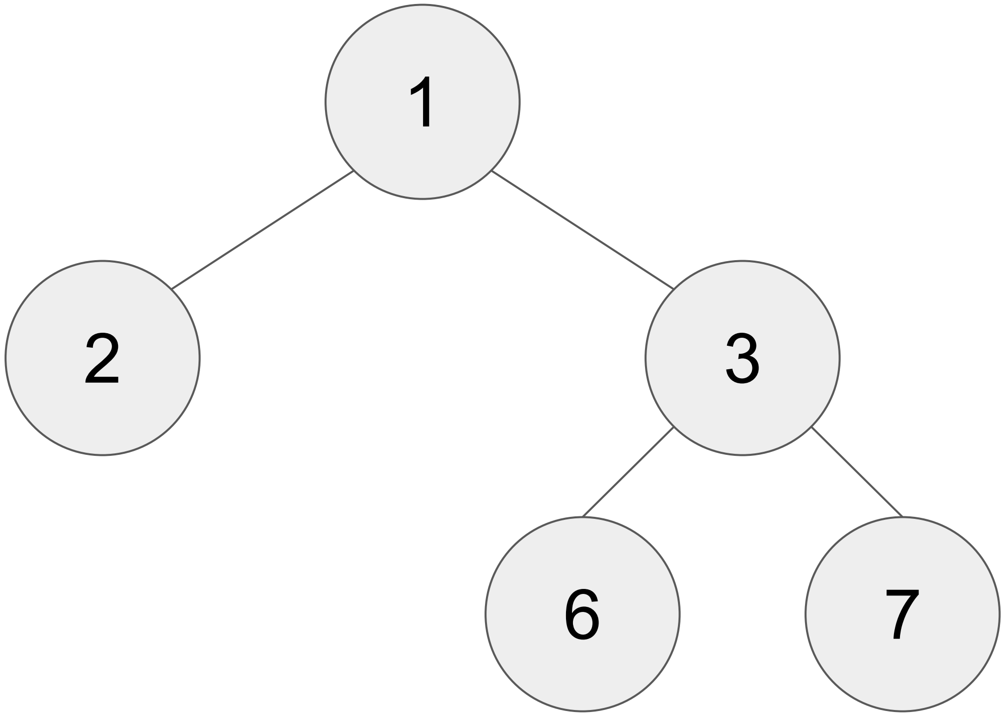
Submits: 2
Accepted: ?

