Editorial RoAlgo Contest 10

29 Iunie 2024

Cuprins

1 Multumiri

Acest concurs nu ar fi putut avea loc fără următoarele persoane:

- Matei Benchea, Traian Mihai Danciu, Darius Hanganu, Rareș Hanganu, Radu Vasile, Ștefan Vîlcescu, autorii problemelor și laureați la concursurile de informatică și membri activi ai comunității RoAlgo;
- Alex Vasiluță, fondatorul și dezvoltatorul principal al Kilonova;
- Ștefan Alecu, creatorul acestui șablon LaTeXpe care îl folosim;
- Horia Andrei Boeriu, Matei Ionescu, Eduard-Lucian Pîrțac, testerii concursului, care au dat numeroase sugestii și sfaturi utile pentru buna desfășurare a rundei;
- Tudor Iacob și Ștefan Dăscălescu, coordonatorii rundei;
- Comunității RoAlgo, pentru participarea la acest concurs.

2 Martin

Autor: Rareș Hanganu

2.1 Soluția 1 - 60 puncte

Observație. $(a + b) \% k = (a \% k + b \% k) \% k$

Înlocuind $a \% k$ cu x, $b \% k$ cu y și $a + b$ cu s, rezultă că $s \% k = (x + y) \% k$.

2.1.1 Cod sursă

Soluție de 60

2.2 Soluția oficială

În plus, știm că $x \leq a$ si $y \leq b$, ceea ce înseamnă că $x + y \leq s$.

2.2.1 Cod sursă

Soluție de 100

3 Vecinul

Autor: Darius Hanganu

3.1 Soluția 1 - 30 de puncte

Putem rezolva această problemă folosind programare dinamică. Definim $dp[i]$ ca fiind suma maximă posibilă folosind primele i numere din șir. Răspunsul la problemă va fi $dp[n]$, unde n este lungimea șirului de numere. Iterăm prin toate elementele șirului. La fiecare pas, încercăm să includem elementul i într-o subsecvență împreună cu predecesorii săi, calculând cel mai mare divizor comun (gcd) al subsecvenței și actualizând $dp[i]$ cu suma maximă posibilă.

$$dp[i] = \max(\ \gcd(v[i], v[i-1], v[i-2], ..., v[i-j]) + dp[i-j-1]) \text{ pentru } j \text{ de la 1 la } i$$

Complexitatea de timp este $O(n^2)$.
Complexitatea de memorie este $O(n)$.

3.1.1 Cod sursă

Soluție de 30

3.2 Soluția 2 - 100 de puncte

Deoarece cmmdc scade cu cât avem mai multe numere, observăm că lungimea grupelor nu poate fi foarte mare, mai exact poate fi maxim 3. Astfel, optimizăm soluția anterioară, calculând dp[i] eficient

$$dp[i] = \max(\ \gcd(v[i], v[i-1]) + dp[i-2], \gcd(v[i], v[i-1], v[i-2]) + dp[i-3]\ )$$

Observăm că putem optimiza și memoria soluției, deoarece avem nevoie doar de 3 poziții din vectorul dp.
Complexitatea de timp este $O(n)$.
Complexitatea de memorie este $O(1)$

3.2.1 Cod sursă

Soluție de 100

4 Paranteze

Autor: Traian Mihai Danciu

4.1 Soluția oficială

Această problemă este una de tip constructivă. Are mai multe soluții, dar noi vom prezenta doar una dintre ele.
Mai întâi, vom pleca de la o observație foarte importantă, care va fi esențială pentru rezolvarea acestei probleme.

Observație. În primul rând, observăm că pentru un șir de forma `()()()...()()`, dacă avem n perechi de paranteze, numărul de subsecvențe corect parantezate este egal cu

$$\frac{n \cdot (n+1)}{2}$$

deoarece putem începe o subsecvență în oricare din parantezele deschise și să o încheiem în oricare din parantezele închise de la dreapta ei.

