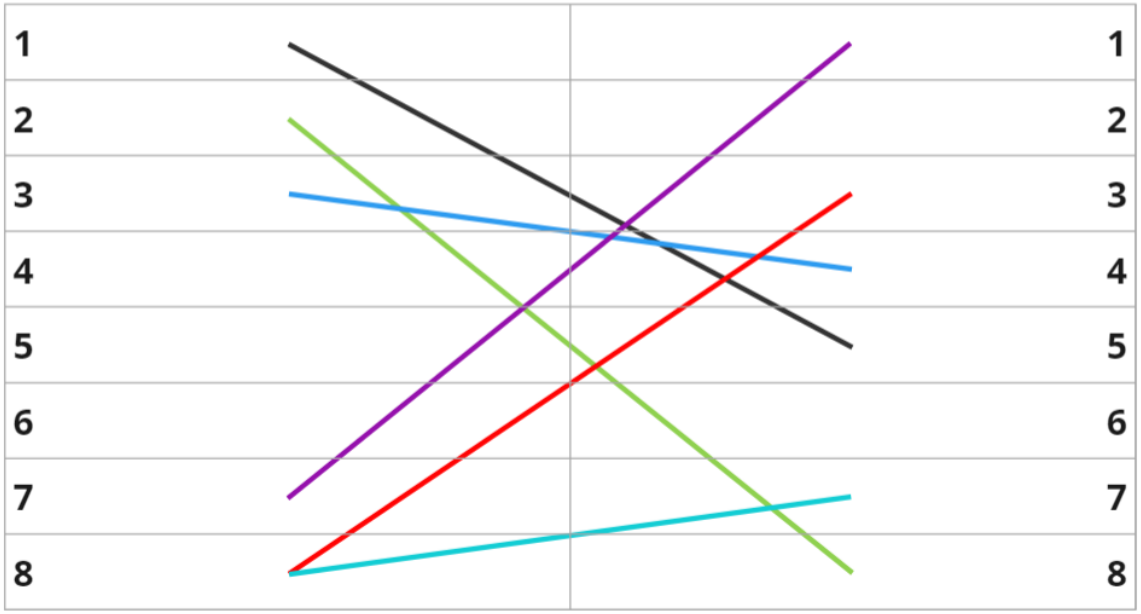
Structuri de Date

Seminar 6

Problema 1:

*Se dau **N** perechi de forma **(y_start, y_end)**, fiecare reprezentând un segment cu proprietatea că acesta începe la înălțimea **y_start** și se termină la înălțimea **y_end**, ca în figura de mai jos:*



Găsiți numărul de intersecții.

Soluție:

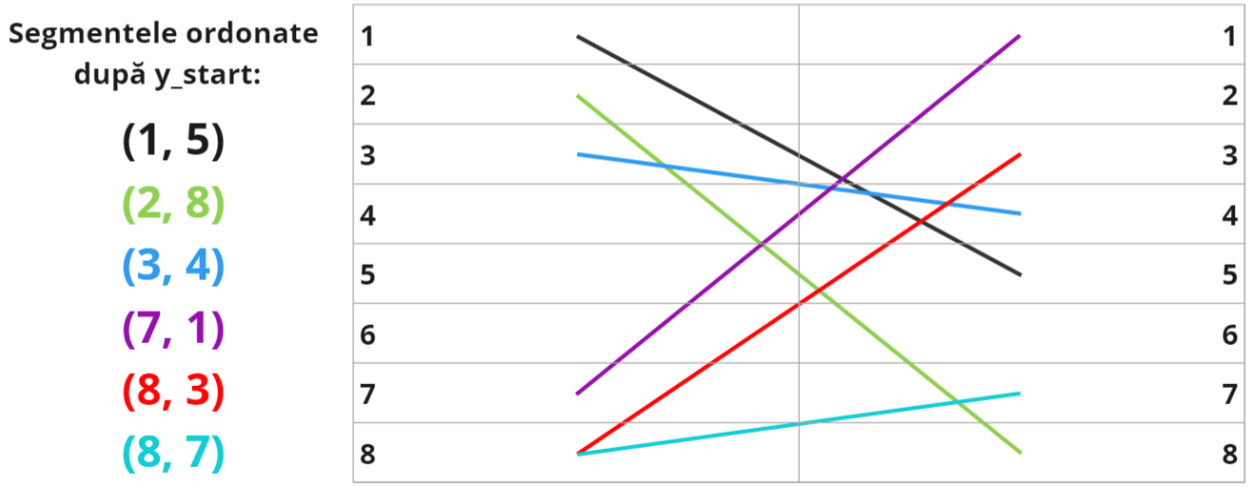
Începem prin a face observația că două segmente **(y1, y2)** și **(y3, y4)** se intersectează dacă și numai dacă **y1 <= y3 && y2 >= y4** (sau invers, dar am număra intersecția de două ori).

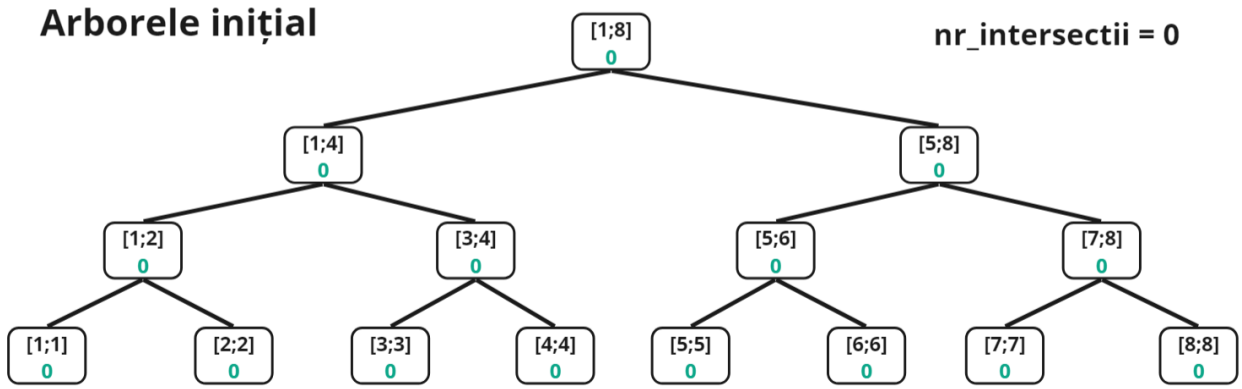
De aici, soluția intuitivă este să verificăm segmentele fiecare cu fiecare (două *for*-uri). **Complexitatea timp** pentru această abordare este **O(N²)**.

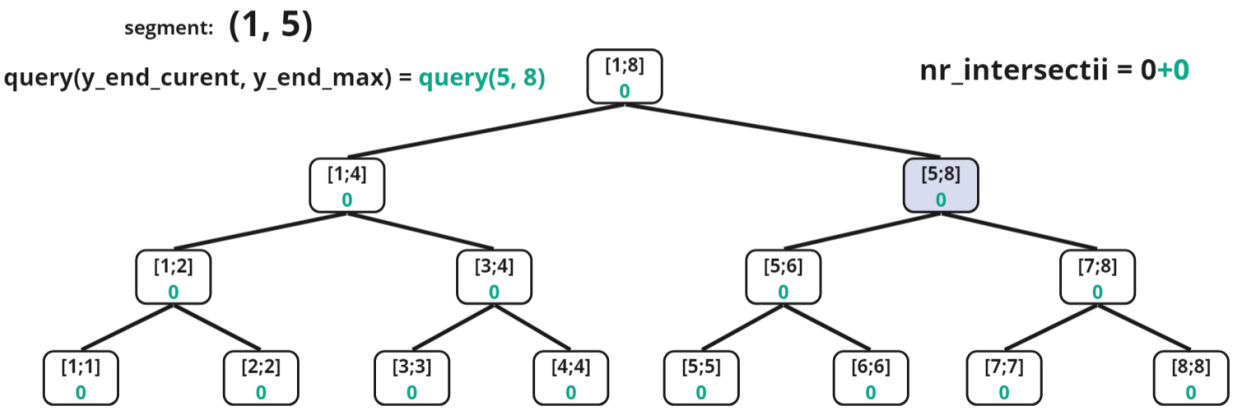
Alternativ, putem construi un arbore de intervale în funcție de **y_end**. Valorea dintr-un nod va fi numărul de segmente cu **y_end** în intervalul respectiv, inițial toate fiind 0. Următorul pas este să sortăm segmentele crescător după **y_start**. Acum, în ordinea sortării, pentru fiecare segment vom face pe arborele de intervale query-ul **(y_end_curent, y_end_maxim)**, unde **y_end_curent** semnifică **y_end** al segmentului

curent și **y_end_maxim** reprezintă cea mai mare valoare **y_end** din input. Adunăm rezultatul query-ului la numărul total de intersecții, și executăm un update pe arborele de intervale, și anume **(y_end_curent, +1)**, cu semnificația că adăugăm 1 la valoarea lui **y_end_curent**.