Author: Ivan Paljak, Luka Kalinovčić

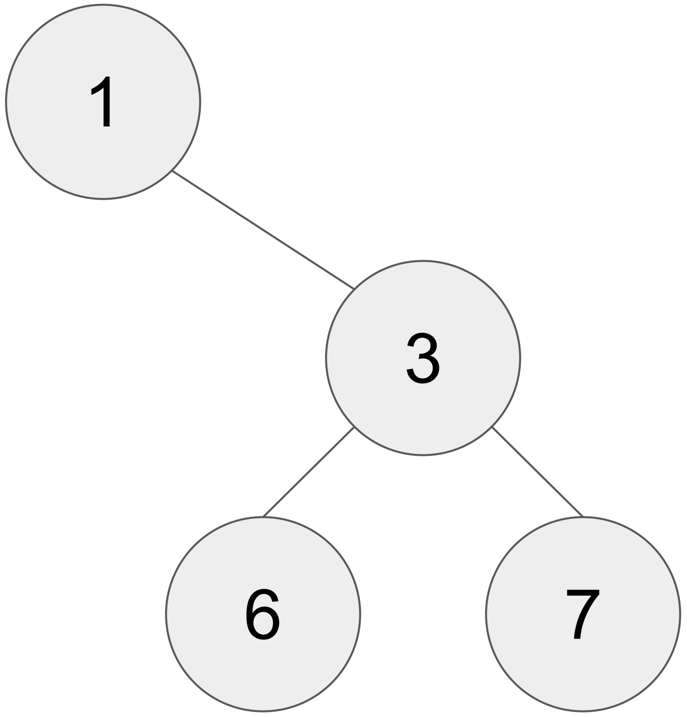

1
2
3
4
5
6
7



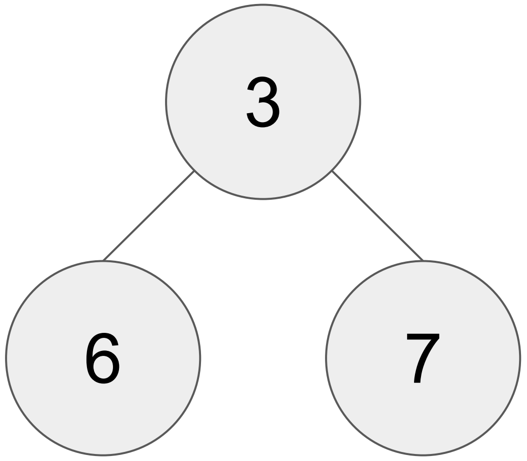
Code: 2



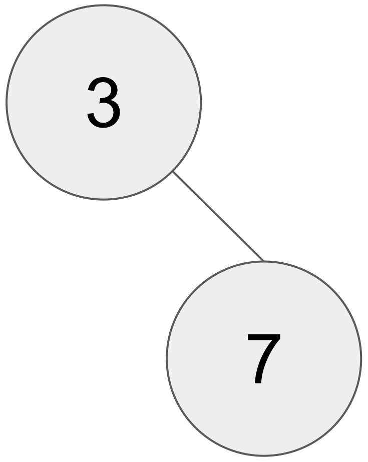
Code: 2 2



Code: 2 2 1

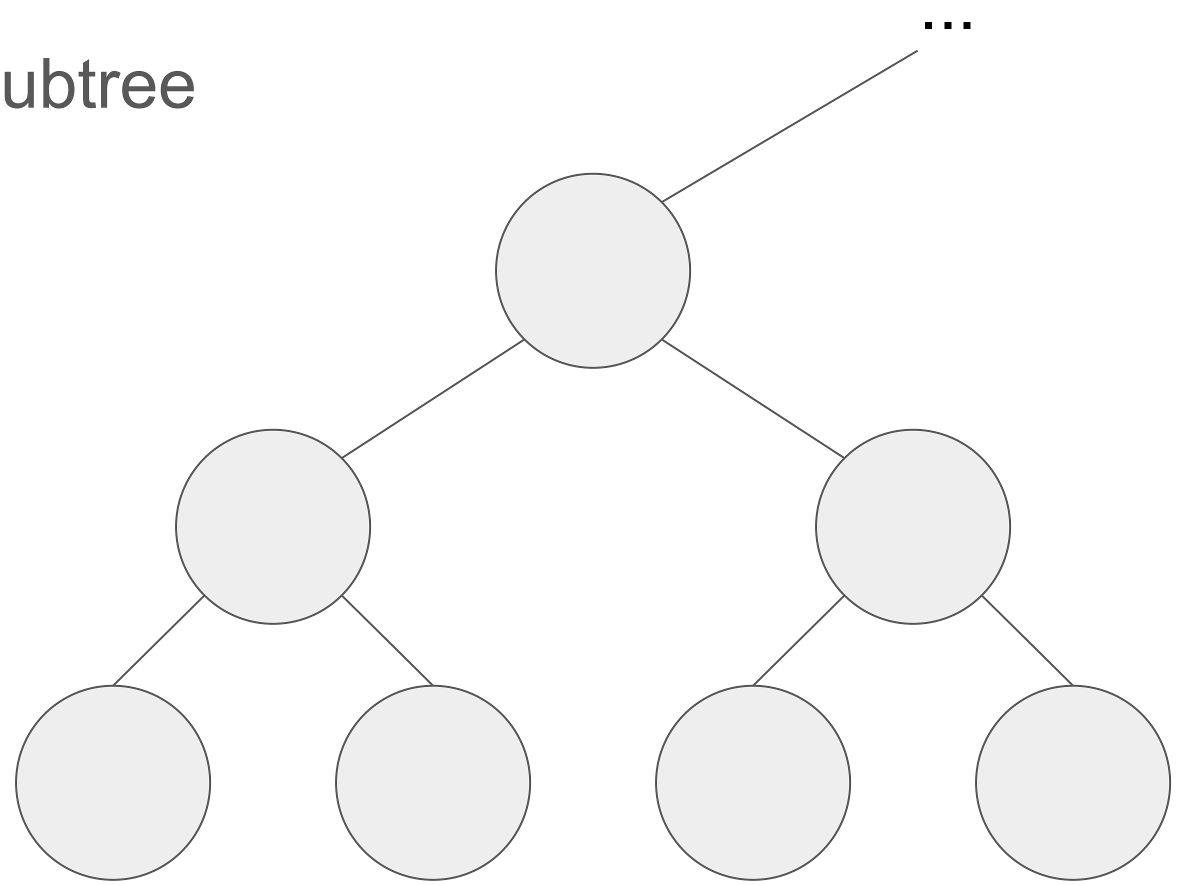


Code: 2 2 1 3



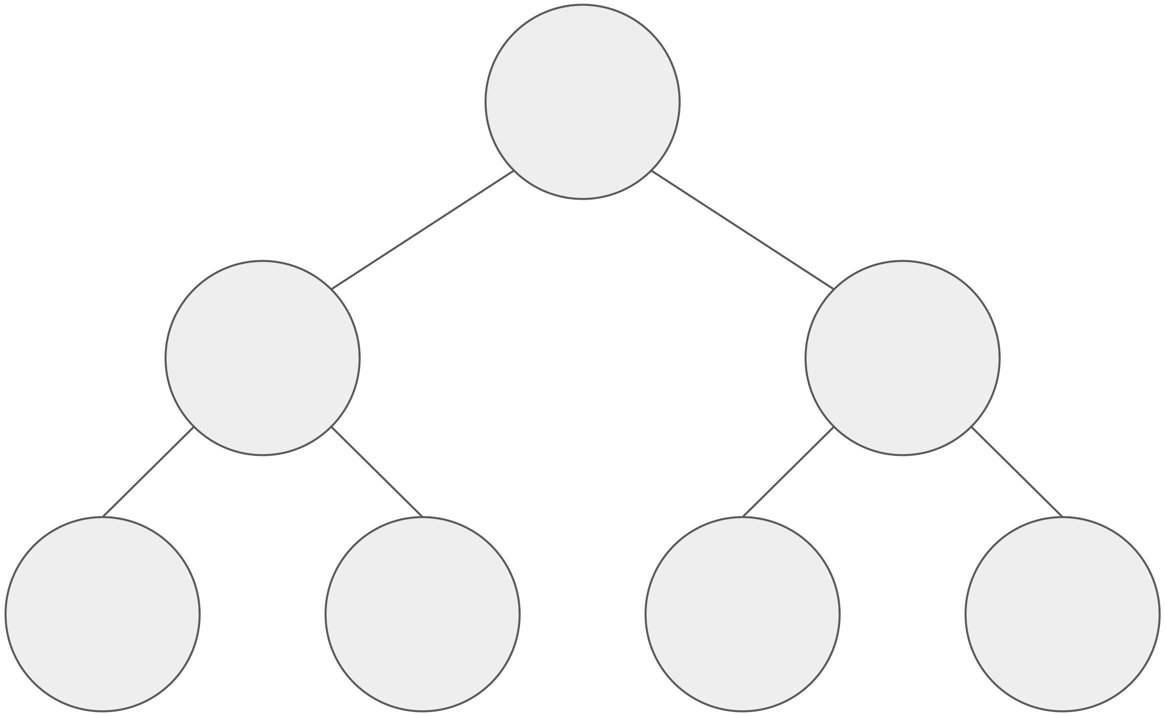
Code: 2 2 1 3 3

Type A subtree



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

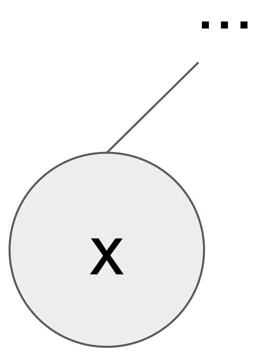
Type B subtree



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.

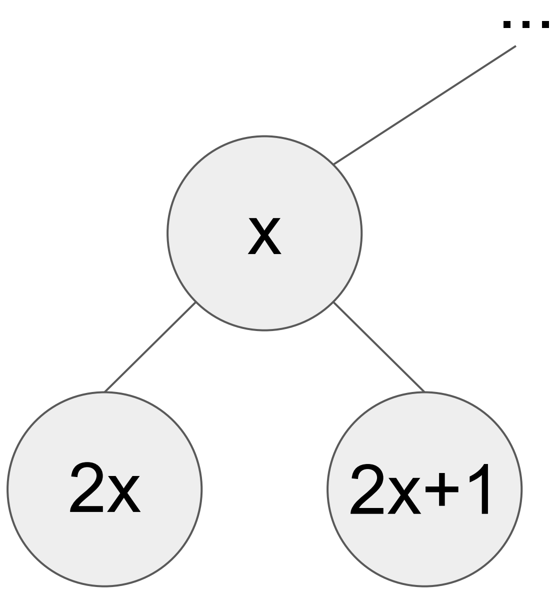


For k = 1, there is only a single node in the subtree.

