Stiva - Aplicatii

Dicu Adrian-Emanuel

Decembrie 2021

Contents

In acest material voi prezenta diverse probleme de informatica de dificultate variata (de la foarte usoare pana la probleme date la concursuri si olimpiade internationale), care folosesc in principal structura de date: stiva. Voi evita problemele care se rezolva cu o combinatie de algoritmi si structuri de date, ca sa evidentiez in special diferitele aplicatii ale stivei, fara sa fie nevoie de alte cunostinte in prealabil. Problemele sunt ordonate crescator dupa dificultate.

1 Probleme usoare

1.1 Problema Trompeta

Problema se rezolva prin metoda greedy.

Prima observatie este ca din moment ce raspunsul va avea fix M cifre, ideal este sa avem 'cifre cat mai mari la inceputul rezultatului'.

A 2-a observatie este ca raspunsul va fi un subsir de M cifre din cele N cifre initiale si conform primei observatii, vrem sa avem prima cifra din acest subsir cat mai mare posibila. In caz de egalitate, vrem sa avem a 2-a cifra cat mai mare posibila, si asa mai departe.

Parcurgem cifrele de la stanga la dreapta si vom folosi o stiva pentru a retine pozitiile cifrelor din solutia finala. Acum trebuie sa vedem daca eliminam cifra de pe pozitia varfului stivei.

Cand vom elimina cifra din varful stivei? Cand cifra curenta este mai mare (deci mai buna) decat cifra din varful stivei, respectiv nu am eliminat mai mult de $N - M$ cifre pana acum (trebuie sa garantam faptul ca in final vom avea cel putin M cifre ramase.

```
int top = 0, eliminated = 0;
for (int i = 1; i <= n; ++i) {
  while (top > 0 && eliminated < n - m && digit[i] > digit[my_stack[top]]) {
    --top;
    ++eliminated;
  }
  my_stack[++top] = i;
}
```

La finalul algoritmului este posibil sa fi eliminat mai putin de $N - M$ cifre, caz in care trebuie sa eliminam cifre din varful stivei pana cand vom ajunge la un subsir cu fix M elemente.

```
while (eliminated < n - m) {
  --top;
  ++eliminated;
}
```

Codul sursa aici

1.2 Problema Par

Vom folosi (iar) metoda greedy.

In primul rand, verificam daca numarul total de paranteze este impar, caz in care nu avem solutie. Daca numarul de paranteze este par, atunci vom parcurge parantezele de la stanga la dreapta si vom folosi o stiva pentru a tine evidenta numarului de paranteze deschise inca necuplate intre ele.

Observatia esentiala este ca trebuie sa *inversam paranteze doar daca este absolut necesar*. Asadar, cand suntem la paranteza curenta, avem 2 optiuni:

- daca este o paranteza deschisa, atunci o adaugam pe stiva
- daca este o paranteza inchisa, avem 2 cazuri: daca stiva este goala atunci suntem fortati sa o inversam (nu putem avea mai multe paranteze inchise decat deschise intr-un prefix). Daca stiva nu e goala, atunci eliminam varful stivei

```
int top = 0, inverted = 0;
for (int i = 1; i <= n; ++i) {
  if (bracket[i] == '(')
    ++top;
  else {
    if (top > 0) {
```

```
      --top;
    } else {
      ++top;
      ++inverted;
    }
  }
}
```

La final, o sa ramanem cu un sir de paranteze deschise inca necuplate, si va trebui sa inversam exact jumatate din ele ca sa obtinem un sir corect parantezat.

```
out << inverted + top / 2 << std::endl;
```

Codul sursa aici

1.3 Problema Editor

Primul pas in rezolvarea problemei este sa ajungem la o expresie care poate fi evaluata direct de catre editor (cu alte cuvinte, sa simulam pasii editorului si sa scapam de stelute). Pentru a realiza acest lucru se va folosi (surpriza surpriza...) o stiva: se parcurg caracterele din sir de la primul la penultimul (ignoram ultimul E din sir) si avem 2 optiuni

- caracterul curent este steluta: trebuie sa eliminam din expresie ultimul caracter (daca stiva e nevida)
- caracterul curent nu este steluta: trebuie sa adaugam la finalul expresiei caracterul curent

```
int length = 0;
for (int i = 1; i <= initial_length; ++i) {
  if (initial_expression[i] == '*') {
    if (length > 0)
      --length;
  } else {
    expression[++length] = initial_expression[i];
  }
}
```