Toți acești pași sunt mult mai ușor de observat într-un **exemplu**:

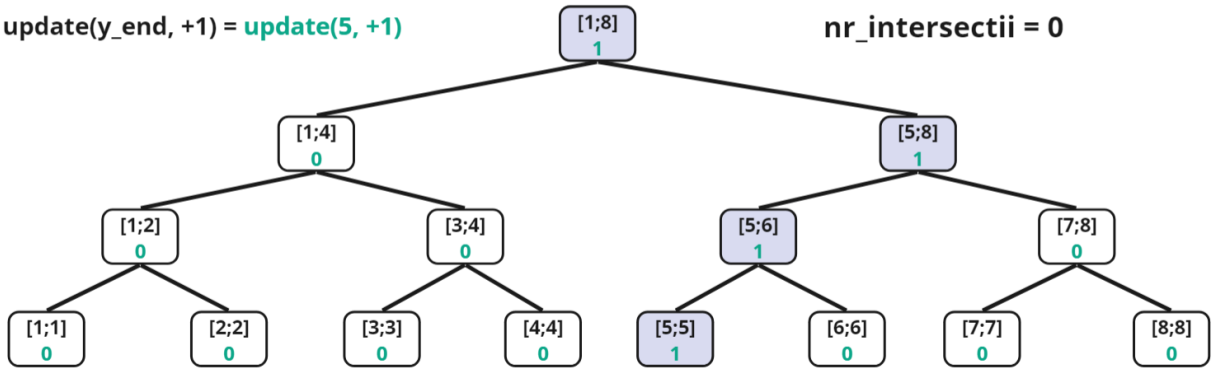






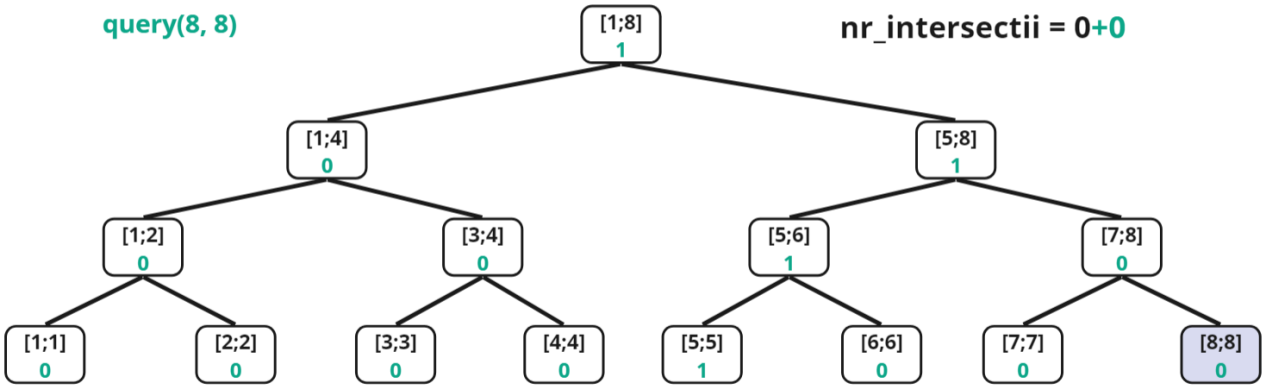
segment: **(1, 5)**

update(y_end, +1) = update(5, +1)



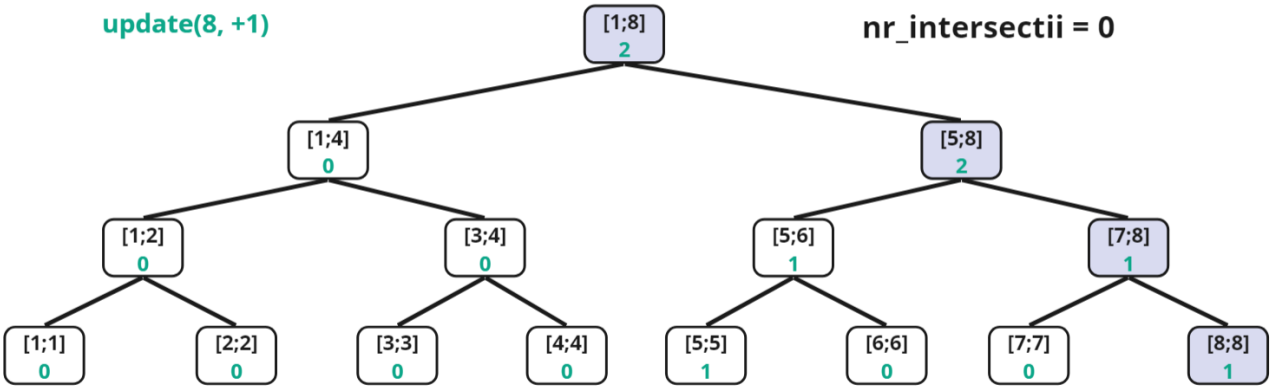
segment: **(2, 8)**

query(8, 8)



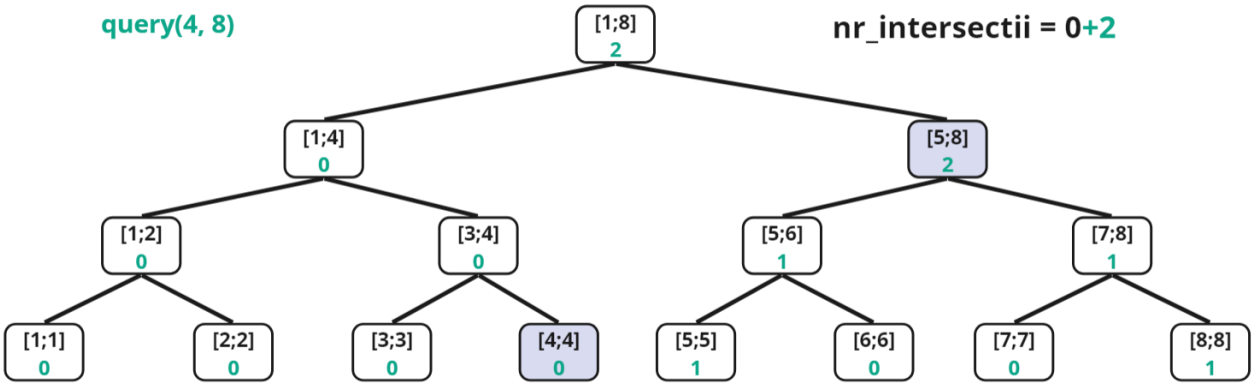
segment: **(2, 8)**

update(8, +1)



segment: **(3, 4)**

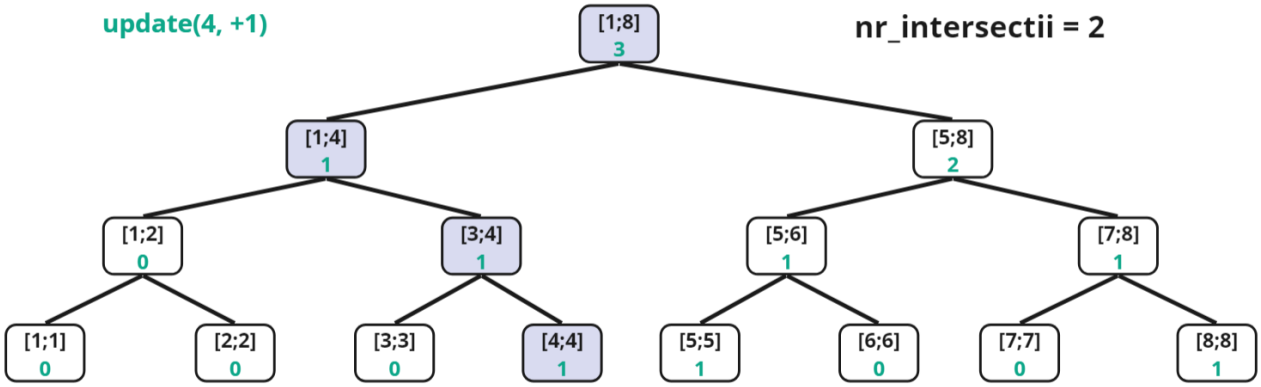
query(4, 8)