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

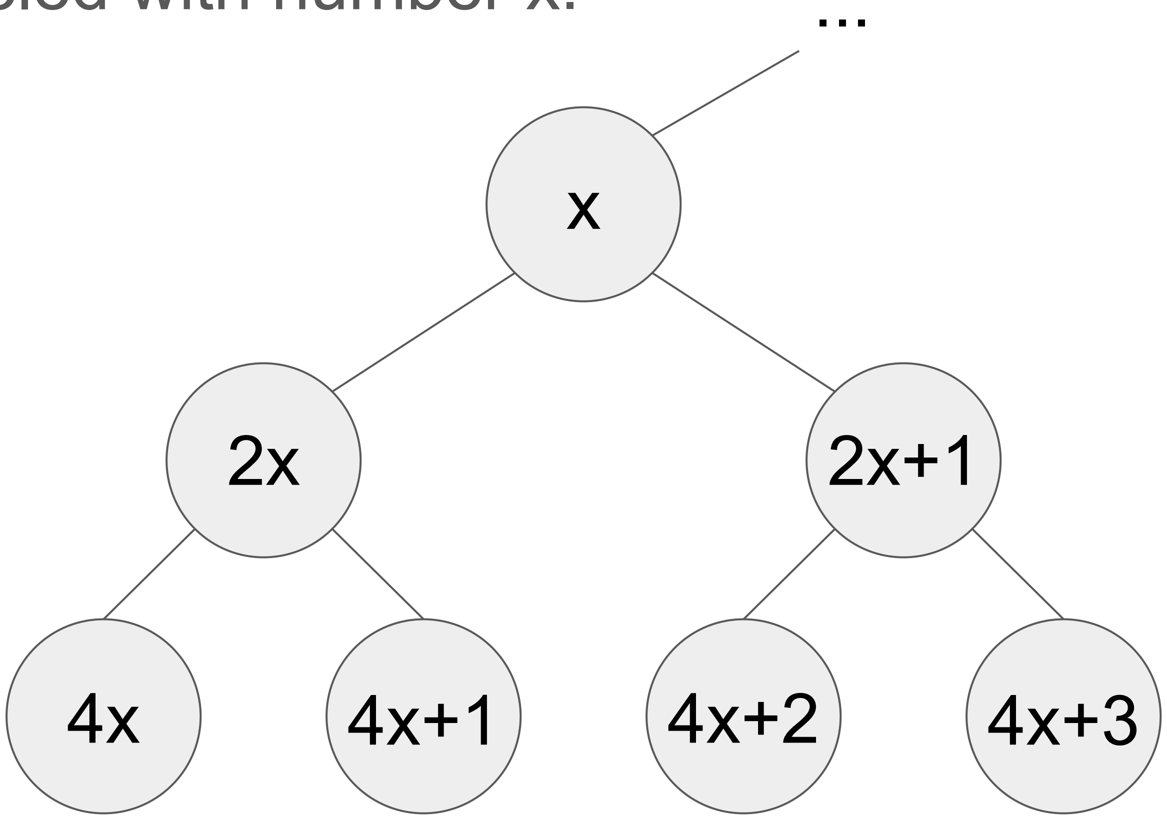
$f_x(1) = (x \text{ div } 2)$

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:

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

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

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

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

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

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 $\text{next}_Q(i)$ 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.

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

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:

- $k \leq K/2$ 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 $(k, \text{next}_Q(i) - i)$.

Because of 1), $\text{next}_Q(i) \leq i + 2^k - 1$, 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 $[a, a + a + (m - 1)\cdot d]$, 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.

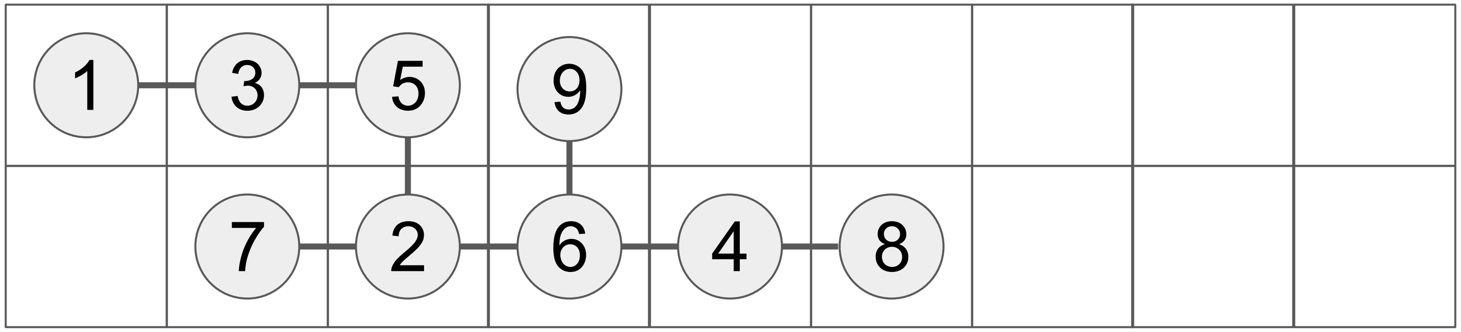
Problem E

Embedding Enumeration

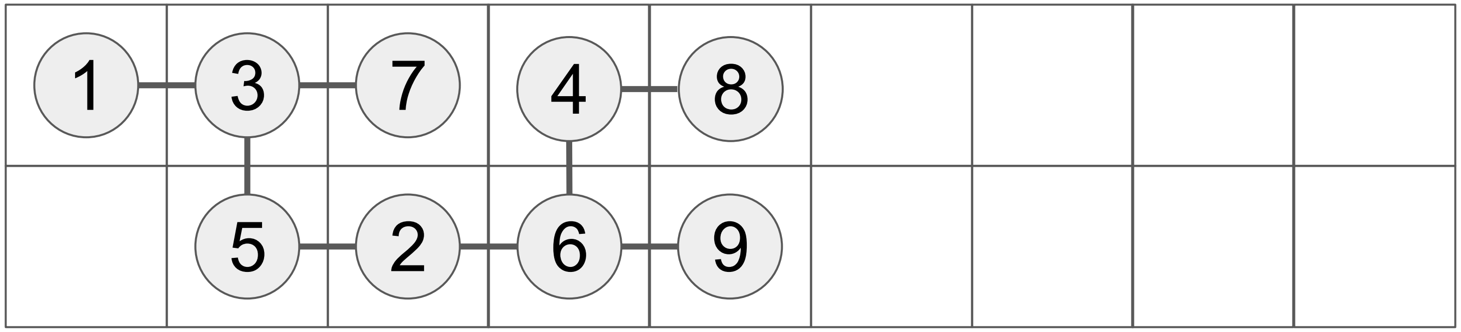
Submits: 1
Accepted: ?

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.



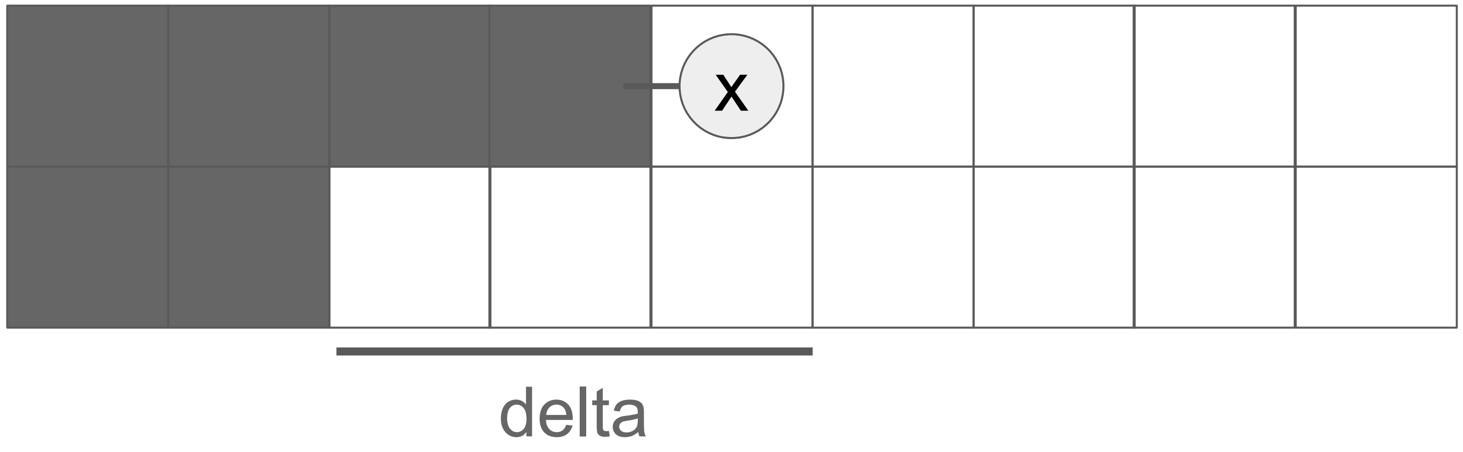
Problem: Given a tree, count the number of ways to embed it in a 2 by N grid, such that two nodes connected by an edge are adjacent in the grid. Node 1 has to be in top-left cell.