La final vom obtine o expresie formata din paranteze rotunde si patrate care trebuie verificata daca este parantezata corect. Pentru acest lucru se va folosi iar o stiva:

- caracterul curent este '[' sau '(': se adauga in varful stive
- caracterul curent este ']' sau ')': se verifica daca se poate cupla cu ultimul caracter din stiva. In caz afirmativ, varful stivei este eliminat, altfel, sirul nu este corect parantezat.

```
int top = 0;
bool good_expression = true;
for (int i = 1; i <= length and good_expression; ++i) {
  if (expression[i] == '(' or expression[i] == '[') {
    stack[++top] = expression[i];
  } else {
    if (expression[i] == ']') {
      if (top > 0 and stack[top] == '[')
        --top;
      else
        good_expression = false;
    } else {
      if (top > 0 and stack[top] == '(')
        --top;
      else
        good_expression = false;
    }
  }
}
```

Codul sursa aici

1.4 Problema Liste

Vom tine fiecare lista intr-un vector de bool de 120 de elemente, fiecare valoare fiind 0 (elementul nu exista in lista) sau 1 (elementul exista in lista). Observatia esentiala este ca din moment ce vrem sa executam cat mai multe operatii de unificare, putem sa le aplicam greedy: *daca avem ocazia sa unim 2 liste adiacente, atunci o vom face*. Acest lucru se poate simula cu o stiva in care mereu vedem daca lista curenta poate fi unita cu lista din varful stivei, caz in care le vom reuni intr-o singura lista mai mare.

Codul sursa aici

2 Probleme medii

2.1 Problema Permsplit

Vom folosi o tehnica foarte des intalnita la olimpiada, si anume: *tehnica mersului invers*. Mai concret: in loc sa incercam sa taiem intervalul initial, plecam de la N intervale compacte de lungime 1 (intervale de un singur element) si incercam sa le reunim.

Pentru fiecare interval compact vom tine capetele lui: $[left, right]$. Acum, 2 intervale compacte $[left1, right1]$ si $[left2, right2]$, pot fi reunite intr-un singur interval compact daca si numai daca $right1 = left2+1$ sau $right2 = left1 + 1$. In functie de caz, dupa reuniune, se va obtine un nou interval $[left1, right2]$ sau $[left2, right1]$.

Functiile care verifica daca 2 intervale se pot reuni, respectiv cum se construieste reuniunea a 2 intervale adiacente:

```
inline bool can_reunite(std::pair<int, int> segment1,
                        std::pair<int, int> segment2)
{
  if (segment1.first == segment2.second + 1 ||
      segment2.first == segment1.second + 1)
    return true;

  return false;
}

inline std::pair<int, int> reunite(std::pair<int, int> segment1,
                                   std::pair<int, int> segment2)
{
  int new_left = std::min(segment1.first, segment2.first),
      new_right = std::max(segment1.second, segment2.second);

  return std::make_pair(new_left, new_right);
}
```

Folosindu-ne de aceasta observatie, putem rezolva problema cu o stiva astfel:

- pornim de la N intervale de forma $[value[i], value[i]]$
- parcurgem intervalele compacte de la stanga la dreapta si tinem o stiva cu toate intervalele intermediare compacte obtinute.
- cat timp este posibil, incercam sa reunim intervalul curent cu intervalul compact din varful stivei
- la final daca in stiva obtinem un singur interval compact $[1, N]$ atunci avem solutie care poate fi gasita efectuand taieturile in ordinea inversa reunirii intervalelor.

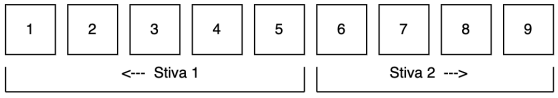
```
  int top = 0, nr_operations = 0;
  for (int i = 1; i <= n; ++i) {
    int x;
    in >> x;

    stack[++top] = std::make_pair(x, x);

    while (top >= 2 && can_reunite(stack[top - 1], stack[top])) {
      operations[++nr_operations] = i - (stack[top].second - stack[top].first + 1);
```

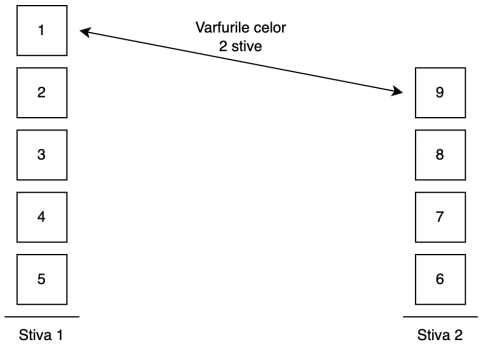
```
        std::pair<int, int> new_segment = reunite(stack[top - 1], stack[top]);
        stack[--top] = new_segment;
    }
}
```