nr_intersectii = 0+2

segment: **(3, 4)**

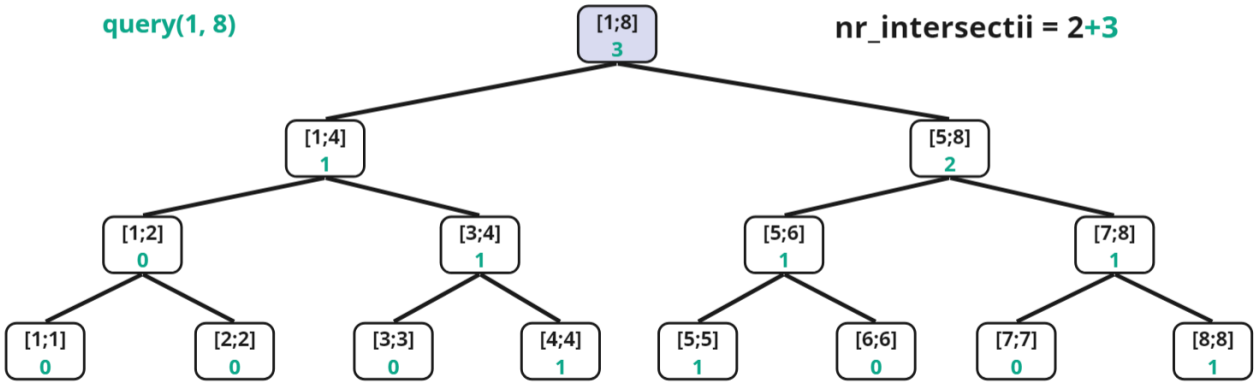
update(4, +1)



nr_intersectii = 2

segment: **(7, 1)**

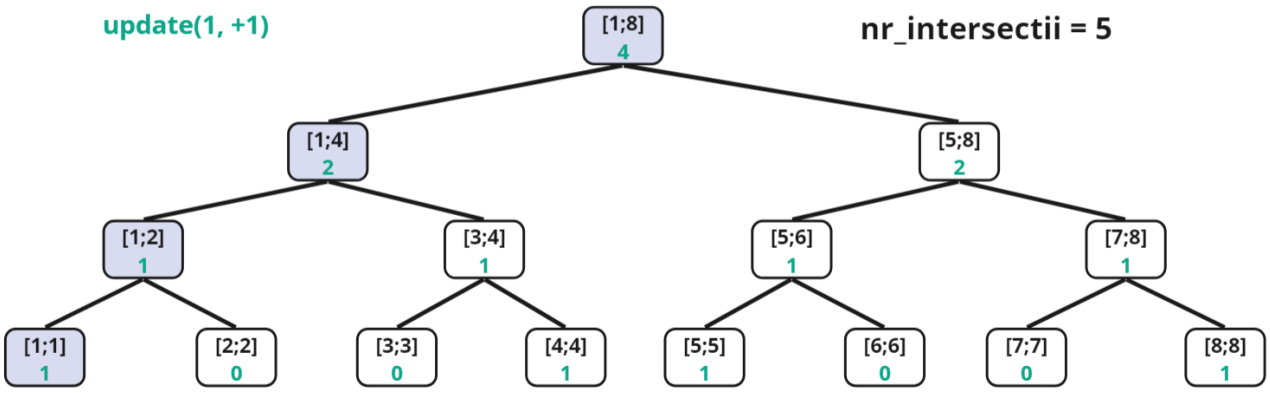
query(1, 8)



nr_intersectii = 2+3

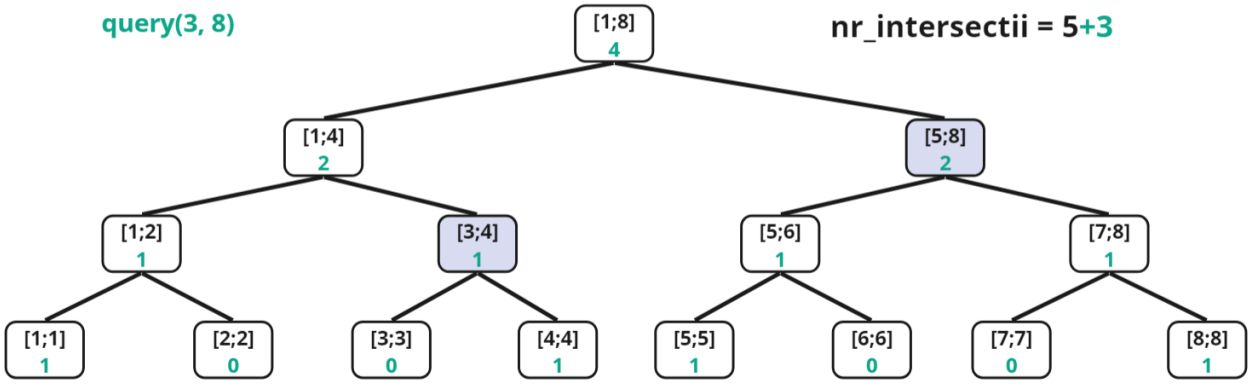
segment: **(7, 1)**

update(1, +1)



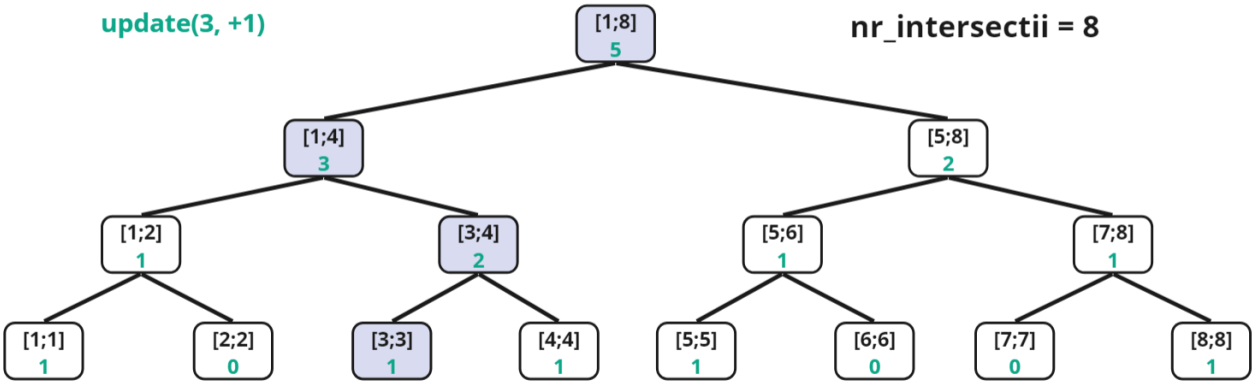
segment: **(8, 3)**

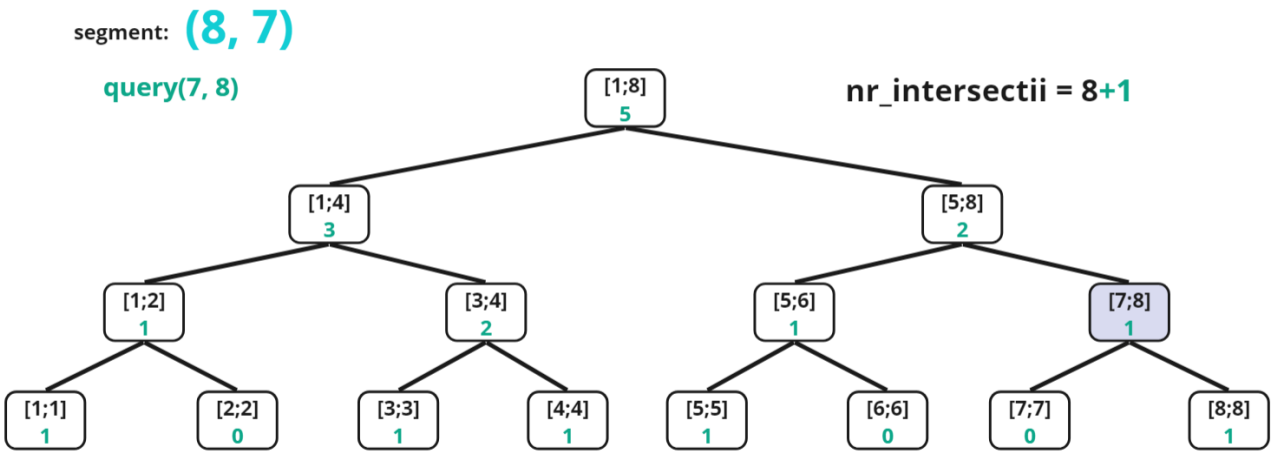
query(3, 8)