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).

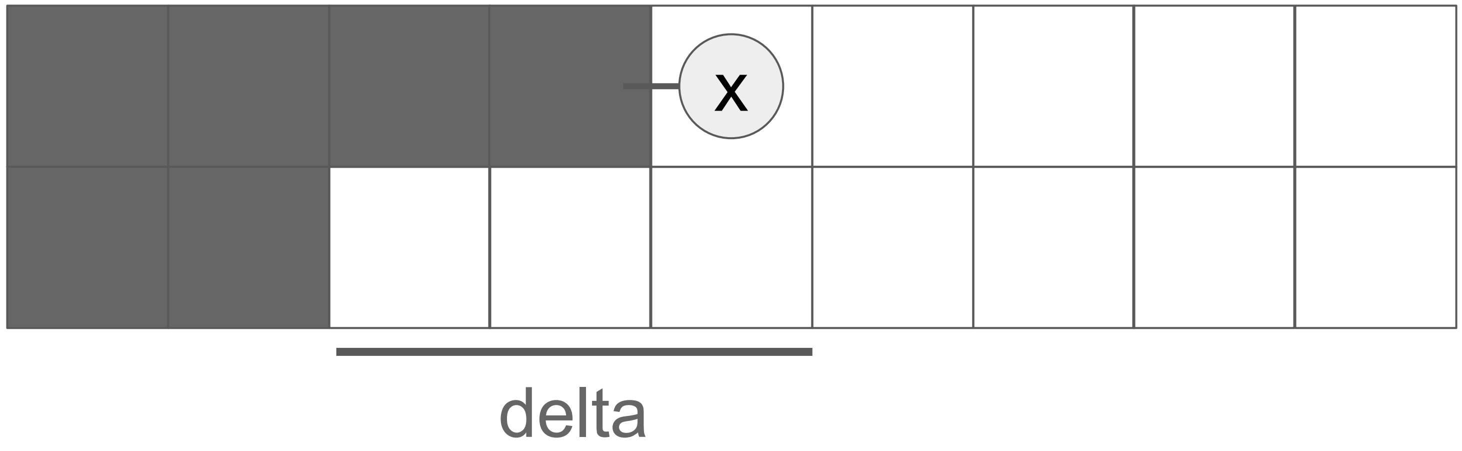
At this state, we have embedded all nodes except for those in x's subtree. Node x is embedded at the last cell of the longer of the two rows, and the delta is the difference in length between the two rows.



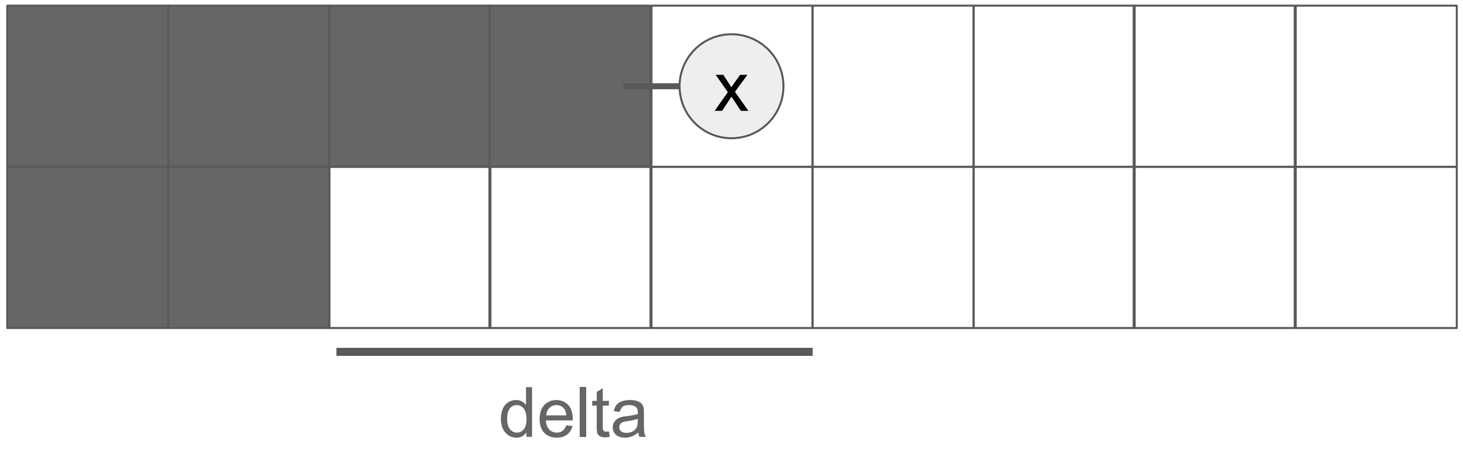
To make the transition, we'll try every possible assignment of node x's children to neighboring cells.

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

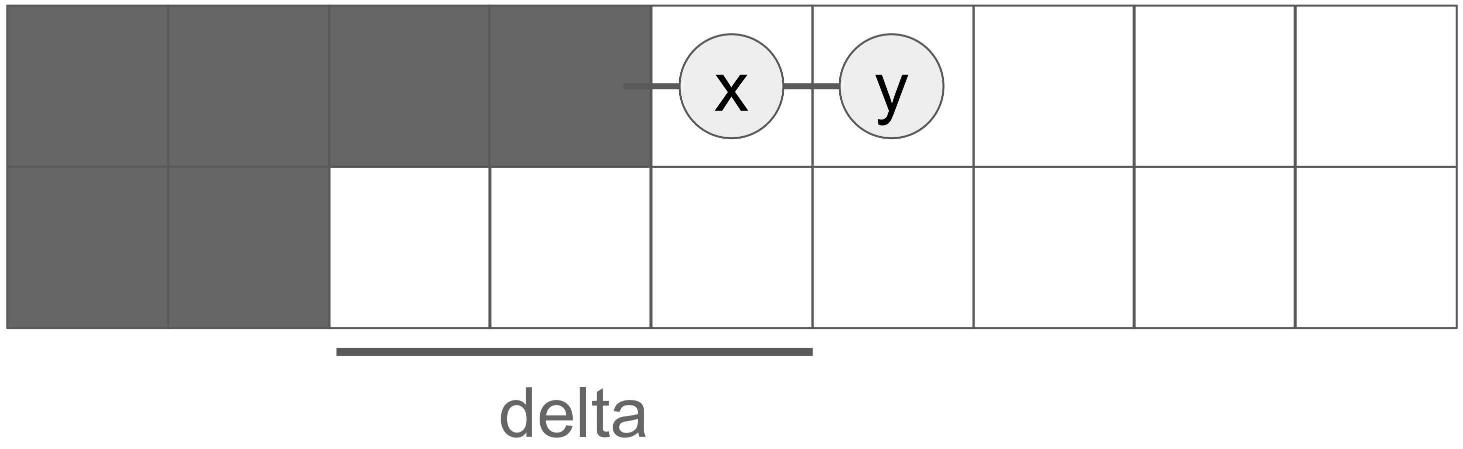
Let's analyze possible outcomes of such assignments.