Codul sursa aici

2.2 Problema Queue

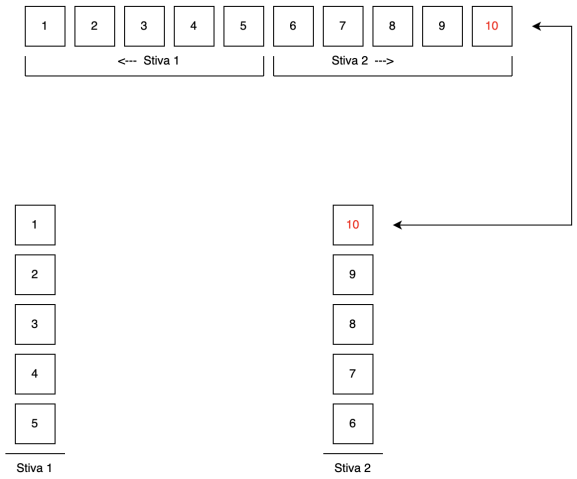
Problema este clasica in informatica. Reprezentam o coada cu 2 stive care sunt puse in directii opuse. Sa presupunem ca in coada avem elementele
$[1, 2, 3, 4, 5, 6, 7, 8, 9]$. Primele 5 elemente presupunem ca sunt in stiva 1 si ultimele 4 in stiva 2. O poza face cat o mie de cuvinte:





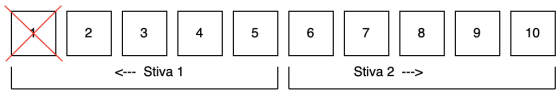
2.2.1 push_back(x)

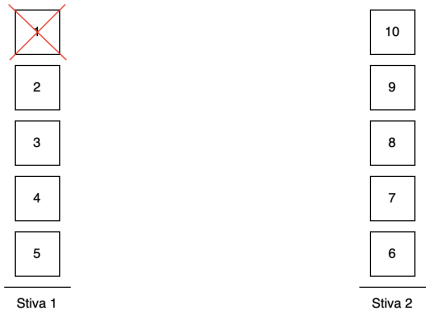
Operatia este foarte simpla: tot ce trebuie sa facem este sa adaugam x la varful stivei 2. Pe exemplul nostru, daca am adauga 10 in coada, stivele ar deveni:



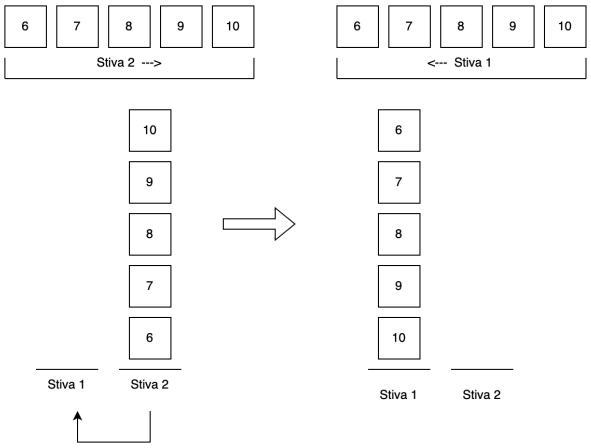
2.2.2 pop_front()

Aici avem 2 cazuri. Daca stiva 1 are cel putin un element, atunci tot ce trebuie sa facem este sa afisam si sa eliminam varful stivei 1.





Ce facem in schimb daca stiva 1 este goala? Solutia este sa 'rasturnam' stiva 2 in stiva 1.



Pentru a calcula eficienta algoritmului, observam ca orice element din coada este mai intai adaugat in stiva 2, apoi la o 'rasturnare' se va muta in stiva 1, apoi va fi eliminat, deci fiecare element din coada va aparea intr-un numar constant de operatii pe stiva.

Complexitatea algoritmului este $O(N)$.

Codul sursa aici

2.3 Problema Secventa6

Vom procesa elementele in ordine crescatoare de la stanga la dreapta. Sa presupunem ca suntem la o pozitie q. Numaram cate secvente bune se termina pe pozitia q. Observam ca putem sa clasificam subsecventele bune de forma $[p, q]$ (unde p este capatul din stanga, q este cel din dreapta egal cu pozitia curenta) in 2 grupe: cele cu $a[p] < a[q]$ si cele cu $a[p] >= a[q]$.

