DESCRIEREA SOLUȚIILOR, OLIMPIADA NAȚIONALĂ DE INFORMATICĂ, CLASA A X-A

Munte

Propusă de: Prof. Szabó Zoltan - Inspectoratul Școlar Județean Mureș
Stud. Theodor-Gabriel Tulbă-Lecu - Universitatea Politehnica București

Toate demonstrațiile lemelor folosite pentru demonstrarea corectitudinii acestei probleme se pot găsi la finalul acestui articol în secțiunea Demonstrații problema munte.

Observații. În primul rând, pentru a rezolva problema trebuie să înțelegem ce efect au transformările `swap` și `double_swap` asupra unei permutari:

- Operația `swap` schimbă între ele două elemente egal distanțate de capetele permutării
- Operația `double_swap` schimbă între ele două elemente arbitrare, dar odată cu acestea schimbă si elementele corespunzătoare egal depărtate de capetele permutării ale acestor elemente.

Astfel, putem observa următorul invariant: Două elemente egal distanțate de capetele permutării a_i și a_{n-i+1} vor rămâne egal distanțate indiferent de ce operații efectuăm asupra permutării. Vom numi astfel de numere *numere perechi*.

De asemenea, în cazul în care permutarea are lungime impară, atunci elementul $a_{\frac{n+1}{2}}$ este un punct fix, adică nu putem să îi modificăm poziția cu niciuna din cele două tipuri de operații. Acest caz este identic cu cel în care permutarea are lungime pară și nu îl vom folosi pentru demonstrația soluției.

În continuare, pentru a oferi un mod vizual de întelegere a demonstrațiilor si pentru a vizualiza care sunt numerele perechi vom reprezenta o permutare astfel:

În cazul în care $n = 2 \cdot k$ este par:

$$\begin{array}{|ccccc|} \mathrm{a}_1 & \mathrm{a}_2 & \dots & \mathrm{a}_{k-1} & \mathrm{a}_k \\ \mathrm{a}_n & \mathrm{a}_{n-1} & \dots & \mathrm{a}_{n-k+2} & \mathrm{a}_{n-k+1} \end{array}$$

Permutare munte. O permutare munte este o permutare care până la un moment dat este strict crescătoare, iar mai apoi devine strict descrescătoare. Vom nota acesată poziție în care se face tranziția între cele două monotonii vf, unde $1 < vf \le k$. Dacă $vf > k$, atunci putem aplica o transformare swap pe toate valorile de la 1 la k, acest proces are ca efect inversarea primei linii cu cea de-a doua.

Deci, fără pierderea generalității, o permutare munte arată astfel:

$$\begin{array}{|ccccccccccccc|} a_1 & < & a_2 & < & \dots & < & a_{n-vf+1} & < & \dots & < & a_{k-1} & < & a_k \\ & & & & & & & & & & & & \wedge \\ a_n & < & a_{n-1} & < & \dots & < & a_{vf} & > & \dots & > & a_{n-k+2} & > & a_{n-k+1} \end{array}$$

Permutarea munte minim lexicografică (PMML). Dintre toate permutările munte care se pot obține dintr-o anumită permutare, există o permutare unică care este minim lexicografică.

O permutare a este mai mică lexicografică decat o altă permutare b dacă si numai dacă $\exists k \in \{1, 2, \dots n\}$ astfel încât $a_i < b_i$ și $a_j = b_j$, $\forall\ i \in \{1, 2, \dots, i-1\}$.

test

Lema 1. *PMML respectă propriatatea că $a_i < a_{n-i+1}$ $\forall\ i \in \{1, 2, \dots, k\}$ și poate fi reprezentată grafic astfel:*

$$\begin{array}{|ccccccccccccc|} a_1 & < & a_2 & < & \dots & < & a_{n-vf+1} & < & \dots & < & a_{k-1} & < & a_k \\ \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & \wedge \\ a_n & < & a_{n-1} & < & \dots & < & a_{vf} & > & \dots & > & a_{n-k+2} & > & a_{n-k+1} \end{array}$$