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

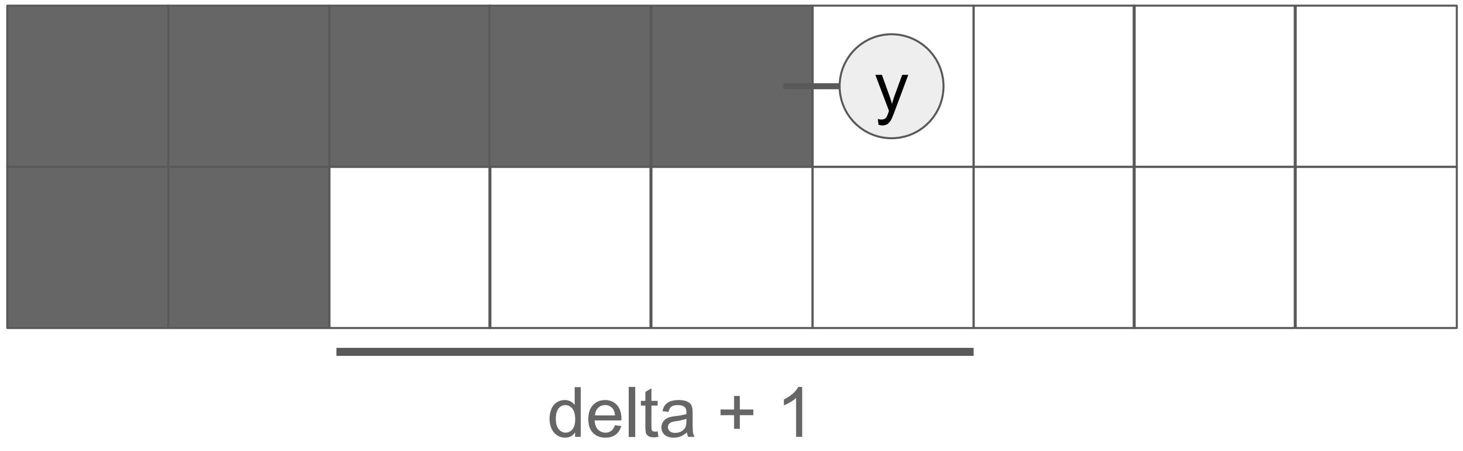


Node x has one child node y that was assigned to the right cell.

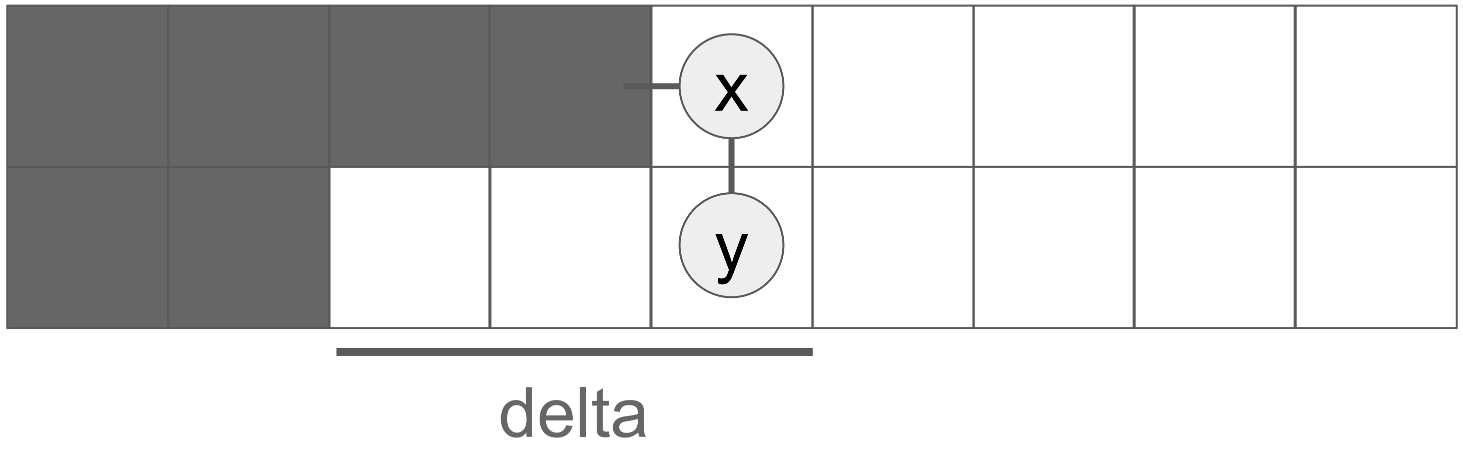


Node x has one child node y that was assigned to the right cell.

We transition to state (y, delta + 1)

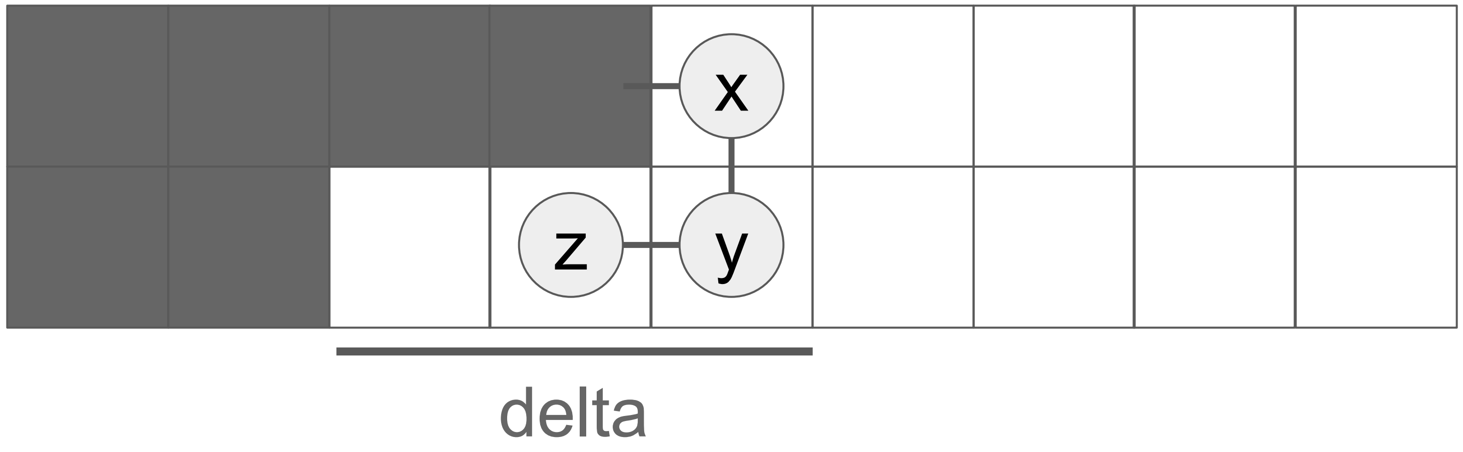


Node x has one child node y that was assigned to the bottom cell. We also assign y's children to neighboring cells.



