Structuri de Date

Seminar 2

Cuprins:

Problema 1:

Se dă un vector cu N elemente. Câte inversiuni are acesta?

Inversiune = pereche **(i, j)** unde **i < j && v[i] > v[j]**

Exemplu: 3 4 1 2 5 → 4 inversiuni: (3 1), (3 2), (4 1), (4 2)

Soluția 1:

Folosim două for-uri ca să comparăm fiecare element cu toate elementele din dreapta lui.

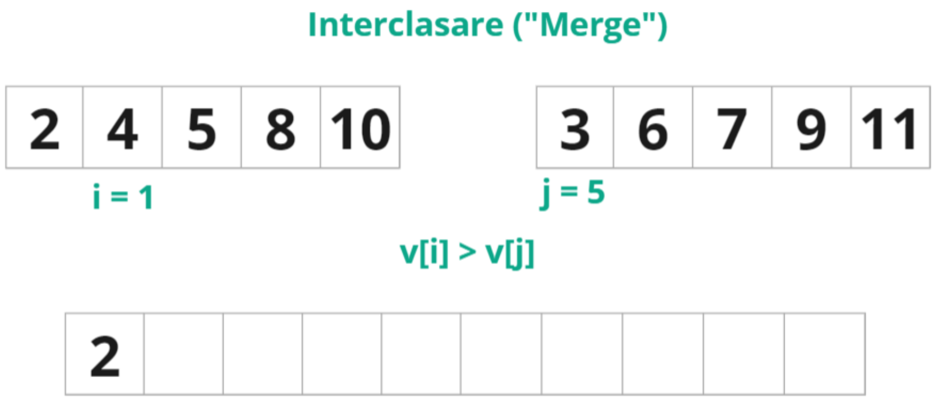
```
int no_of_inversions (const vector<int> &v) {
    int inversions = 0;
    for (int i = 0; i < v.size(); ++i) {
        for (int j = i + 1; j < v.size(); ++j) {
            if (v[i] > v[j])
                ++inversions;
        }
    }

    return inversions;
}
```

Complexitate timp: **$O(N^2)$**

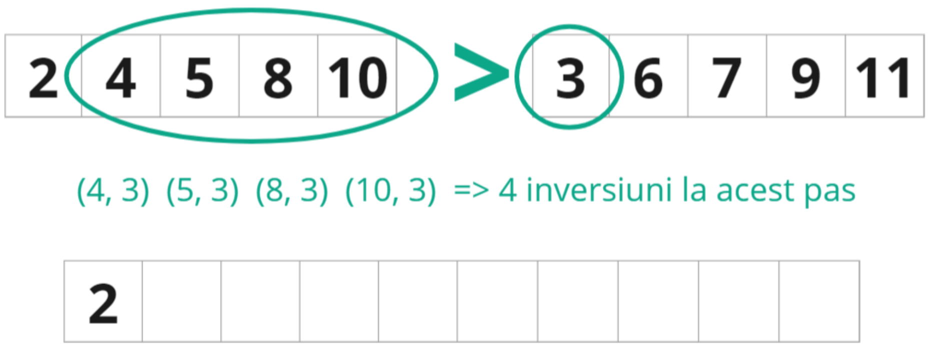
Soluţia 2:

Putem folosi Merge Sort. Dacă avem un vector în care prima jumătate este sortată și a doua jumătate este sortată (exemplu: 5 6 7 8 1 2 3 4), putem deduce despre numărul de inversiuni următorul lucru:

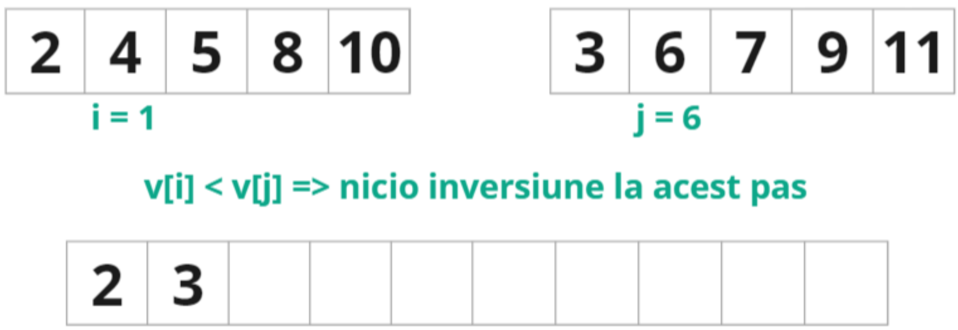
Să ne gândim că sortăm un vector folosind Merge Sort. Am împărțit vectorul în două și am sortat fiecare jumătate. Suntem la pasul de *merge* :