Ne vom folosi de o stiva care ne va ajuta sa determinam pentru fiecare pozitie q cea mai din dreapta pozitie p de la stanga lui q cu proprietatea ca $a[p] >= a[q]$. Algoritmul poate fi descris sumar astfel:

- ne aflam la o pozitie q si scoatem din stiva o pozitiile p cu $a[p] < a[q]$. O astfel de subsecventa $[p, q]$ va contine strict in interiorul ei numai valori mai mici strict decat capetele deci incrementam raspunsul cu 1.
- o data eliminata o pozitie din stiva, ne uitam sa vedem daca nu cumva in stiva era un grup de valori egale. Acestea nu mai pot forma impreuna cu q o subsecventa in care capetele sunt strict mai mari decat elementele din interiorul subsecventei, deci trebuie eliminate din stiva fara sa incrementam raspunsul
- daca in urma eliminarilor stiva a ramas nevida, atunci inseamna ca varful stivei va fi o pozitie p cu $a[p] >= a[q]$ si deci trebuie sa incrementam raspunsul cu 1
- adaugam pozitia q la stiva

```
int top = 0, answer = 0;
for (int i = 0; i < n; ++i) {
  int x = i + (value[i / 8192] ^ value[i % 8192]);

  while (top >= 1 and stack[top] < x) {
    ++answer;

    while (top >= 2 and stack[top] == stack[top - 1])
      --top;
    --top;
  }

  if (top >= 1)
    ++answer;
  stack[++top] = x;
}
```

Codul sursa aici

2.4 Problema Undo2

Problema pare mult mai complicata la prima vedere decat este. Observatia esentiala este ca la o stergere/readaugare de elemente (o operatie de tip 2 sau 3), nu este necesar sa stergem propriu zis elementele din stiva, ci doar sa modificam fictiv dimensiunea stivei si sa presupunem ca in stiva exista doar elementele care se afla pe pozitii mai mici sau egale cu dimensiunea fictiva a stivei. De asemenea, vom mai tine pentru fiecare valoare distincta cate un *std* :: *vector* in care retinem pozitiile din cadrul stivei unde se afla aceasta valoare. Simulam astfel operatiile:

- operatia 1: primul pas este sa aducem dimensiunea reala la cea fictiva eliminand varful stive cat timp $top > fictional\_stack\_size$, apoi crestem dimensiunea stivei cu 1 si adaugam elmentul x in stiva. De asemenea adaugam in vectorul corespunzator lui x pozitia pe care aceasta s-a inserat (adica dimensiunea stivei)

```
while (top > fict_size) {
  position[stack[top]].pop_back();
  --top;
}

stack[++top] = x;
position[x].push_back(top);
fict_size = top;
```

- operatia 2: scadem $fictional\_stack\_size$ cu y

```
fict_size -= y;
```

- operatia 3: crestem $fictional\_stack\_size$ cu z

```
fict_size += z;
```

- operatia 4: observam ca pozitiile corespunzatoare lui t din vectorul sau sunt ordonate crescator. Trebuie sa numaram cate din aceste pozitii exista cu adevarat in stiva (cu alte cuvinte sunt $<= fictional\_stack\_size$). Se va folosi o cautare binara.

```
out << std::upper_bound(std::begin(position[t]),
                        std::end(position[t]), fict_size)
    - std::begin(position[t]) << "\n";
```

Codul sursa aici

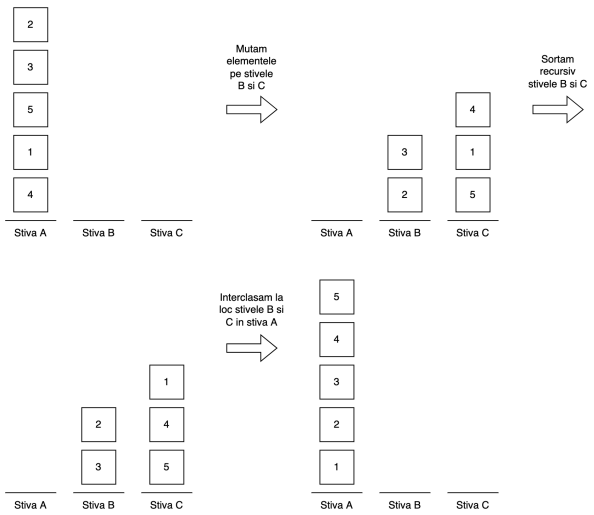
2.5 Problema Aranjare3

Vom folosi 2 stive ajutatoare. Algoritmul este relativ simplu dupa ce va vine ideea. Vrem sa implementam urmatoarea procedura recursiva:

```
void sort_stack(
    int n, int stack_a, int stack_b, int stack_c, bool increasing);
```