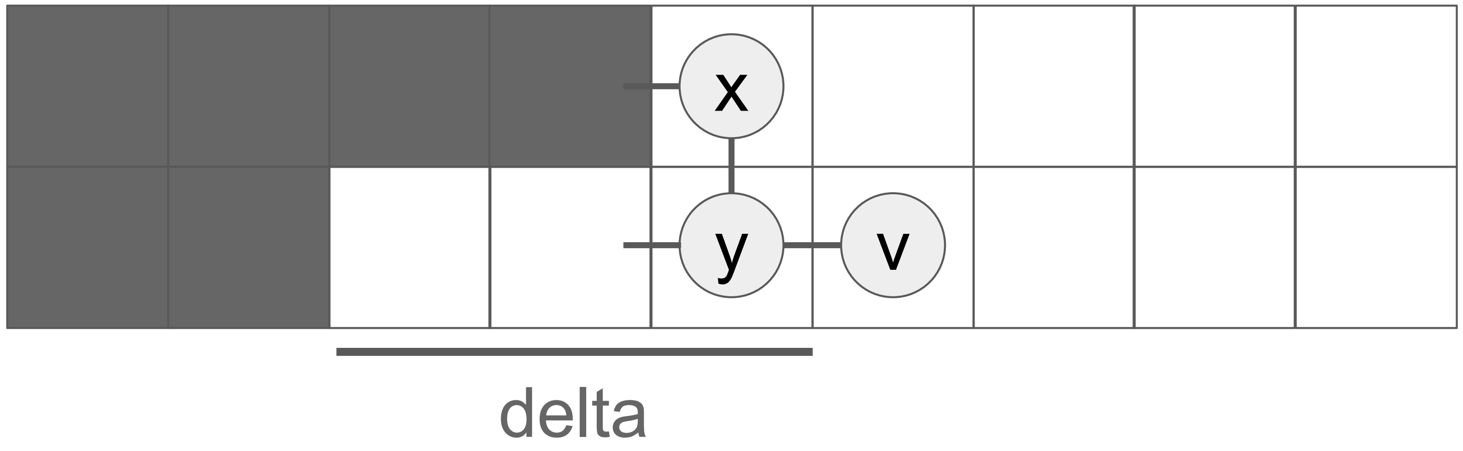
Node x has one child node y that was assigned to the bottom cell. We also assign y's children to neighboring cells.

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.



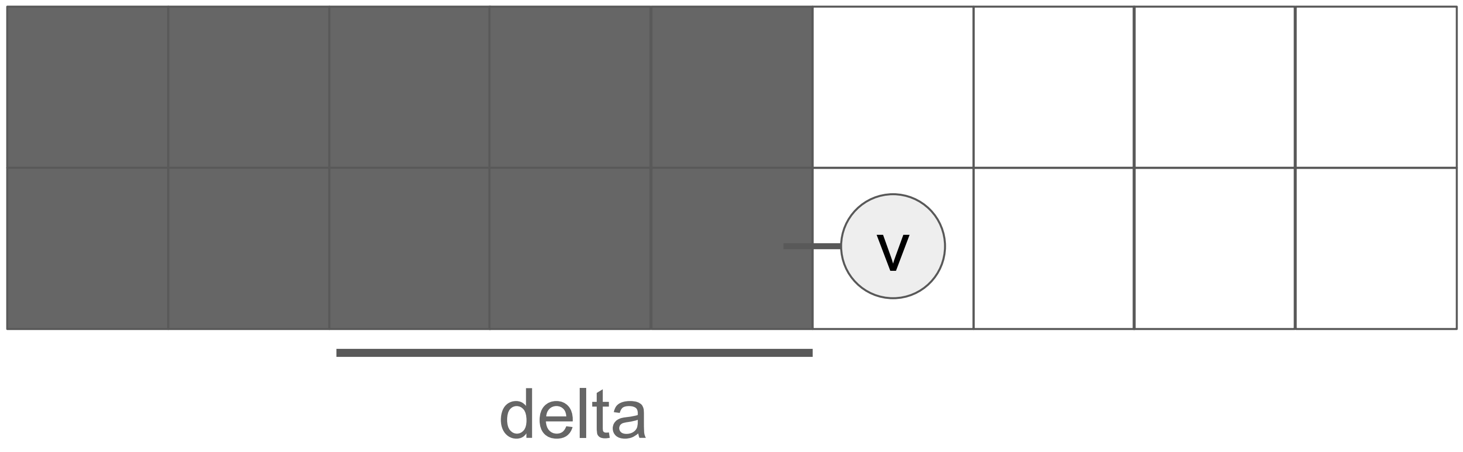
Node x has one child node y that was assigned to the bottom cell. We also assign y's children to neighboring cells.

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



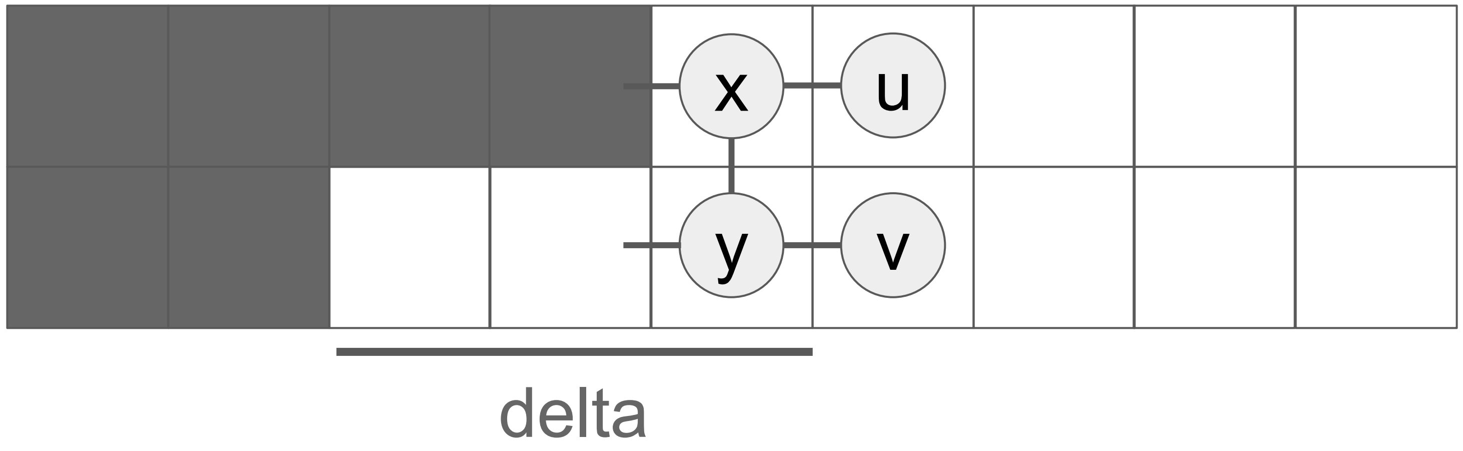
Node x has one child node y that was assigned to the bottom cell. We also assign y's children to neighboring cells.