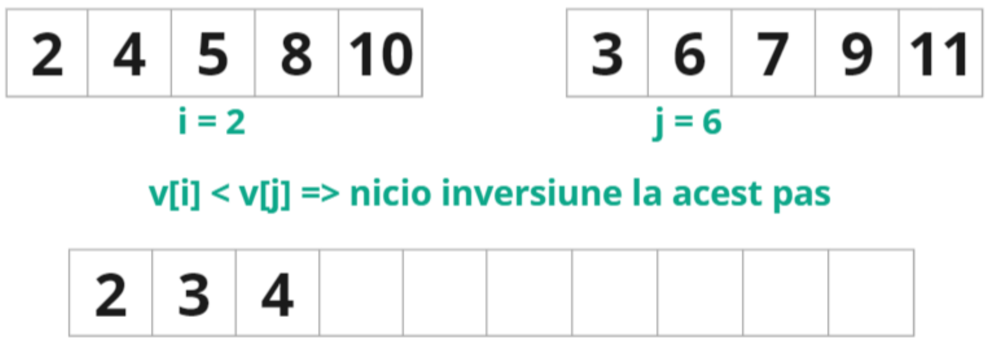


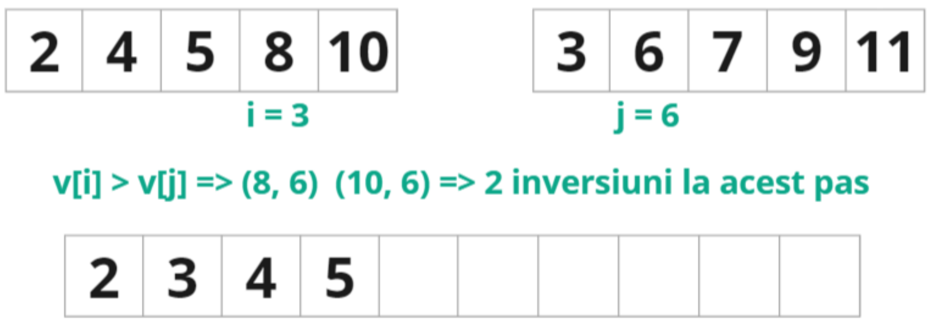
Observăm că un element din stânga **(4)** este **mai mare** decât un element din dreapta **(3)**. Nu știm pe ce poziție erau 4 și 3 înainte de sortări, dar știm cu siguranță că 4 era în prima jumătate din vector, iar 3 în a doua jumătate (=> mai în dreapta decât 4). Așadar, aceasta (perechea 4 3) este o inversiune! Mai mult decât atât, toate elementele din vectorul din stânga care se află în dreapta lui 4, vor fi și ele mai mari decât 3, deoarece vectorul din stânga este sortat:



Dacă ținem cont de această proprietate la toți pașii din Merge Sort, la final vom obține numărul de inversiuni (și vom avea și vectorul sortat). Mai jos găsim continuarea exemplului:







2	4	5	8	10

i = 3

3	6	7	9	11

j = 7

v[i] > v[j] => (8, 7) (10, 7) => 2 inversiuni la acest pas

2	3	4	5	6					

2	4	5	8	10

i = 3

3	6	7	9	11

j = 8

v[i] < v[j] => nicio inversiune la acest pas

2	3	4	5	6	7				

2	4	5	8	10

i = 4

3	6	7	9	11

j = 8

v[i] > v[j] => (10, 9) => o inversiune la acest pas

2	3	4	5	6	7	8			

2	4	5	8	10

i = 4

3	6	7	9	11

j = 9

nicio inversiune până la final

2	3	4	5	6	7	8	9	10	11

Însumând, obținem 9 inversiuni pentru acest *merge*. Rezultatul final este 9, la care adunăm inversiunile obținute până atunci pentru vectorii și mai mici.

```
inversions += mergeSort(v, left, mid);
inversions += mergeSort(v, mid + 1, right);

inversions += merge(v, left, mid, right);
```

Complexitatea timp este cea de la Merge Sort, anume **O(N log N)**.

Alternativ:

Se poate rezolva în O(N log N) folosind arbori binari de căutare echilibrați, sau arbori de intervale după normalizare.

Problema 2:

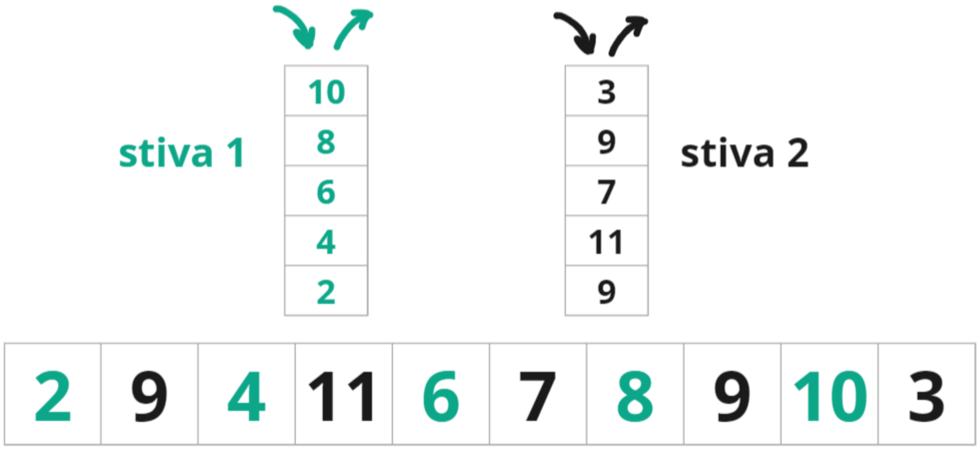
Cum implementăm două stive folosind un vector?