Astfel, vom construi numărul de secvențe necesare folosind un număr cât mai mic de șiruri de forma celor menționate mai sus. De exemplu, dacă avem nevoie de 14 subsecvențe parantezate corect, am putea avea `()()()()` urmat de `()()` și `()`. În acest caz, am putea pune un caracter auxiliar (de exemplu, o paranteză deschisă suplimentară) între aceste secvențe, însă lungimea

secvențelor ar fi mult prea mare și în cazul anumitor situații limită, nu am putea atinge limita de *n* caractere pe care o avem.
În cele ce urmează, vom introduce o nouă observație care poate fi descoperită fie folosind un program brute-force, fie analizând cazuri limită de mână.

Observație. Un șir de forma `((( ...)))` cu *x* perechi de paranteze, numărul de subsecvențe corect parantezate este egal cu *x* deoarece nu putem începe decât cu una din parantezele deschise și termina cu paranteza închisă de pe poziția simetrică.

Cu alte cuvinte, dacă avem două secvențe corect parantezate, *a* și *b*, cu proprietatea că *b* este inclus în *a*, numărul de subsecvențe corect parantezate va fi $secv(a) + secv(b)$, unde $secv(a)$ este numărul de subsecvențe corect parantezate ale șirului *a*.
De ce este această observație importantă? Acum, putem să ne folosim de ea pentru a evita folosirea acelor caractere suplimentare, deoarece putem pur și simplu să construim secvențele de paranteze unele în altele, având grijă la implementarea finală.
Acest lucru poate fi abordat în diverse moduri, dar implementarea pe care am avut-o procedează în felul următor:
Dacă avem o subsecvență de tip `()()() ...()()`, o vom poziționa înăuntrul ultimei perechi de forma `()` din șirul deja existent. Acest lucru se poate face fie recursiv, fie ținând în memorie numărul de paranteze închise de la finalul șirului, pentru a le plasa la final.
În cele din urmă, dacă lungimea șirului de caractere rezultat este mai mare decât *n*, nu vom avea soluție. Altfel, dacă avem mai puțin de *n* caractere, vom completa șirul cu paranteze deschise.
Pentru mai multe detalii recomandăm a vedea implementarea atașată.

4.1.1 Cod sursă

Soluție de 100

5 StrinK

Autor: Ștefan Vîlcescu

5.1 Subtaskul 1

În primul rând, trebuie să facem două observații:

Observație. Dacă trebuie să aleg un caracter diferit de mine cu indicele mai mare ca mine, este optim să mă duc la indicele j minim ($j > i$) astfel încat s_i si s_j sunt distincte;

Observație. Dacă mă pot duce la orice caracter cu indicele j ($j > i$), atunci este optim să mă duc la caracterul cu indicele $i + 1$;

Deci, putem face următorul algoritm:
Ne vom seta capătul stâng si capătul drept st, respectiv dr, și vom verifica dacă substringul $[st, dr]$ este strink
Cum verificăm acest lucru optim? Bazându-ne pe cele două observații de mai sus:
Dacă trebuie să gasim un caracter diferit de al nostru, ne vom baza pe prima observație, cautând liniar
Dacă trebuie să găsim orice caracter cu indice mai mare, ne vom baza pe a doua observație.
Complexitate: $O(N^3)$ timp, $O(N)$ memorie

5.2 Subtaskul 2

Pentru acest subtask trebuie să optimizăm soluția anterioară, bazându-ne pe o altă observație:

Observație. Dacă substringul $[st, dr]$ este strink, atunci și substringul $[st, dr + 1]$ este strink.

Deci, putem să optimizăm soluția astfel:
Ne vom fixa indicele stâng st, si vom căuta $dr - ul$ minim astfel încât $[st, dr]$ este strink. Dacă $dr \leq N$, atunci la răspuns vom adăuga $N - dr + 1$, altfel nu vom face nimic
Complexitate: O(N^2) timp, O(N) memorie