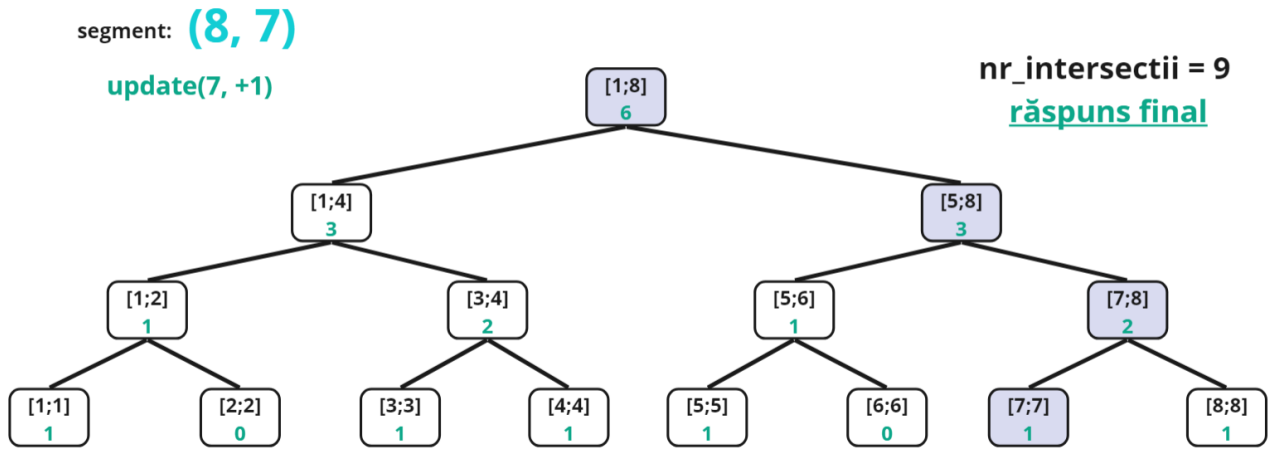


segment: **(8, 3)**

update(3, +1)







Explicație de ce funcționează metoda: luând segmentele în ordinea sortării, când ne uităm la un segment anume știm că arborele până în punctul respectiv reține informații doar de la segmentele care încep *mai sus* decât segmentul curent (prima condiție, **y1<=y3**). Pentru a doua condiție (**y2>=y4**), ne uităm la cele care se termină *mai jos* decât segmentul curent. Mai rămâne doar să actualizăm arborele, pentru ca segmentul curent să fie detectat de următoarele (cele care încep mai jos decât el).

Problema 2:

*Se dau **N** numere și se cere să se determine, pentru fiecare număr, câte numere mai mici decât el sunt în dreapta lui.*

Link problemă: https://leetcode.com/problems/count-of-smaller-numbers-after-self/description/

Soluţie:

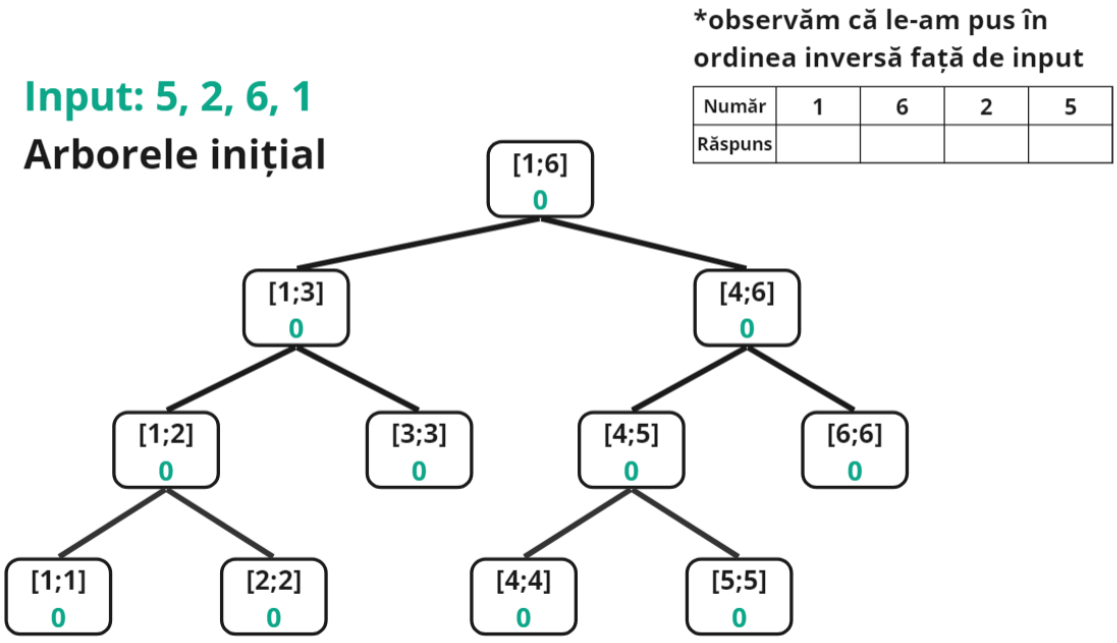
Foarte similar cu prima problemă! Putem privi un număr din vectorul **V** ca pe un punct **(i, V[i])**. În vectorul nostru avem deja aceste puncte ordonate crescător după **i**.

Vom construi un arbore de intervale în funcţie de valorile din **V** (iniţializat cu valori de 0 peste tot; un nod va reţine, pentru intervalul căruia îi corespunde, câte numere din intervalul respectiv s-au descoperit în vector până la pasul curent), şi vom vrea să parcurgem 'punctele' în ordine *descrescătoare* după **i** (practic, dinspre final spre început).

Pentru fiecare punct **(i, V[i])**, vom face pe arborele de intervale următoarele:

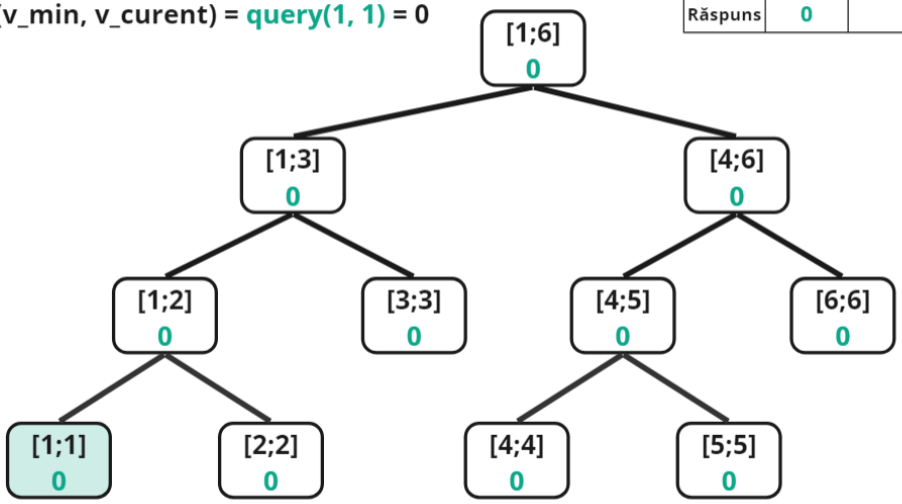
1. **Query(v_min, V[i])**: **v_min** este cea mai mică valoare din vector. Acest query va fi răspunsul de pe poziţia **i**, deoarece până în acest punct arborele reţine informaţii doar despre elementele de la dreapta lui **V[i]** (cele cu **i** mai mare decât cel curent, care au fost deja procesate).
2. **Update(V[i], +1)**: adăugăm 1 ca să contorizăm descoperirea numărului respectiv.