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

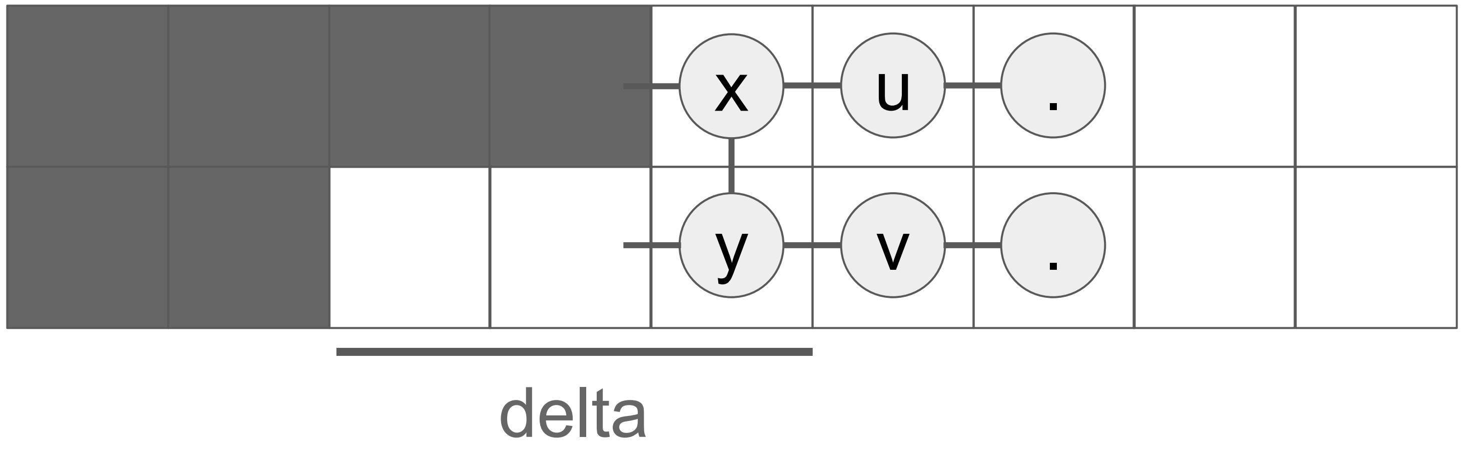


In the general case, x has two children y and u, and y has a child v assigned to the lower right cell.

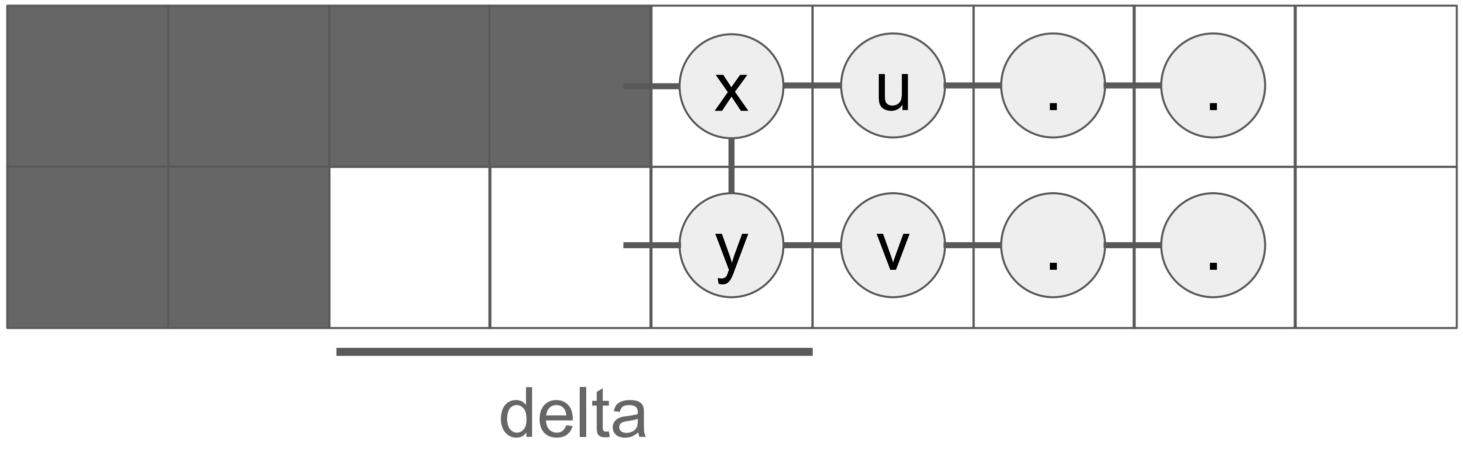
We now have two nodes, u and v, whose subtrees are not yet embedded, so we can't transition to any simple state just yet.



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).

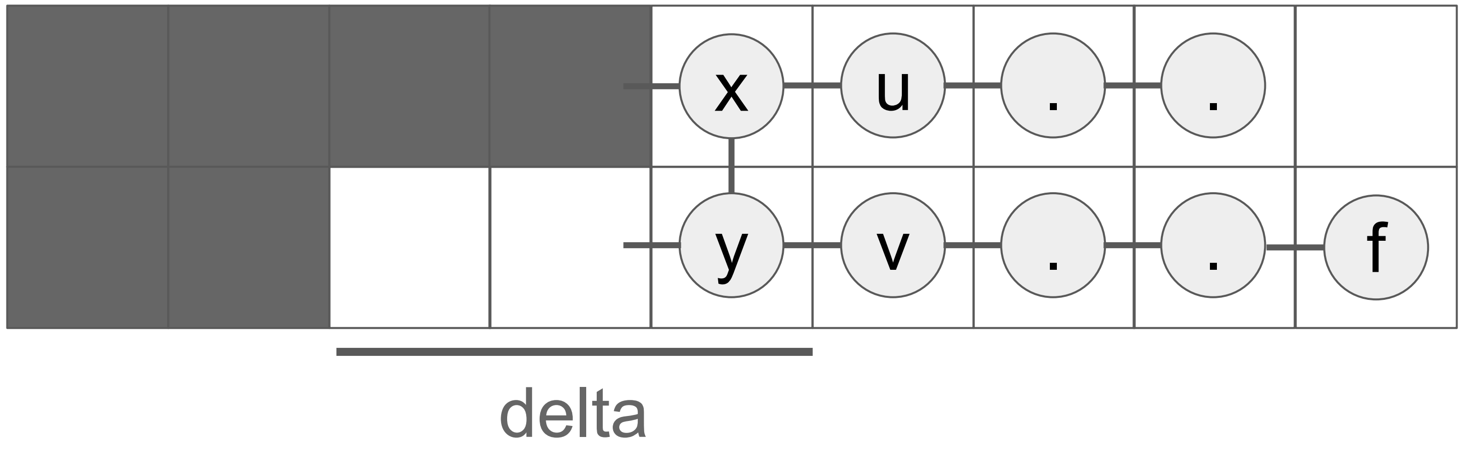


We keep appending children to the right until one of the chain runs out of nodes (or we encounter a node with two children which would make this assignment invalid).



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).

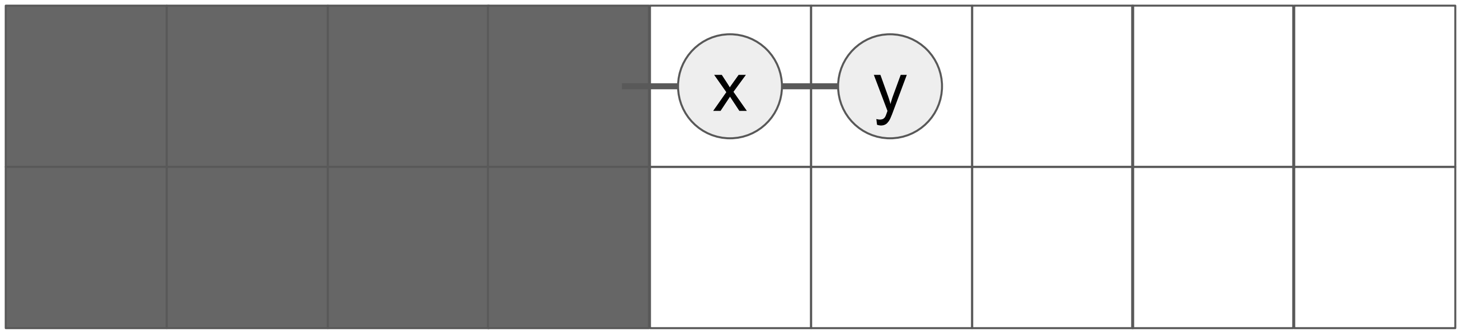
Once that happens, we can transition to state (f, 1).