Pentru început, să ne gândim cum implementăm o singură stivă folosind un vector:

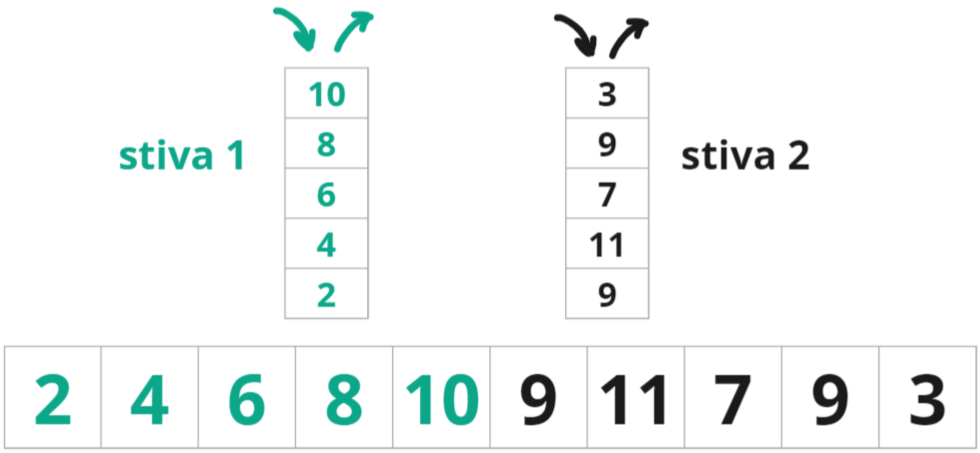
```
vector<int> my_stack;

void push (vector<int> &stack, int val) {
    stack.push_back(val);
}

int pop (vector<int> &stack) {
    if (stack.empty())
        throw runtime_error("Cannot perform Pop() on empty stack.");

    int top = stack.back();
    stack.pop_back();

    return top;
}
```

Acum, pentru a implementa două stive folosind un singur vector, **soluțiile** sunt următoarele:

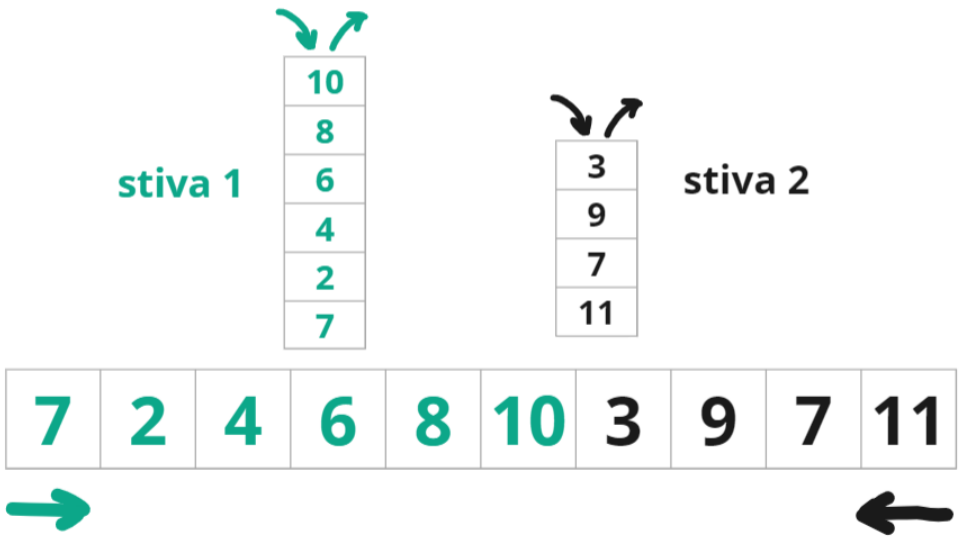
- Indici pari pentru o stivă și indici impari pentru cealaltă
    - ◊ Fiecare stivă va putea avea N/2 elemente
    - ◊ Push și pop in O(1)



- O stivă de la începutul vectorului până la mijloc, iar cealaltă de la mijloc + 1 până la final
  - ◊ Fiecare stivă va putea avea N/2 elemente
  - ◊ Push și pop in O(1)



- Prima stivă începe la începutul vectorului și se continuă spre dreapta, iar cealaltă stivă începe la sfârșitul vectorului și se continuă spre stânga (practic, dacă ar avea număr egal de elemente, vârfurile stivelor s-ar întâlni la mijlocul vectorului)
  - ◊ Push și pop in O(1)
  - ◊ Stivele pot avea oricâte elemente, atâta timp cât suma lor este mai mică sau egală cu N: acest lucru înseamnă că o stivă poate avea și N elemente



Problema 3:

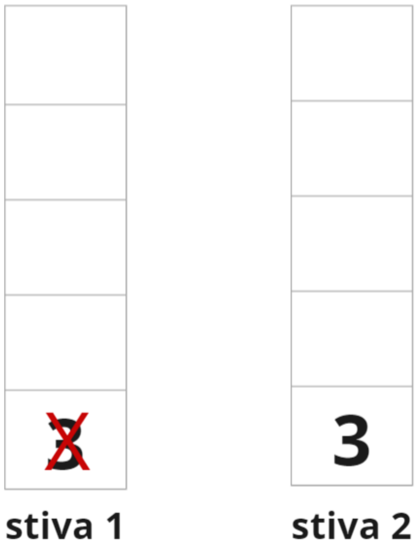
Implementați o coadă folosind două stive.