La fel ca la problema anterioară, un exemplu va clarifica uşor lucrurile:



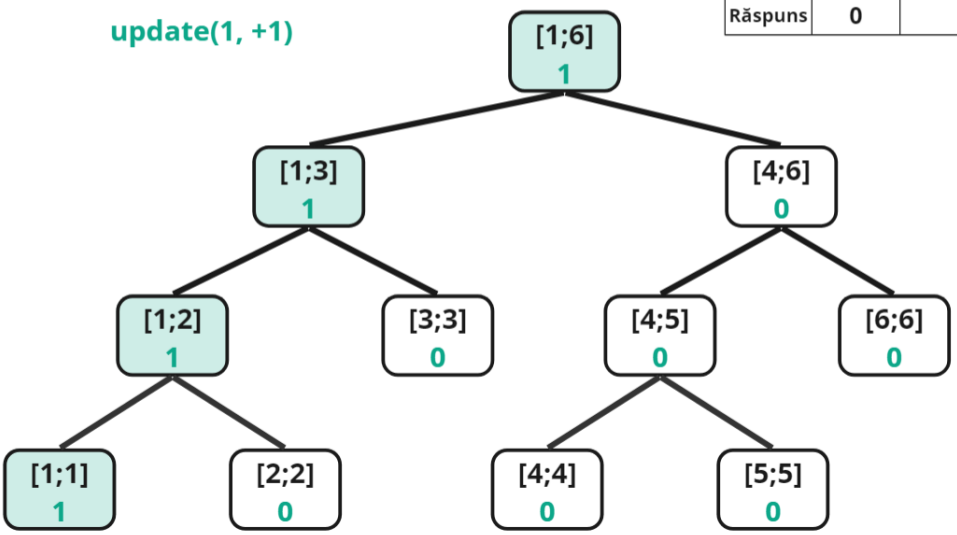
Număr	1	6	2	5
Răspuns	0			

query(v_min, v_curent) = query(1, 1) = 0



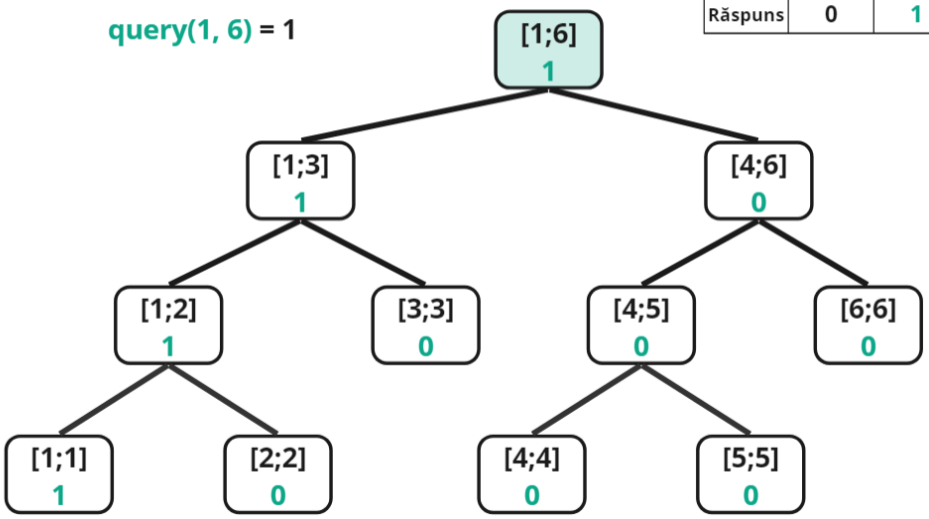
Număr	1	6	2	5
Răspuns	0			

update(1, +1)



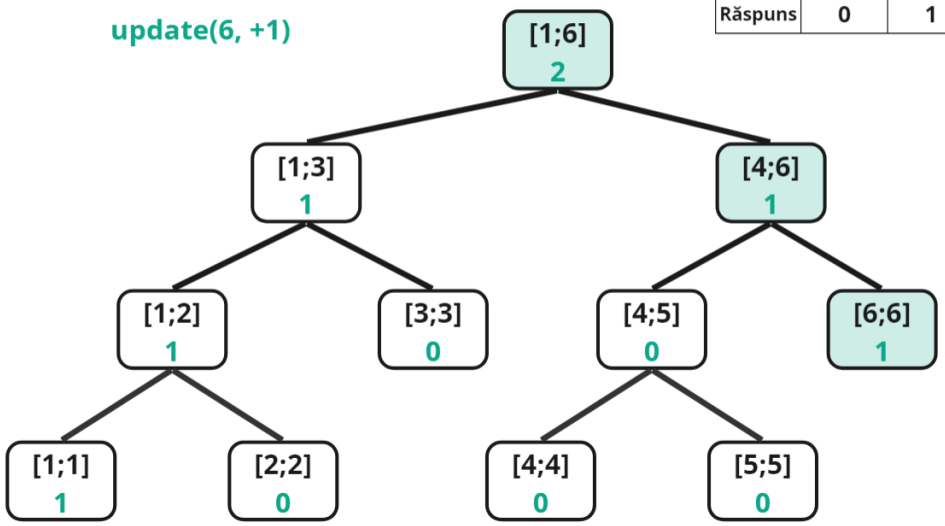
Număr	1	6	2	5
Răspuns	0	1		

query(1, 6) = 1



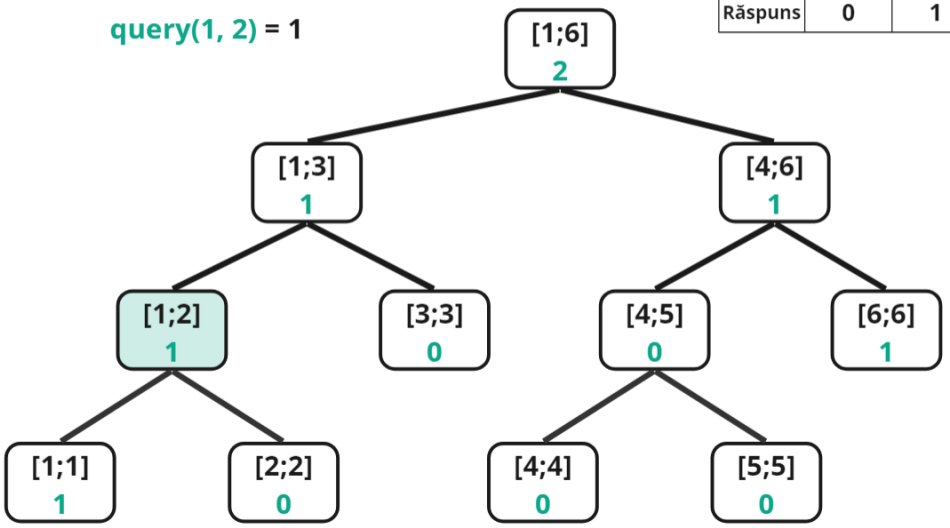
Număr	1	6	2	5
Răspuns	0	1		

update(6, +1)



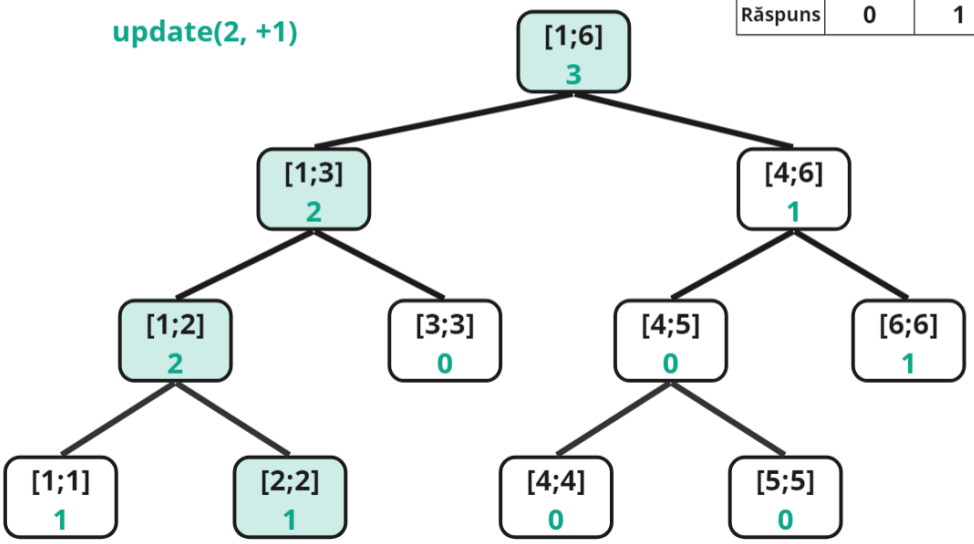
Număr	1	6	2	5
Răspuns	0	1	1	

query(1, 2) = 1



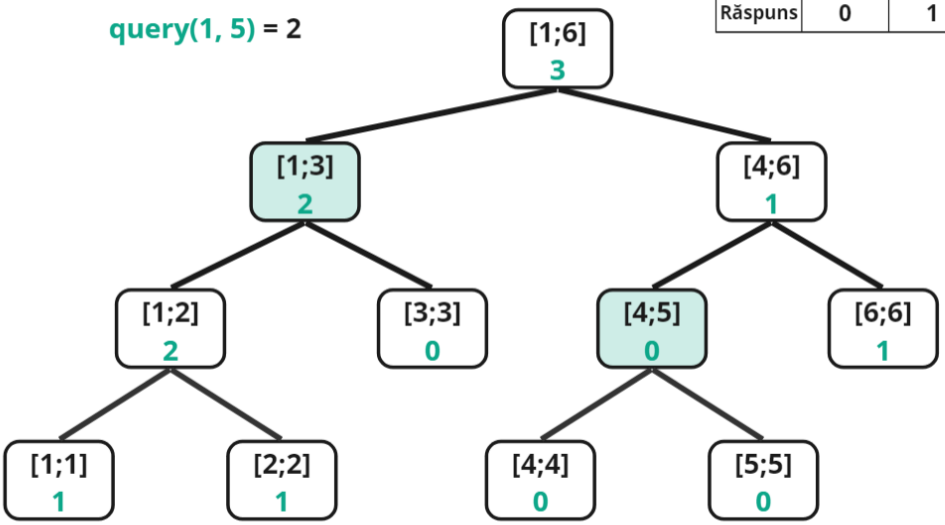
Număr	1	6	2	5
Răspuns	0	1	1	

update(2, +1)