5.3 Subtask 3

Pentru acest subtask, compresăm stringul astfel incât nu există două caractere pe poziții adiacente egale. Ne vom ține două valori pe poziția i în string-ul compresat:
$carac_i$ = caracterul de pe poziția i
$size_i$ = cu cât am compresat indicele i
De exemplu, pentru stringul $s = aaabbccc$, cei 2 vectori noi vor arăta așa:
$carac_1 = a, carac_2 = b, carac_3 = c$
$size_1 = 3, size_2 = 2, size_3 = 3$
Fie size-ul string-ului compresat m
Acum ca avem acești vectori, subtaskul acesta presupune că $size_i > 1$, oricare ar fi i

Acum că avem asta, ne vom ține sume parțiale pe sufix, adica $sp_i =$

$$\sum_{j=i}^{m} size_j$$

Acum că avem acest vector, trebuie să vedem un anumit lucru:
Dacă sunt la indicele i în vectorul $size$, atunci dacă trebuie să aplic prima observație, mă voi duce la $size_{i+1}$, iar dacă trebuie să aplic a doua observație, trebuie să mă duc cu un indice în fața, dar cum $size_i > 1$, înseamnă că voi rămăne în același size
Deci, asta rezultă că pentru fiecare indice i din $size$, $dr - ul$ minim ar veni primul indice din $size_{i+K}$
Folosindu-ne de observația de la subtaskul 2, inseamnă că pentru fiecare indice i din $size_j$, $dr - ul$ minim este indicele p minim din $size_{j+K}$, deci vom avea $N - p + 1$ substringuri strink, iar $N - p + 1 = sp_{j+K}$, deci pentru $size_i$ vom avea $size_i \cdot sp_{i+K}$ substringuri bune
Astfel, răspunsul este:

$$\sum_{i=1}^{m-K} size_i \cdot sp_{i+K}$$

Complexitate: O(N) timp și memorie

5.4 Subtask 4

Putem optimiza soluția de la subtaskul precedent folosindu-ne de binary lifting. Fie p_i cel mai mare j, $j < i$ astfel încât $s_i \neq s_j$. Putem să ducem muchie între i și $p_i - 1$, problema rezumându-se la căutarea celui de-al k-lea strămoș al lui i. Soluția ar rula într-un $\mathcal{O}\left(N \cdot logN\right)$ evident.
Soluție de 52

5.5 Soluția oficială

Pentru soluția de 100 de puncte, trebuie să facem încă câteva observații:

Observație. Dacă $s_i = s_{i+1}$, atunci $dr - ul$ minim al lui i va fi și $dr - ul$ minim al lui $i + 1$

Observație. Dacă s_i și s_{i+1} sunt distince și s_i și s_{i-1} sunt distincte, atunci secvența de indicii ai lui $i - 1$ și a lui $i + 1$ vor arăta așa (pentru simplitate o să presupunem că stringul este format doar din litere distincte):
$i - 1, i, i + 1, i + 2, i + 3, ...$
$i + 1, i + 2, i + 3, ...$

Observația este că secvența de indici ai lui $i - 1$ si $i + 1$ diferă din primele două elemente, deci $dr - ul$ lui $i + 1$ va fi și $dr - ul$ lui $i - 1$, aplicând prima si a doua observație încă o dată

Observație. Dacă s_i și s_{i+1} sunt distincte și $s_i = s_{i-1}$, atunci trebuie să gasim indicele j minim ($j > i$) astfel încat $s_j = s_{j+1}$. Ne vom uita din nou la secvențele indicilor:
$i, i + 1, i + 2, i + 3, ..., j - 1, j, j + 1, ...$
$i + 1, i + 2, i + 3, ..., j - 1, j, ...$

Se poate vedea că până la indicele j, indicii rămân la fel. Astfel, vom împărți această observație în trei cazuri:
Cazul 1: Dacă j nu se află printe cei $2 * K$ indicii sau j este ultimul indice, atunci $dr - ul$ lui $i + 1$ va fi $dr - ul$ lui i aplicând prima observație încă o dată
Cazul 2: Dacă $j - (i + 1)$ este impar, atunci secvența de indici va fi aceasta:
$i, i + 1, i + 2, ..., j - 1, j, j + 1, ...$
$i + 1, i + 2, ..., j - 1, j, ...$
Cați indici are prima secvența până la j? O să aibă $j - i$ indicii
Cați indici are a doua secvența până la j? O să aibă $j - (i + 1)$ indicii