Soluția 1:

Ne propunem să avem mereu primul element din coadă în vârful primei stive, ca atunci când facem **pop** să o facem în O(1). Pentru acest lucru, operațiile vor presupune următoarele:

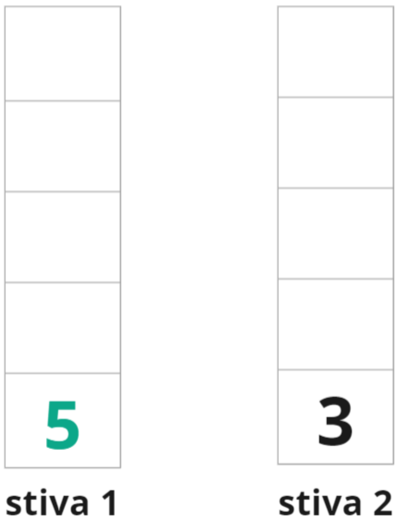
- **Push în coadă**: Mutăm toate elementele din stiva_1 în stiva_2, introducem elementul nou în stiva_1, mutăm toate elementele din stiva_2 înapoi în stiva_1; Complexitate timp: O(N)
- **Pop din coadă**: Pop la elementul din vârful primei stive; Complexitate timp: O(1)

push 3



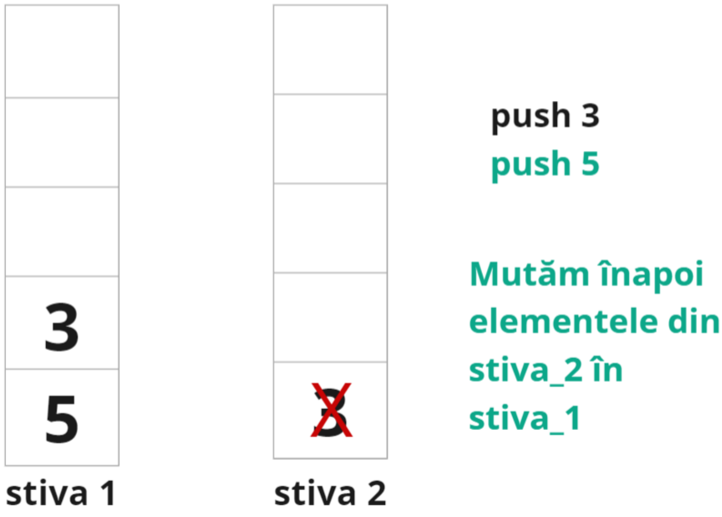
push 3
push 5

Mutăm totul
din stiva_1
în stiva_2

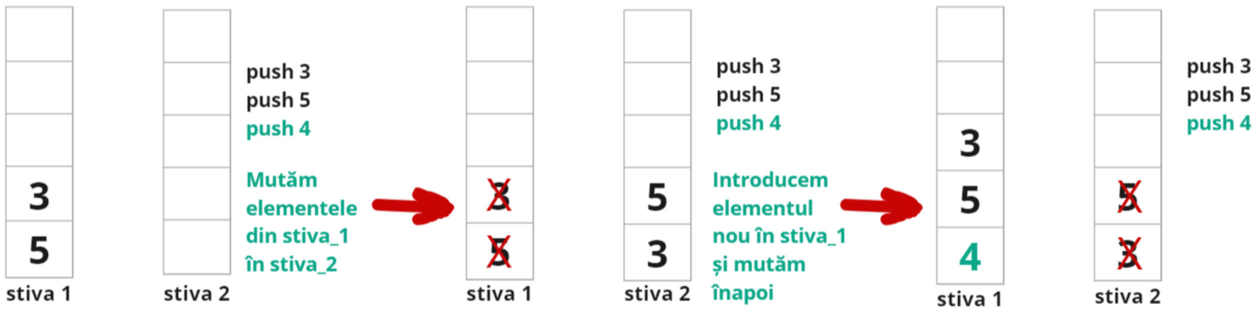


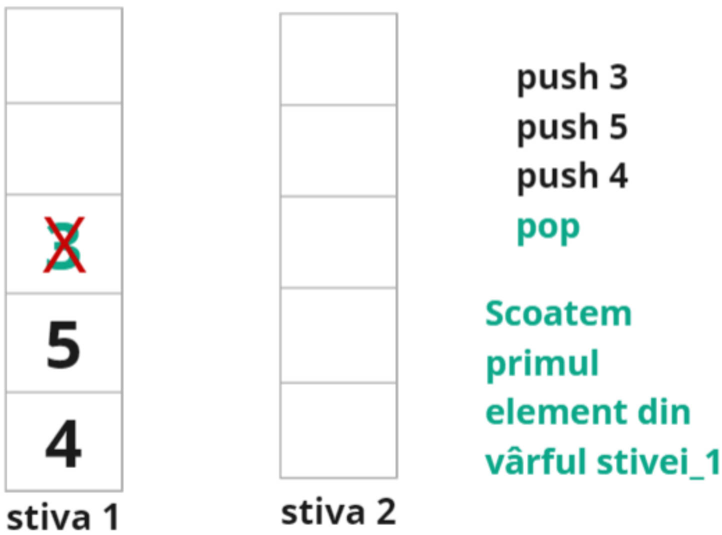
push 3
push 5

Introducem
elementul nou