Aceasta functie va sorta primele n elemente din varful stivei A folosindu-ne de stivele ajutatoare B si C in ordine crescatoare/descrescatoare, in functie de valoarea *increasing*.

Pentru a realiza acest lucru, vom muta $[\frac{n}{2}]$ elemente de pe stiva A pe stiva B si restul de $n - [\frac{n}{2}]$ elemente de pe stiva A pe stiva C. Apoi vom sorta recursiv aceste elemnte de pe cele 2 stive in ordine inversa valorii *increasing* si apoi vom interclasa cele 2 stive B si C inapoi in stiva A, ca in exemplul de mai jos:



Complexitatea este $O(n\ log\ n)$ deoarece noi practic implementam algoritmul *"Merge Sort"* cu 3 stive.
Codul sursa aici

3 Probleme grele

3.1 Problema Campion

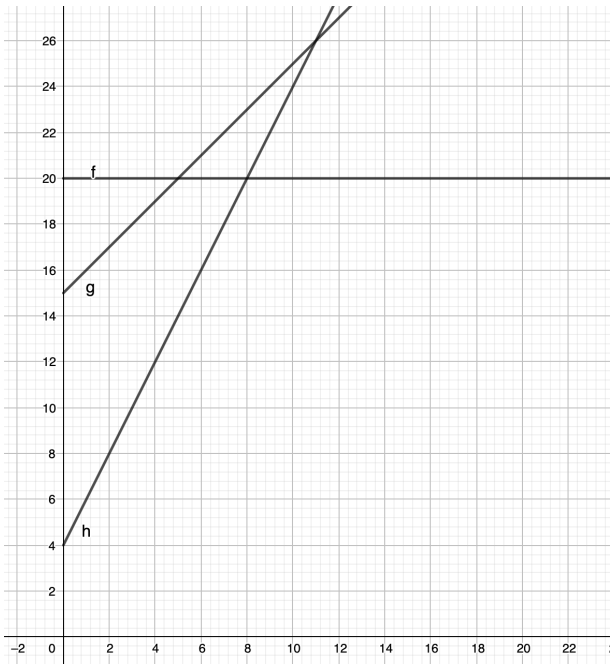
Primul pas este sa pastram doar campionii distincti (ca rating initial sau crestere a rating-ului).

```
int m = 0;
for (int i = 1, j = 1; i <= n; i = ++j) {
  while (j + 1 <= n and champion[i] == champion[j + 1])
    ++j;
  champion[++m] = champion[i];
  count[m] = j - i + 1;
}
```

Dupa sortam toti campionii descrescator dupa ratingul initial si in caz de egalitate descrescator dupa viteza cu care isi imbunatatesc ratingul.

```
std::sort(champion + 1, champion + n + 1,
[](const std::pair<int, int> &ch1, const std::pair<int, int> &ch2) {
  if (ch1.first != ch2.first)
    return ch1.first > ch2.first;
  return ch1.second > ch2.second;
});
```

Intuitia din spatele acestei sortari este ca in momentele de timp mici vor castiga campionii cu scorul initial mare, dar pe parcurs ce trece timpul, vor avea din ce in ce mai mult avantaj cei care progreseaza repede. O alta observatie esentiala este ca putem reprezenta progresul campionilor ca o dreapta in plan. Pentru exemplul din enunt avem:



Ultima observatie **foarte importanta** este ca daca avem 2 oameni $(R1, D1)$ si $(R2, D2)$ cu $R1 > R2$ dar $D1 < D2$, al doilea il va depasi la un moment de timp pe primul dar e in continuare posibil sa nu ajunga niciodata campion pentru ca e posibil sa existe o a 3-a persoana $(R3, D3)$ care sa o depaseasca pe a 2-a inainte ca aceasta sa apuce sa devine campion.

Vom adauga intr-o stiva candidatii la a deveni campioni intr-o stiva iar cand adaugam o noua persoana vom scoate din stiva persoanele care sunt eliminate de catre aceasta folosindu-ne de observatia de mai sus.