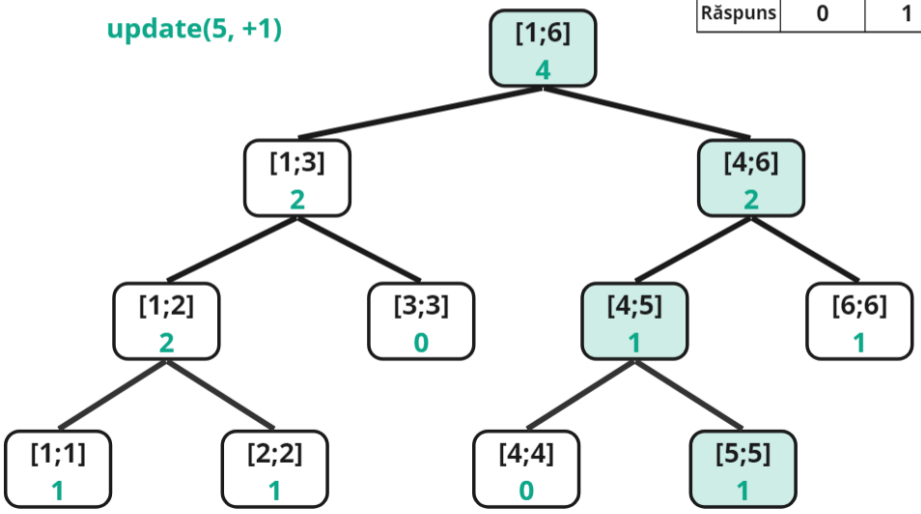
Număr	1	6	2	5
Răspuns	0	1	1	2

query(1, 5) = 2



Număr	1	6	2	5
Răspuns	0	1	1	2

update(5, +1)



Răspuns final:

Număr	1	6	2	5
Răspuns	0	1	1	2

Problema 3:

*Avem **N** copii aşezaţi într-un cerc. Începem de la primul copil. La fiecare pas **i = [1..N]**, se numără **i** copii începând cu copilul curent, iar copilul la care se ajunge este eliminat (din joc). Copilul curent va fi acum următorul copil de după copilul eliminat şi se va trece la pasul următor **(i+1)**. La primul pas se elimină copilul 2. Spuneţi în ce ordine sunt eliminaţi copiii.*

Link problemă: https://www.infoarena.ro/problema/order

Exemplu: N=6 → răspuns: 2 4 1 3 5 6

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

Soluţie:

Copiii sunt numerotaţi de la **1** la **N**. Vom considera **V[x] = 1**, dacă copilul cu indicele **x** încă este în joc, respectiv **V[x] = 0**, dacă copilul cu indicele **x** a fost eliminat. Vom crea un arbore de intervale, unde fiecare nod va reţine câţi copii activi (adică încă în joc) sunt în intervalul respectiv.

La un pas **i**, ne aflăm la copilul **x** şi trebuie să numărăm **i** copii începând cu **x**. În acest stadiu, s-au eliminat deja la paşii anteriori **(i-1)** copii, aşadar acum sunt **(n-i+1)** copii. Dacă avem de numărat mai mulţi copii decât mai sunt în joc, este clar că la un moment dat ne vom reîntoarce în poziţia în care suntem, adică în **x**. Din acest motiv, ne permitem să spunem că, în loc să numărăm **i** copii, este suficient să numărăm **i' = i %(n-i+1)**.

Dacă **query(x,N) >= i'**, atunci vrem să căutăm **y** *minim* cu proprietatea că **query(x,y)=i'**. Copilul eliminat va fi **y**. Acest lucru are sens deoarece ne putem gândi că dacă **query(x,y-1)=i'-1** şi **query(x,y)=i'**, atunci este clar că **y** este al **i'**-lea copil activ.

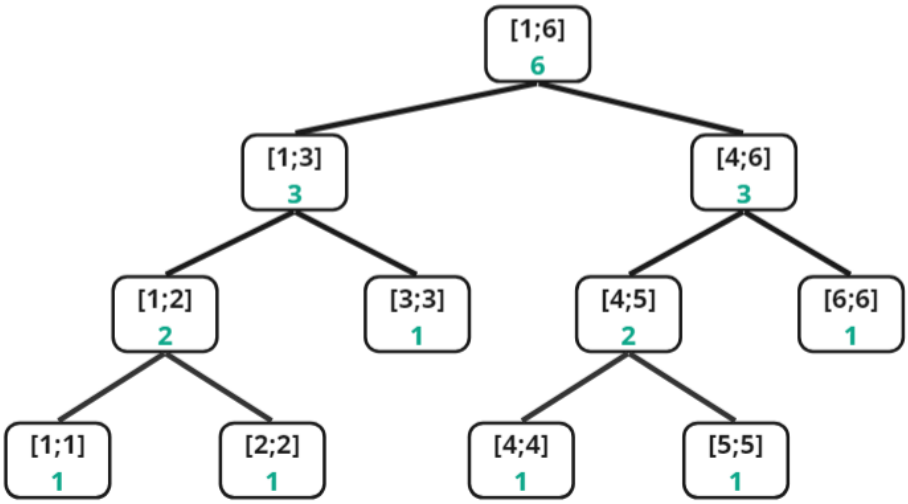
Dacă **query(x,N) < i'**, adică numărătoarea depăşeşte şirul de copii şi se continuă de la primul, atunci vom considera **i'' = i' - query(x,N)**, cu semnificaţia că am numărat copiii până la finalul şirului, şi ne-au mai rămas de numărat **i''** copii pe care îi numărăm

începând cu 1. Pentru acești **i''** copii, vom căuta **y** *minim* cu proprietatea că **query(1,y)= i''**, iar acest **y** va fi copilul eliminat.

Pentru a căuta într-un arbore de intervale (căutarea **y**-ului de mai sus), folosim căutare binară. Intuitiv, acest lucru se face căutând binar **y** și comparând **query(x,y)** cu valoarea dorită. Căutarea binară propriu zisă se face în **O(log N)**, query-ul tot în **O(log N)**, așadar căutarea pe arbore se va face în **O(log^{2}N)**. *Dar*, acest lucru poate fi îmbunătățit și putem obține **O(log N)** - detalii aici.

Arborele inițial

copil	1	2	3	4	5	6
V[copil]	1	1	1	1	1	1



*problema spune că se începe de la 2

copil	1	2	3	4	5	6
V[copil]	1	1	1	1	1	1

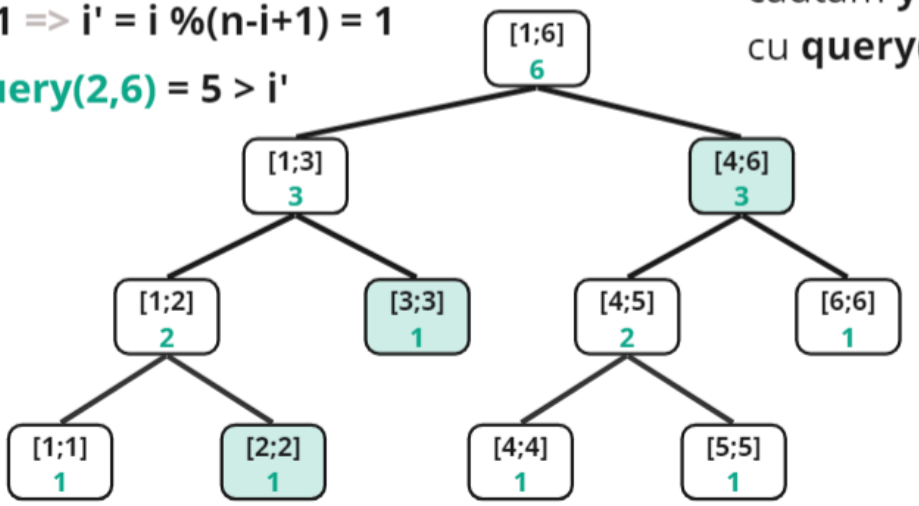
copil_curent = 2

i = 1 => i' = i %(n-i+1) = 1

query(2,6) = 5 > i'