în stiva_1



Observăm că ultimul element introdus a ajuns la baza stivei, ceea ce înseamnă că va ieși ultimul din stivă, exact ce ne dorim când simulăm o coadă (ultimul intrat = ultimul ieșit, precum și primul intrat = primul ieșit).

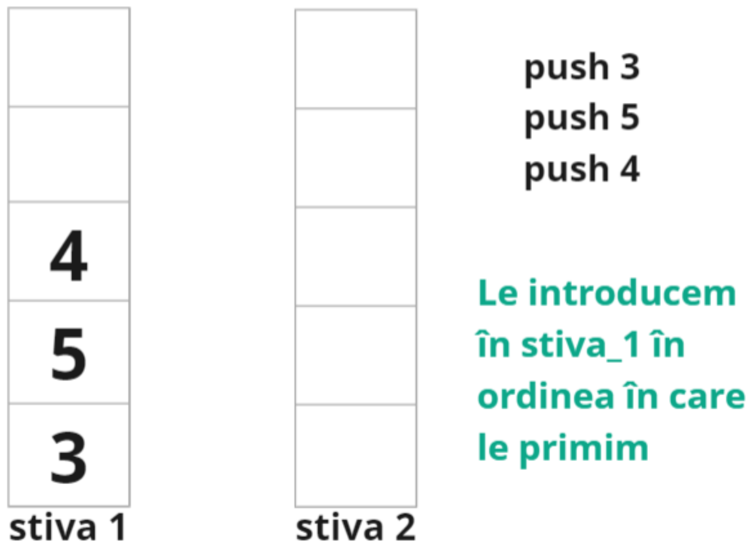


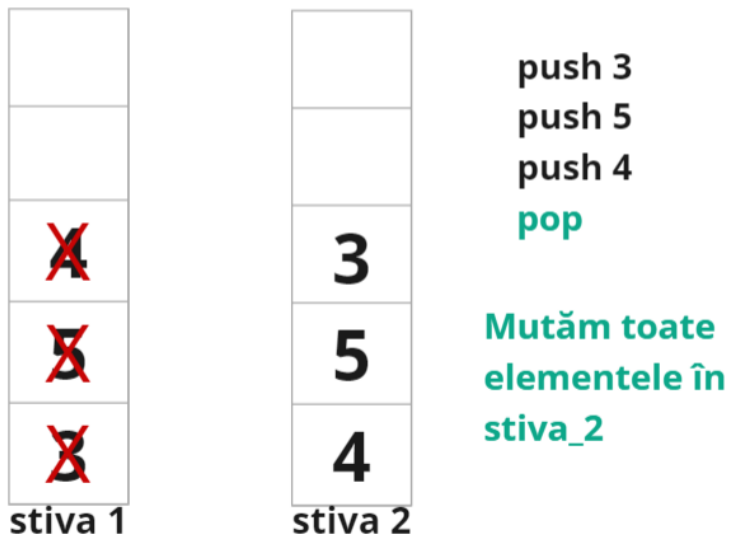


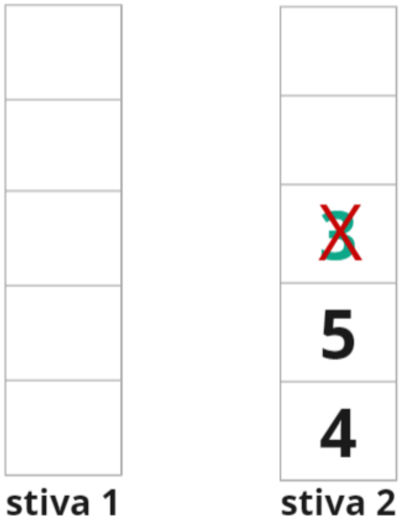
Soluția 2:

Ne propunem să facem operația de **push** în **O(1)**. Pentru acest lucru, operațiile vor arăta astfel:

- **Push în coadă**: adăugăm elementul nou în stiva_1; Complexitate timp: **O(1)**
- **Pop din coadă**: dacă stiva_2 este goală, mutăm totul din stiva_1 în stiva_2. Dăm pop la elementul din vârful stivei_2; Complexitate timp: **O(N) (amortizat O(1))**