Initial campioni vor fi toti campionii care au ratingul initial maxim. Pentru fiecare persoana care poate deveni campion, vom retine si momentul de timp la care acesta a devenit campion. Vom adauga intr-o stiva candidatii care pot deveni campioni, initial adaugand doar persoana cu ratingul initial maxim si cea mai mare viteza de crestere a rating-ului (dintre cele cu rating initial maxim).

```
int top = 1;
stack[1] = 1;
int answer = count[1];

int start = 2;
while (start <= m and champion[start].first == champion[1].first) {
  answer += count[start];
  ++start;
}
```

In primul rand daca persoana curenta are atat rating-ul initial, cat si viteza de crestere a ratingului mai mica decat a persoanei din varful stivei, atunci nu va avea nicio sansa de a deveni campion, deci nu il adaugam in stiva.

In schimb, daca adaugam un nou campion in stiva, incercam sa scoatem cat mai multi campioni din varful stivei care 'ajung campioni dar prea tarziu'. Pentru asta trebuie sa vedem cand campionul curent va depasi campionul din varful stivei si trebuie sa comparam acest moment de timp cu momentul in care campionul din varful stivei ar ajunge pe primul loc. Daca acesta ajunge la un moment de timp prea mare (>= momentul de timp la care este depasit de noul campion), atunci trebuie de scos din stiva. Dupa toate eliminarile, vom adauga persoana curenta in stiva si vom calcula si momentul de timp la care ea va deveni campion.

```
for (int i = start; i <= m; ++i) {
  if (top > 0 and champion[i].second <= champion[stack[top]].second)
    continue;
  while (top > 0 and intersect(stack[top], i) < moment[stack[top]]) {
    answer -= count[stack[top]];
    --top;
  }
  moment[i] = intersect(stack[top], i);
  stack[++top] = i;
```

```
  answer += count[i];
}
```

Ultimul lucru care mai ramane de facut este sa scoatem din varful stivei toti oamenii care devin campioni la momente de timp $> T$

```
while (top > 0 and moment[stack[top]] > t) {
  answer -= count[stack[top]];
  --top;
}
```

Codul sursa aici

3.2 Problema Password1

Observatia cheie este ca daca exista un caracter c care apare de mai multe ori in B decat in A, atunci nu exista nicio anagrama a celui de-al 2-lea sir care sa se regaseasca in primul sir ca un subsir, deci afisam *impossible*. In schimb, daca orice caracter din A apare de cel putin la fel de multe ori in sirul B, atunci exista cel putin o solutie.

Vom parcurge primul sir si vom adauga caracterele intr-o stiva similar cu algoritmul de la problema Trompeta. Avem in stiva o parte din solutie construita (o parte din caracterele sirului B adaugate) si suntem la o pozitie p in primul sir si incercam sa adaugam un caracter c in solutie. La orice moment de timp, vom tine cate un vector de frecventa pentru fiecare din cele 2 siruri cu caracterele care au mai ramas (pentru sirul A avem vectorul de frecventa pentru caracterele ramase din intervalul $[p, |A|]$ si pentru sirul B avem vectorul de frecventa pentru caracterele pe care mai avem sa le adaugam in solutie). Cei 2 vectori de frecventa ii numim $count1$ respectiv $count2$. Cand adaugam un caracter c executam urmatorii pasi:

- verificam daca $count2[c] > 0$ (acest caracter poate face parte din caracterele ramase ale sirului B.
- daca da, atunci eliminam caractere din varful stivei cat timp stiva este nevida, $c < stack[top]$ si se pastreaza observatia de la inceput (in urma eliminarii caracterelor din stiva, numarul de caractere ramase in sirul B creste si implicit e posibil sa depaseasca numarul de caractere de acelasi tip din sirul A)
- dupa ce am trecut de caracterul c, decrementam cu 1 pe $count1[c]$ (acum nu mai face parte din caracterele ramase ale sirului A).

```
int top = 0;
for (int i = 1; i <= n; ++i) {
  if (count2[string1[i] - 'a'] > 0) {
    while (top > 0 and
           string1[i] < stack[top] and
           count1[stack[top] - 'a'] > count2[stack[top] - 'a']) {
      ++count2[stack[top] - 'a'];
      --top;
    }
    --count2[string1[i] - 'a'];
    stack[++top] = string1[i];
  }
  --count1[string1[i] - 'a'];
}
```

Codul sursa aici