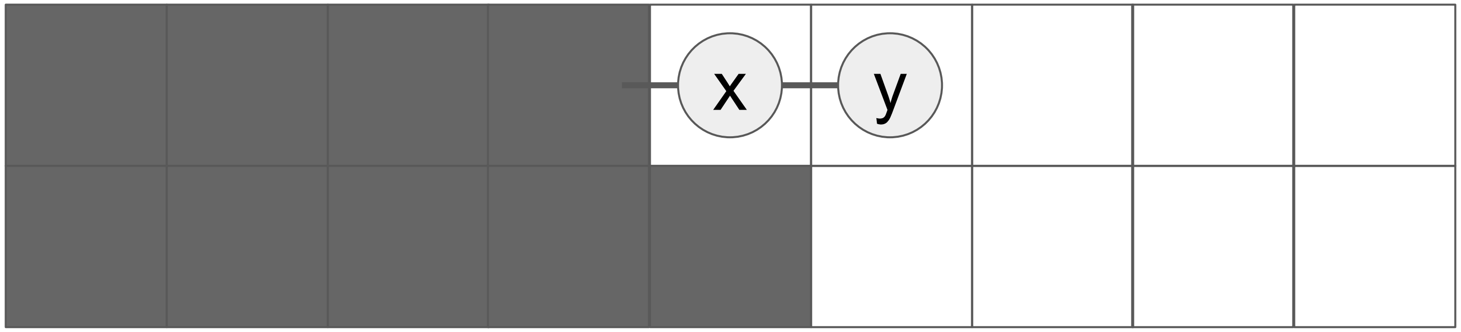
There are $O(N^2)$ states, and it's possible to implement all transitions in O(1) with some precomputation.

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

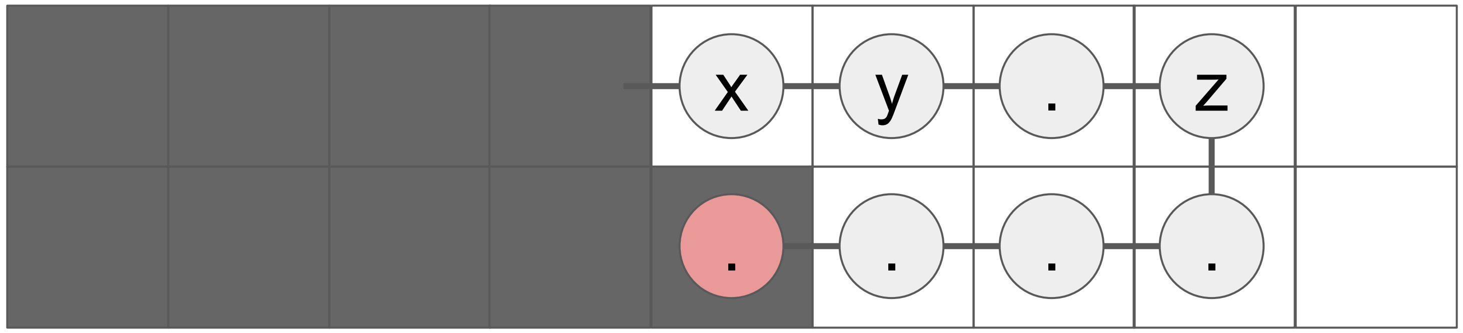
The only case where we actually increase the delta is the one where node x has one child assigned to the right cell.



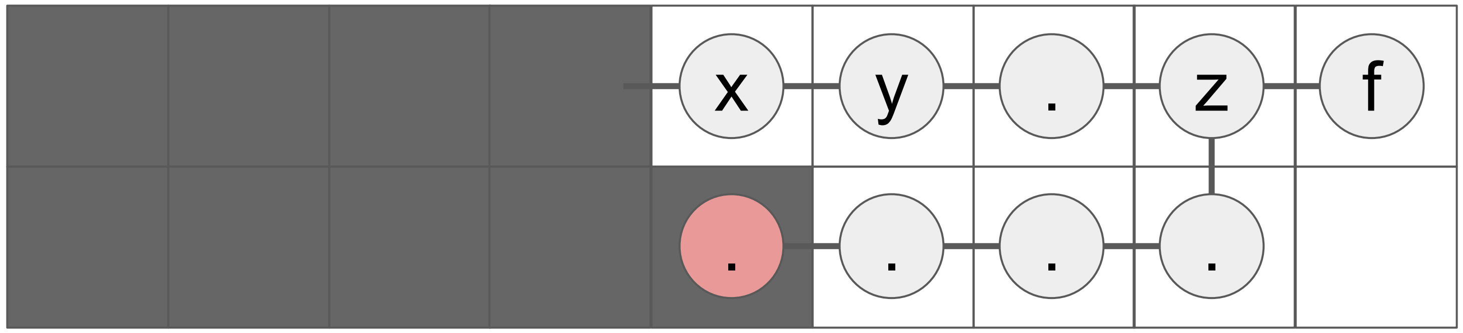
Originally we would transition to state (y, 2), but what kinds of embeddings would we miss if we transitioned to (y, 1) instead?



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

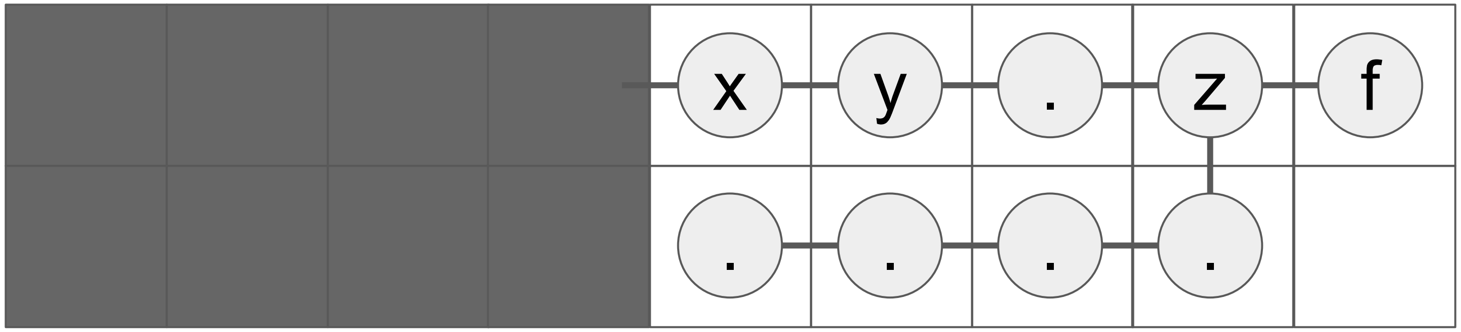


Originally we would transition to state (y, 2), but what kinds of embeddings would we miss if we transitioned to (y, 1) instead?



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!