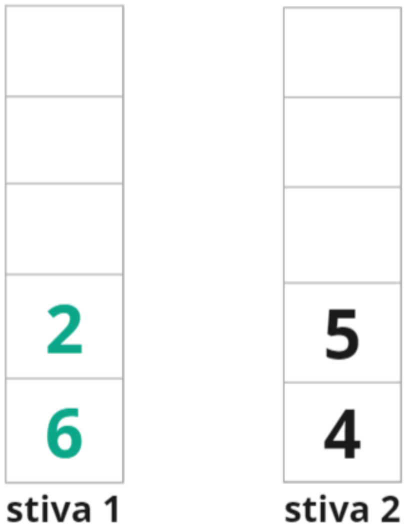






push 3
push 5
push 4
pop

Dăm pop
din stiva_2



push 3
push 5
push 4
pop
push 6
push 2

Adăugăm
în stiva_1

push 3
push 5
push 4
pop
push 6
push 2
pop

Dăm pop
din stiva_2

Pentru acasă: *Implementați o stivă folosind două cozi.*

<u>Problema 4:</u>

Se dă un șir de paranteze. Să se verifice dacă este o "parantezare" corectă.

a) Șirul conține doar paranteze rotunde ( ).
b) Șirul conține paranteze rotunde ( ) și paranteze pătrate [ ].

```
<șir parantezat corect> = <șirul vid>
<șir parantezat corect> = "(" + <șir parantezat corect> + ")"
<șir parantezat corect> = "[" + <șir parantezat corect> + "]"
<șir parantezat corect> = <șir parantezat corect> + <șir parantezat corect>
```

a) Şirul conţine doar paranteze rotunde ( ).

Exemple:

( )( )( ) → corect, explicație: ( )( )( )

(( ))) ( ) → incorect, explicație: (( ))) ( ) cea roșie se închide dar nu se deschide

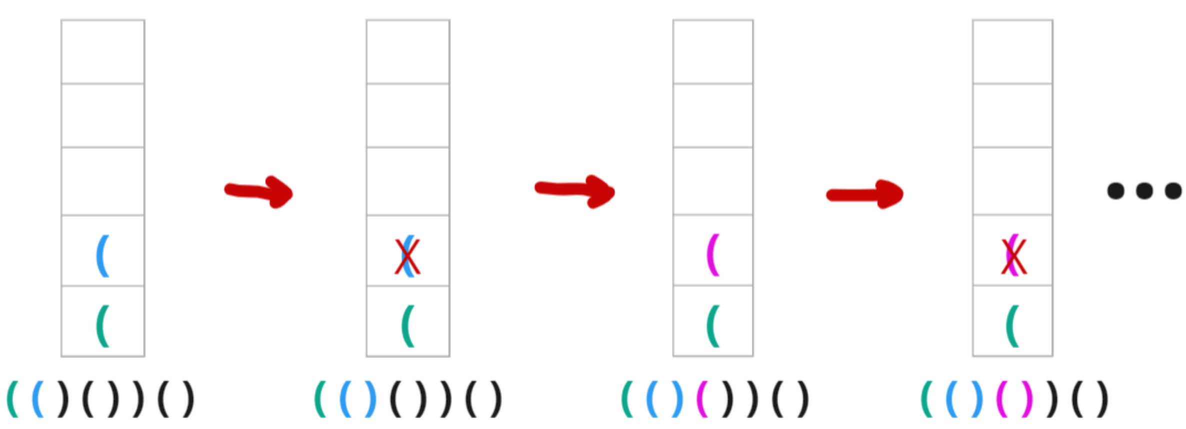
) (→ incorect

Soluţia 1:

Folosim o stivă. Când citim o paranteză deschisă "(" o introducem în stivă. Dacă în vârful stivei avem o paranteză deschisă "(" și citim o paranteză închisă ")", acestea formează o pereche corect parantezată, deci scoatem paranteza deschisă din stivă. Dacă citim o paranteză închisă, dar nu avem nicio paranteză deschisă în stivă, atunci șirul nu este corect parantezat. Șirul este corect parantezat dacă ajungem la final și stiva este goală.

Complexitate timp: **O(N)**, N = lungimea șirului

Spațiu auxiliar: **O(N)** - stiva



Soluția 2:

Folosim o variabilă cu rol de contor. Când întâlnim o paranteză deschisă, incrementăm variabila, iar când întâlnim o paranteză închisă decrementăm variabila. Șirul este corect parantezat dacă variabila nu ajunge negativă în niciun punct al algoritmului și dacă la final este egală cu 0.

Complexitate timp: **O(N)**

<u>Spațiu auxiliar:</u> **O(1)**

b) Şirul conţine paranteze rotunde ( ) şi paranteze pătrate [ ].