căutăm **y** *minim*, cu **query(2,y) = i'**

y = 2

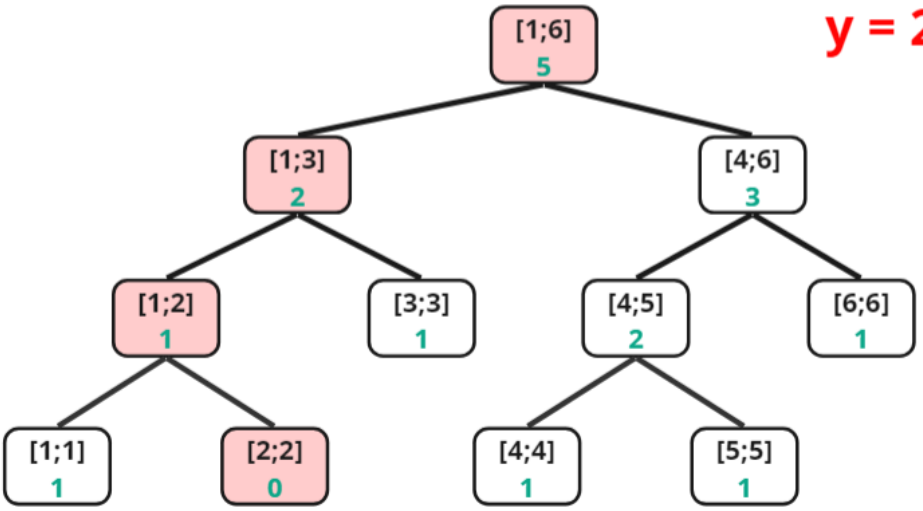


copil_curent = 2

i = 1 => i' = 1

copil	1	2	3	4	5	6
V[copil]	1	0	1	1	1	1

y = 2



copil_curent = 2

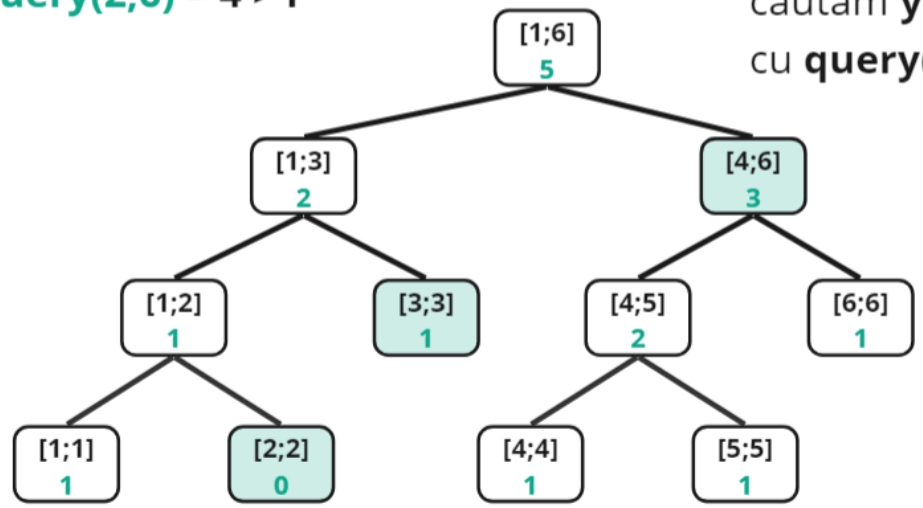
i = 2 => i' = 2

copil	1	2	3	4	5	6
V[copil]	1	0	1	1	1	1

query(2,6) = 4 > i'

căutăm **y** *minim*,
cu **query(2,y) = i'**

y = 4

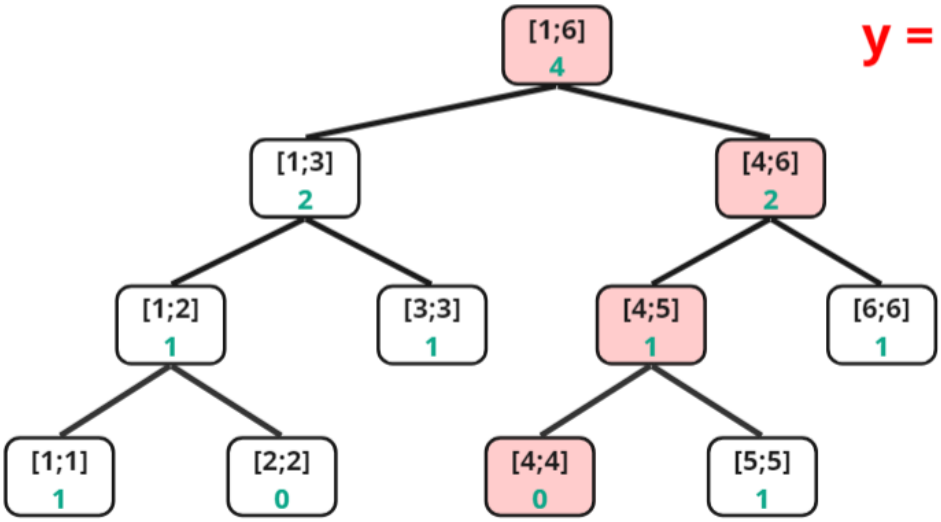


copil_curent = 2

i = 2 => i' = 2

copil	1	2	3	4	5	6
V[copil]	1	0	1	0	1	1

y = 4



copil_curent = 4

i = 3 => i' = 3

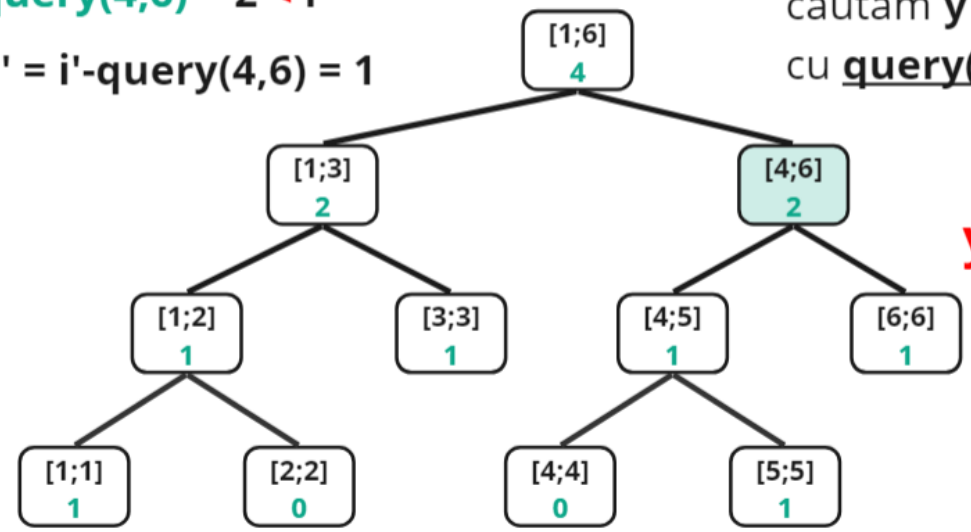
copil	1	2	3	4	5	6
V[copil]	1	0	1	0	1	1

query(4,6) = 2 < i'