Deci, în prima secvență după j, indicele care ar trebui să apară este j aplicând a doua observație încă o dată, iar în a doua secvență ar trebui să apară j aplicând prima observație încă o dată. Doar că, indicele j aplicând a doua observație încă o dată este $j + 1$, deci după $j - i$ indici, următorul indice trebuie să fie $j + 1$ aplicând Obs. 1 încă o dată, iar asta rezultă același indice, urmând aceeași observație, până la final

Deci, secvența de indici ai lui i este egală cu secvența de indici ai lui $i + 1$, deci $dr - ul$ lui $i + 1$ este egal cu $dr - ul$ lui i

Cazul 3: Dacă $j - (i + 1)$ este par, atunci secvența de indici va fi aceasta:

$i, i + 1, i + 2, ..., j - 1, j, ...$

$i + 1, i + 2, ..., j - 1, j, j + 1, ...$

Câți indici are prima secvență până la j? O să aibă $j - i$ indici

Câți indici are a doua secvență pana la j? O să aibă $j - (i + 1)$ indici

Deci, în prima secvență după j, indicele care ar trebui să apară este j aplicând a doua observație încă o dată, iar în a doua secvență ar trebui să apară j aplicând prima observație inca o data. Doar că, indicele j aplicând a doua încă o dată este $j + 1$, deci după $j - i$ indici, următorul indice trebuie să fie $j + 1$ aplicând Obs. 1 încă o dată, pentru secvența a doua, iar pentru prima secvență, după $j - i + 2$ indici, următorul indice trebuie să fie $j + 1$ aplicând prima observație încă o dată, iar asta rezultă același indice, urmând aceeași observație, până la final

Deci, secvența de indici ai lui i diferă prin 2 indici ai lui $i + 1$, deci $dr - ul$ lui $i + 1$ este egal cu $dr - ul$ lui i, aplicând prima și a doua observație încă o dată

Bun, acum că știm toate observațiile, cu ce ne ajută? Ne vom ține 3 vectori:

poz_i=indicele j minim $(j > i)$ astfel încat s_i și s_j sunt distincte;

$ultim_i$=indicele j minim $(j > i)$ astfel încat $s_j = s_{j+1}$;

$rmin_i$=$dr - ul$ lui i;

Acum că avem acești vectori, vom face următorul algoritm:

Pentru $i = 1$ și $i = 2$, vom aplica algoritmul descris la al doilea subtask pentru a afla $rmin_1$ și $rmin_2$
Pentru fiecare i de la 3 până la $N - 2 \cdot K + 1$, vom face astfel:
Aplicăm a patra observație, dacă condiția este adevarată o aplicăm, dacă nu este adevarată aplicăm a cincea observație, dacă condiția este adevarată, iar dacă nici condiția aceasta nu este adevarată, aplicăm a șasea observație
Răspunsul va fi

$$\sum_{i=1}^{N-2\cdot K+1} N - rmin_i + 1$$

Complexitate: $O(N)$ timp și memorie
Atenție: Soluția oficială scapă de vectorul rmin, adunând la răspuns incremental. De asemenea, în soluția oficială vectorul începe de la 0, deci se va aduna $N - rmin_i$ înloc de $N - rmin_i + 1$

5.5.1 Cod sursă

Soluție de 100

6 Arborel

Autor: Matei Benchea

6.1 Soluția de 20 puncte

Date fiind valorile mici ale lui N și Q, pentru fiecare întrebare se poate parcurge subarborele nodului și număra efectiv câte noduri au valori în intervalul dat.
Complexitate: $O(N \cdot Q)$

6.1.1 Cod sursă

Soluție de 20

6.2 Soluția oficială

În primul rând, vom liniariza arborele prin intermediul unei pargurgeri euler (fie tin_i - timpul de intrare al nodului i și $tout_i$ - timpul de ieșire al nodului i). Astfel, o întrebare pe subarborele nodului i se va transforma într-o intrebare pe intervalul $[tin_i, tout_i]$.