Exemple:

([ ]) → corect

([ )] → incorect; acest exemplu ne arată că soluția cu variabila contor de mai sus nu mai funcționează

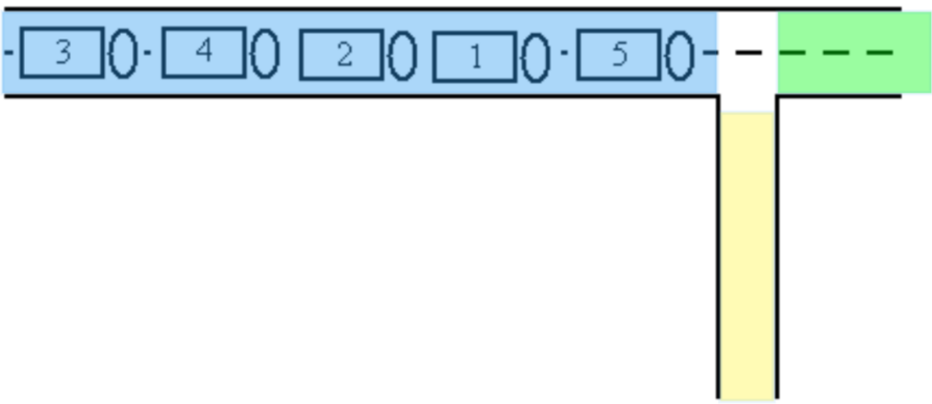
Soluţia:

Adaptăm soluția cu stiva de la punctul a). Tot ce trebuie să facem în plus este să verificăm, atunci când citim o paranteză închisă, dacă aceasta și paranteza deschisă din vârful stivei au același tip (ambele rotunde sau ambele pătrate).

<u>Problema 5:</u>

Enunţ complet: https://www.spoj.com/problems/STPAR/

Rezumat: Avem o stradă îngustă și **N** camioane ca în figura de mai jos:

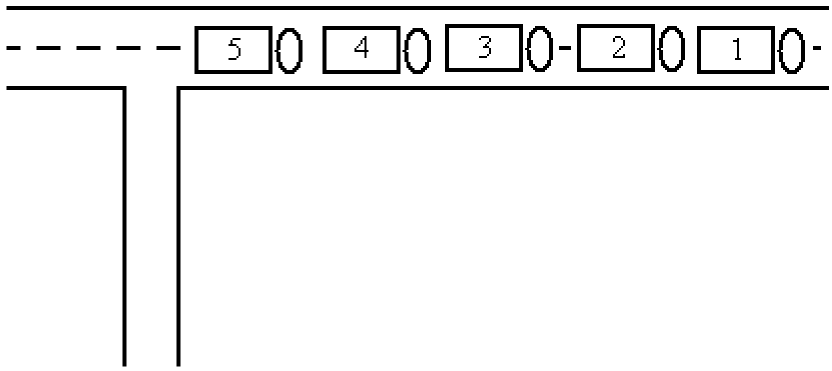


Albastru: zona în care sunt dispuse inițial cele N camioane, într-o ordine dată.

Verde: zona în care vrem să ajungă cele N camioane pe rând, în ordine crescătoare de la 1 la N.

Galben: zona în care pot intra și staționa camioanele (i.e.: să permită altui camion din zona albastră să intre în zona verde primul); camioanele nu se pot întoarce din zona galbenă în zona albastră, ci pot avansa doar în zona verde.

Configurația finală trebuie să arate astfel:



Cerinţă: *Spuneți dacă mașinile pot ajunge pe strada din dreapta în configurația dorită.*

Soluţie:

Începem cu un contor de la 1, reprezentând următoarea mașină care trebuie să ajungă pe strada finală (aka în zona verde). Ne uităm la mașinile de pe strada din stânga (zona albastră), și le mutăm pe străduța lăturalnică (zona galbenă) până ajungem la mașina căutată (valoarea contorului). Mutăm mașina cu valoarea contorului în dreapta (zona verde) și incrementăm contorul. La fiecare pas verificăm dacă mașina căutată este în vârful străzii lăturalnice, caz în care o mutăm pe strada finală și incrementăm contorul. Dacă mașina căutată nu este în vârful străzii lăturalnice, facem din nou ca la început (mutăm mașinile din stânga pe strada lăturalnică până găsim mașina căutată).

Dacă ne blocăm undeva (am mutat toate mașinile din stânga și nu am găsit mașina căutată), atunci mașinile **NU** pot fi mutate în configurația dorită.

Problema 6:

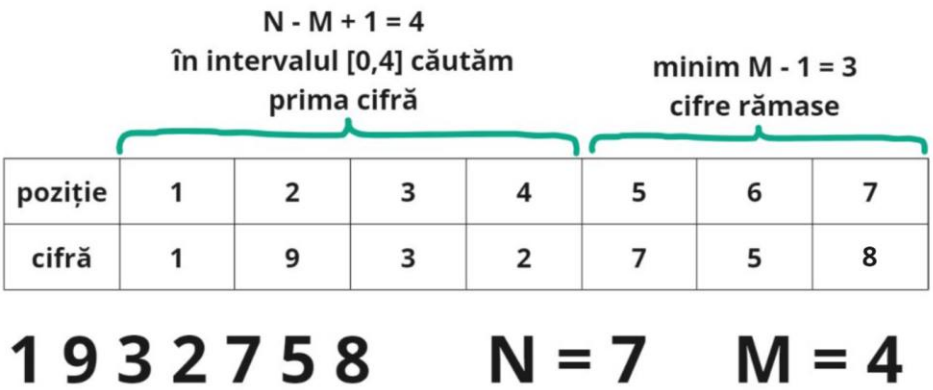
Enunţ complet: https://www.infoarena.ro/problema/trompeta