i'' = i'-query(4,6) = 1

căutăm **y** *minim*,
cu **query(1,y) = i''**

y = 1



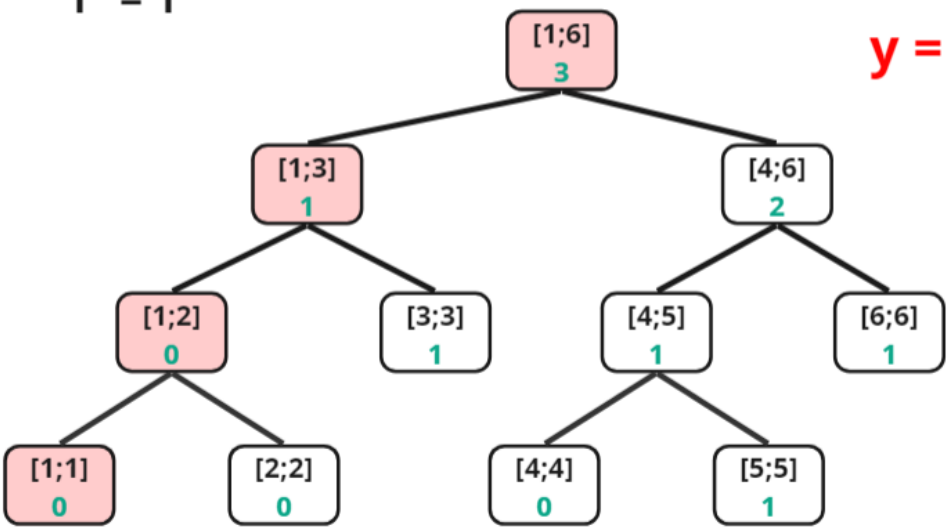
copil_curent = 4

i = 3 => i' = 3

i'' = 1

copil	1	2	3	4	5	6
V[copil]	0	0	1	0	1	1

y = 1



copil_curent = 1

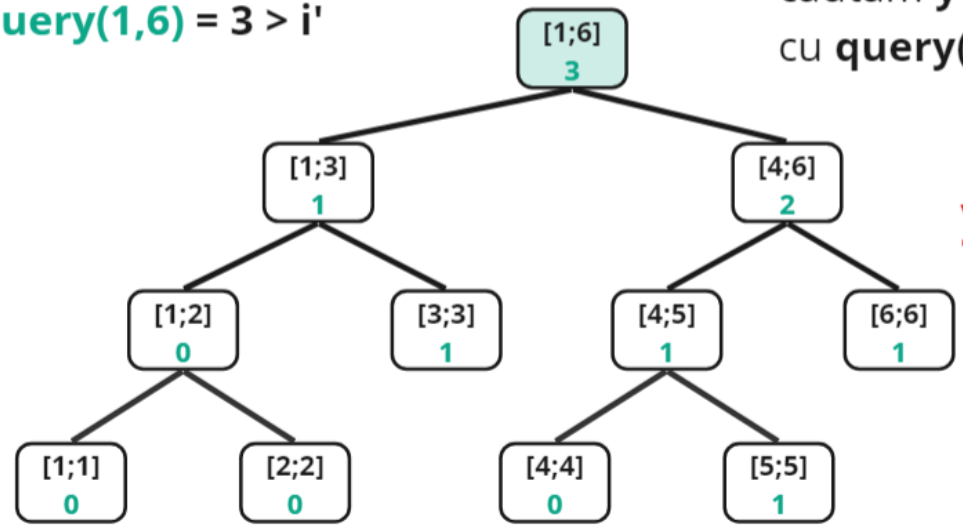
i = 4 => i' = 4%3 = 1

copil	1	2	3	4	5	6
V[copil]	0	0	1	0	1	1

query(1,6) = 3 > i'

căutăm **y** *minim*,
cu **query(1,y) = i'**

y = 3

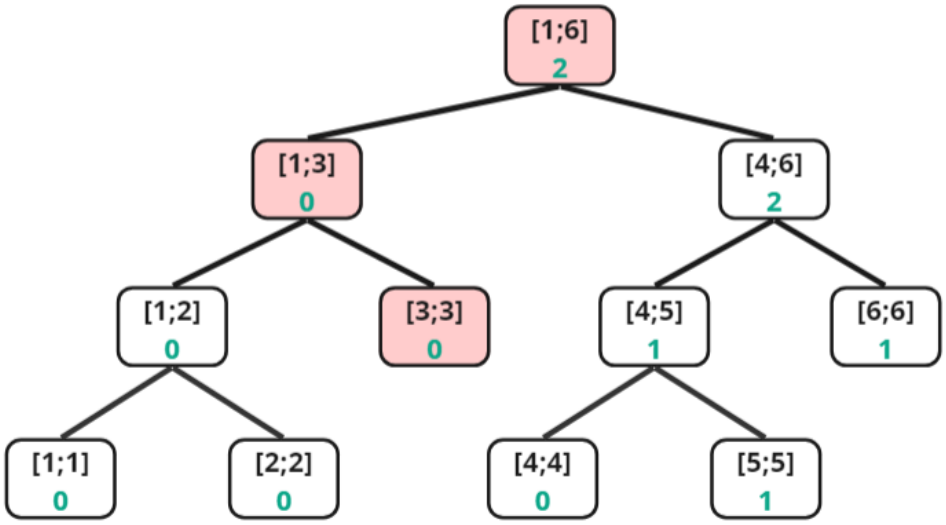


copil_curent = 1

i = 4 => i' = 4%3 = 1

copil	1	2	3	4	5	6
V[copil]	0	0	0	0	1	1

y = 3



copil_curent = 3

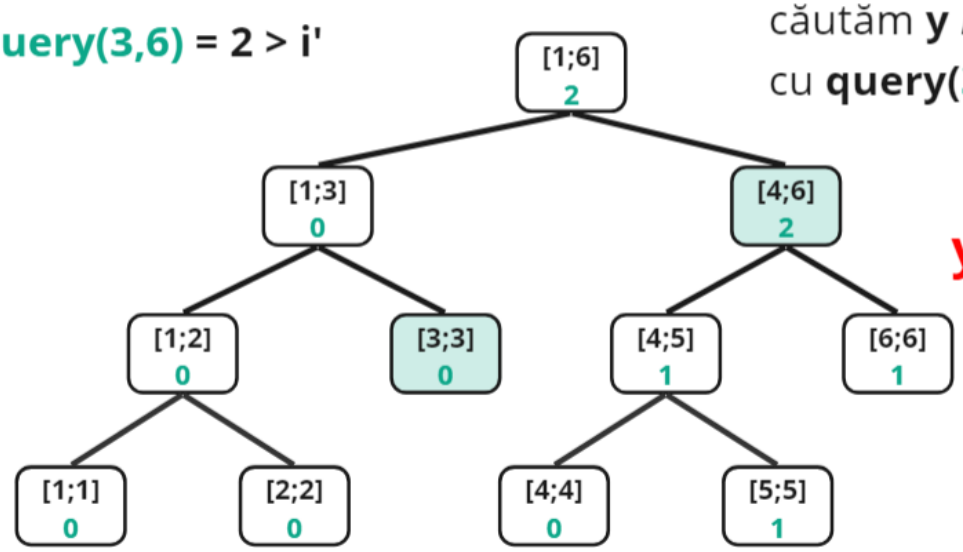
i = 5 => i' = 5%2 = 1

copil	1	2	3	4	5	6
V[copil]	0	0	0	0	1	1

query(3,6) = 2 > i'

căutăm **y** *minim*,
cu **query(3,y) = i'**

y = 5

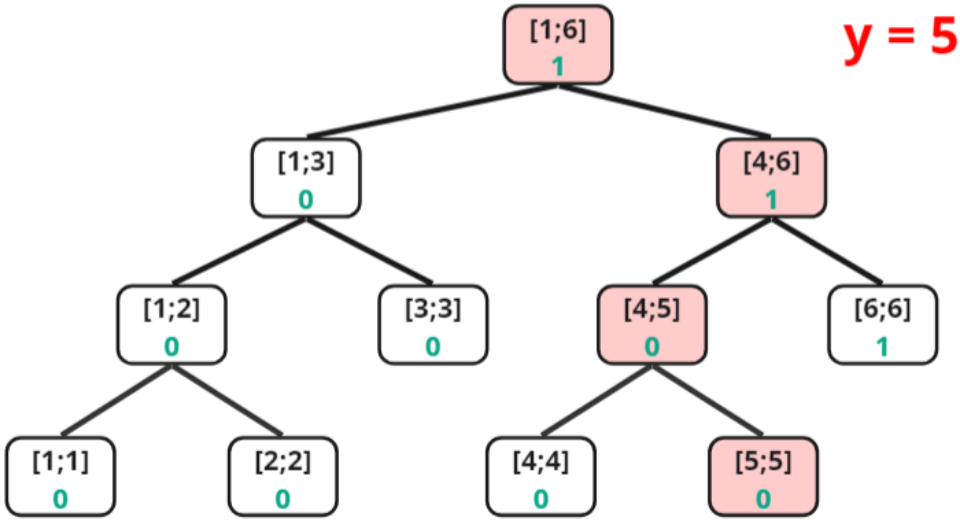


copil_curent = 3

i = 5 => i' = 5%2 = 1

copil	1	2	3	4	5	6
V[copil]	0	0	0	0	0	1

y = 5



A mai rămas doar 6 așa că îl eliminăm și ne oprim. Răspunsul este ordinea eliminării, adică valorile pe care le-a luat y: 2 4 1 3 5 6

Disclaimer: Toate problemele se pot rezolva și folosind Sqrt Decomposition (aka Șmenul lui Batog) în **O(N*sqrt(N))** timp. Mai multe detalii:

- Descriere algoritm: https://www.geeksforgeeks.org/square-root-sqrt-decomposition-algorithm/
- Video pe larg: https://www.youtube.com/watch?v=tLDEZkytmXQ&ab_channel=DanPracsiu

În general, o soluție care folosește Sqrt Decomposition în loc de arbori de intervale asigură între 50 și 100 de puncte pe problemă, dar de multe ori este mai ușor de înțeles și de implementat.