Observație. Fie $F_{x,st,dr}$ = numărul de nodrui din subarborele nodului x cu

valori cuprinse în intervalul $[st, dr]$. Avem:

$$F_{x,st,dr} = F_{x,1,dr} - F_{x,1,st-1}$$

Putem astfel să împărțim întrebările noastre în întrebări la care putem răspunde mai ușor, caracterizate doar de nodul x și valoarea y ($F_{x,1,y}$).

Pentru a putea găsi eficient răspunsul la întrebările nou formate vom folosi următoarea abordare: vom sorta valorile nodurilor și întrebările crescător după valoarea lor. Le parcurgem și atunci când întâlnim un nod marcăm poziția sa în parcurgere ca fiind activată, iar răspunsul la o întrebare (să spunem pe nodul i) îl constituie numărul de poziții activate din intervalul $[tin_i, tout_i]$. Faptul că am sortat crescător valorile nodurilor și întrebările ne garantează că atunci când vom număra pozițiile activate dintr-un interval pentru a răspunde la o întrebare, acestea vor corespunde unor noduri a căror valoare este ≤valoarea întrebării.
Pentru a manageria eficient activarea pozițiilor și interogarea pe un anumit interval se poate utiliza o structură de date cum ar fi un arbore indexat binar sau un arbore de intervale.
Complexitate: $O((N + 2 \cdot Q)log(N + 2 \cdot Q) + (N + 2 \cdot Q)logN)$

6.2.1 Cod sursă

Soluție de 100

6.3 Soluție alternativă

Se poate obține de asemenea punctaj maxim cu soluții care se folosesc de tehnica small to large.

6.3.1 Cod sursă

Soluție de 100

7 Subșiruri

Autor: Traian Mihai Danciu

7.1 Soluții parțiale

7.1.1 Subtaskul 1

Pentru fiecare întrebare, vom menține $dp_{i,j}$, care va reprezenta de câte ori apare $a_1 a_2 .. a_j$ în intervalul $[i, l]$. Recurența este aceasta:
$dp_{i,0} = 1$, pentru $l - 1 \leq i \leq r$
$dp_{l-1,j} = 0$, pentru $j > 0$
$dp_{i,j} = dp_{i-1,j} + dp_{i-1,j-1}$, pentru $l \leq i \leq r, j > 0$ și $b_i = a_j$
$dp_{i,j} = dp_{i-1,j}$, pentru $l \leq i \leq r, j > 0$ și $b_i \neq a_j$
Complexitate: $O(q \cdot n \cdot k)$ timp, $O(n \cdot k)$ memorie

Cod sursă

Soluție de 10 puncte

7.1.2 Subtaskul 2

Să ne uităm la soluția pentru subtaskul 1. Avem două observații:

Observație. În primul rând, observăm că $dp_{i,j}$ depinde doar de $dp_{i-1,p}$ pentru anumiți p.

Observație. În al doilea rând, doar unul dintre $dp_{i,j}$ diferă de $dp_{i-1,j}$, deoarece elementele șirului a sunt distincte. Așa că putem calcula poz_c care reprezintă poziția în șirul a pe care se află caracterul c. De asemenea, $poz_c = 0$ dacă c nu apare în a. Deci doar $dp_{i,poz_{a_i}} \neq dp_{i-1,poz_{a_i}}$ din linia i.

Din aceste observații obținem că noi, la fiecare i vom ține doar un vector dp de $k+1$ elemente, unde dp_j care va reprezenta $dp_{i,j}$. Și la linia i, noi vom actualiza doar dp_{poz_i}, dacă $poz_i \neq 0$. Complexitate: $O(q \cdot n)$ timp, $O(n)$ memorie

Cod sursă

Soluție de 22 de puncte

7.1.3 Subtaskul 3

Cod sursă

Șirul a are un singur caracter, așa că putem menține niște sume parțiale, pentru care sp_i reprezintă de câte ori apare caracterul din a în primele i elemente. La final, răspunsul va fi $sp_r - sp_{l-1}$, pentru fiecare întrebare. Complexitate: $O(n)$ timp, $O(n)$ memorie
Soluție de 5 puncte

7.1.4 Subtaskul 4

Din moment ce problema pune o întrebare pe un interval, una din idei este să folosim un arbore de intervale.