Rezumat: Avem N cifre într-o ordine dată (practic, un număr foarte lung) și un număr **M ≤ N**. Vrem să alegem din cele N cifre, M cifre fără a le schimba ordinea inițială (nu trebuie să fie consecutive, dar nu putem inversa două cifre) astfel încât dacă le lipim pe toate într-un număr, acesta este maxim.

Exemplu: M=4, cifre = 19990 => numărul maxim obținut cu M cifre fără a schimba ordinea inițială este 9990

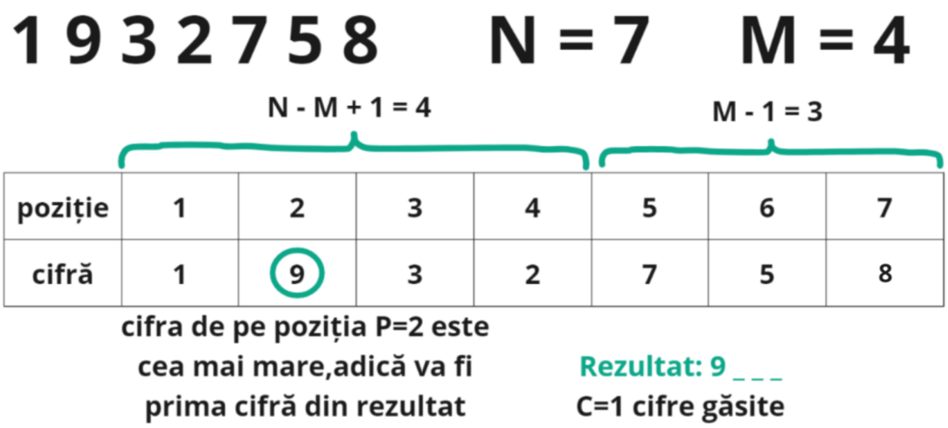
Soluția 1:

Ne putem gândi că cel mai mare număr va fi mereu obținut folosind ca primă cifră cea mai mare cifră din șir. Însă avem nevoie de încă **(M-1)** cifre după aceasta, și nu avem garanția că cea mai mare cifră mai are atâtea cifre după ea în șir. Așadar, ne-am dori ca prima cifră să se găsească în primele **(N-M+1)** cifre (astfel am avea garanția că are cel puțin **M-1** cifre după ea în șir).



Punem cifrele într-un vector și găsim cifra maximă din intervalul **[0, N-M+1]**. Aceasta va fi prima cifră din numărul rezultat. Să notăm poziția ei cu **P**.

Dacă notăm cu **C** numărul de cifre din rezultat găsite până acum, **M-C** va însemna câte cifre mai trebuie să găsim.



În continuare vrem să găsim **valoarea maximă din dreapta cifrei alese deja, dar care mai are după ea cel puțin (M-C-1) cifre în șir.** Așadar căutăm în intervalul **[P+1, N-M+C+1]**.

1 9 3 2 7 5 8 N = 7 M = 4

[P+1=3 ; N-M+C+1=5] M-C-1=2

poziție	1	2	3	4	5	6	7
cifră	1	9	3	2	7	5	8

P=2

Rezultat: 9 _ _ _
C=1 cifre găsite

1 9 3 2 7 5 8 N = 7 M = 4

poziție	1	2	3	4	5	6	7
cifră	1	9	3	2	7	5	8

P=5

Rezultat: 9 7 _ _
C=1 cifre găsite

1 9 3 2 7 5 8 N = 7 M = 4

M-C=2

poziție	1	2	3	4	5	6	7
cifră	1	9	3	2	7	5	8

Rezultat: 9 7 5 8
~~C=2 cifre găsite~~

N-P = M-C => ne oprim și adăugăm la rezultat toate cifrele rămase

Soluția 2:

Folosim o stivă. Intuitiv, această abordare presupune mai mult “Ce cifre scoatem?”, pe când cealaltă presupune “Ce cifre păstrăm?”.

Reținem un contor care începe de la **N-M** și reprezintă câte cifre mai trebuie să eliminăm din cele N.

Parcurgem cele N cifre. Dacă stiva este goală, adăugăm cifra curentă în stivă. Altfel, scoatem din stivă toate elementele care sunt mai mari decât cifra curentă, decrementând contorul la fiecare cifră scoasă și având grijă ca acesta să nu ajungă negativ. Când am ajuns în stivă la un element mai mic sau egal decât cifra curentă (vom observa că stiva va fi mereu ordonată descrescător de jos în sus) sau când stiva a ajuns goală, adăugăm cifra curentă în stivă și continuăm.

Exemplu: N=7, M=4, cifre: 1932759