Ce ținem în fiecare nod?

Noi vom dori ca în fiecare nod să avem răspunsul pentru tot intervalul acoperit de nod. Dar cum calculăm răspunsul pentru tot intervalul? Avem nevoie de răspunsul pentru un prefix al intervalului și pentru un sufix al intervalului. Dar, totuși, cum calculăm un prefix al intervalului? Punând aceste întrebări obținem că în fiecare nod trebuie să ținem răspunsul pentru un subinterval al intervalului acoperit de nod.
Fie $v_1, v_2, .., v_p$ valorile din intervalul acoperit de nod. Noi vom ține un $dp_{i,j}$ (unde $0 \leq i < j \leq k$) care va reprezenta de câte ori apare $a_{i+1}, .., a_j$ ca subșir în $v_1, v_2, .., v_p$.

Cum unim două noduri?

Dacă $dpnod_{i,j}$ reprezintă $dp_{i,j}$ pentru intervalul acoperit de nodul curent, $dpstanga_{i,j}$ reprezintă $dp_{i,j}$ pentru intervalul acoperit de fiul din stanga al nodului, și $dpdreapta_{i,j}$ reprezintă $dp_{i,j}$ pentru intervalul acoperit de fiul din dreapta al nodului, atunci $dp_{i,j} = \sum_{p=i}^{j} dpstanga_{i,p} * dpdreapta_{p,j}$
Așa că combinarea a două noduri are complexitate $O(k^3)$, ceea ce înseamnă că construirea arborelui de intervale are complexitate $O(n \cdot k^3)$

Răspunderea la întrebări

Vom alege cele aproximativ $\log n$ noduri interne disjuncte, care reunite să aibă ca rezultat intervalul $[l, r]$. Dacă le combinăm cum am scris mai sus, am obține o complexitate de $O(k^3 \cdot \log n)$ pentru fiecare întrebare.
Observație. La final, avem nevoie doar de $dp_{0,p}$. Așa că vom reține $pref_i = dprezultat_{0,i}$, iar răspunsul final va fi $dprezultat_k$.
Când adăugăm un nod intern, avem $pref_i += \sum_{j=0}^{i} pref_j \cdot dpnod_{j,i}$. Un detaliu de implementare este că vom calcula noile valori din $pref$ în ordine

inversă, pentru a nu număra anumite valori de mai multe ori. Astfel, când adăugăm un nod intern, vom obține o complexitate de $O(k^2)$. Așa că, complexitatea finală este $O(n \cdot k^3 + q \cdot k^2 \cdot \log n)$ timp, $O(n \cdot k^2)$ memorie.

Cod sursă

Soluție de 28 de puncte

7.2 Soluția oficială

Vom începe cu o observație.

Observație. Noi nu avem actualizări, avem doar întrebări. Iar în soluția de la subtaskul 4, avem o programare dinamică în fiecare nod. Aceste două lucruri ne duc înspre o soluție cu divide et impera.

Vom folosi o tehnică de forma ”divide and conquer by queries”. Mai multe detalii despre această tehincă puteți găsi aici.
Această tehincă funcționează, deoarece carcaterele din șirul a sunt distincte, ceea ce înseamnă că actualizarea vectorului $dp_{i,j}$ când adăugăm un element la final sau la început se poate face în complexitate de timp $O(k)$. Pentru mai multe detalii asupra acestui lucru vedeți implementarea mai jos.
Deoarece tehnica divide and conquer by queries are complexitate $O(n \cdot \log n)$, și adăugarea unui element la final sau la început are complexitate $O(k)$, complexitatea totală va fi $O(n \cdot \log n \cdot k + q)$ timp, $O(n \cdot k + q)$ memorie.

7.2.1 Cod sursă

Soluție de 100 de puncte