Corolar 1. *Orice permutare pentru care se poate ajunge folosind doar operații de* `swap` *și* `double_swap` *la o permutare munte poate fi transformată în complexitate $\mathcal{O}(n)$ sau $\mathcal{O}(n \cdot \log_2 n)$ în PMML.*

Calcularea numărului de permutări munte. Datorită Corolar 1:, știm că putem obține din permutarea inițială PMML. Totodată, cum în Lema 1: am demonstrat că din orice permutare munte se poate ajunge la PMML doar prin operații de swap și putem observa că operația de swap este propria sa inversă (dacă aplicăm swap de 2 ori pe aceeasi coloană ne întoarcem la starea inițială), și reciproca este adevărată: putem ajunge din PMML la orice permutare munte ce poate fi obținută folosind doar operații de swap .

O altă observație este că două operații de swap pe două poziții distincte $i \neq j$, sunt independente și putem alege pentru fiecare dacă o aplicăm sau nu. Astfel, numărul de permutări munte ce poti fi obținute va fi 2^C, unde C reprezintă numărul maxim de operații de swap distincte pe care le putem aplica asupra PMML astfel încât rezultatul să rămână o permutare munte.

$$\begin{array}{ccccccccccccccccccc} a_1 & < & a_2 & < & \dots & < & a_{i-1} & < & a_i & < & \dots & < & a_{n-vf+1} & < & \dots & < & a_{k-1} & < & a_k \\ \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & \wedge \\ a_n & < & a_{n-1} & < & \dots & < & a_{n-i} & < & a_{n-i+1} & < & \dots & < & a_{vf} & > & \dots & > & a_{n-k+2} & > & a_{n-k+1} \end{array}$$

Din desenul anterior putem observa că pentru a putea aplica swap pe o anumită poziție $1 \leq i \leq k$, trebuie fie ca $i = 1$ sau $a_i > a_{n-i}$.

Complexitatea algoritmului este dată de complexitatea aducerii permutării inițiale la PMML și se arată in Corolar 1: că poate fi implementată în $\mathcal{O}(n)$ sau $\mathcal{O}(n \cdot \log_2 n)$. Această soluție obține 100 de puncte.

ChangeMin

Propusă de: Stud. Popa Bogdan-Ioan - Universitatea din București

Cerința 1. Se parcurge șirul de la final la început și se introduc elementele rând pe rând într-o stivă. Înainte de a introduce un element A_i în stivă vom scoate din vârful stivei acele valori care sunt mai mici sau egale cu A_i. Se observă că parcurgând stiva din vârful ei în jos vom obține șirul de valori pe care variabila *min* (din algoritmul în pseudocod prezentat) le va lua. Astfel, după inserarea lui A_i în stivă este suficient să adunăm la *cnt* lungimea stivei. Complexitate $\mathcal{O}(N)$.

Cerința 2. Pentru a rezolva cerința 2 va trebui mai întâi să obținem valoarea lui *cnt* în urma rezolvării cerinței 1. Mai departe vom face o parcurgere asemănătoare cu cea de la cerința 1. Aflându-ne la indicele i după ce am făcut inserarea lui A_i in stivă vom scadea din *cnt* pe L, lungimea stivei S. Pentru a putea calcula variabila *score* va trebui să ținem încă două informații:

$sumCoef = 1 \cdot A_{S_L} + 2 \cdot A_{S_{L-1}} + \dots + L \cdot A_{S_1}$, care ne va ajuta la calcularea lui *score* și $sumAll = A_{S_L} + A_{S_{L-1}} + \dots + A_{S_1}$ care ne va ajuta să calculăm atât *score* cât și *sumCoef*. Având aceste valori calculate vom putea aduna la *score* valoarea $cnt \cdot sumAll + sumCoef$

Dragon Fruit

Propusă de: Stud. Coroian David-Nicolae - Delft University of Technology
Asist. Drd. Andrei-Costin Constantinescu - ETH Zürich
Stud. Cotor Andrei - Universitatea „Babeș-Bolyai” Cluj-Napoca

Subtask-urile 1, 2 și 3. Pentru date de intrare suficient de mici, putem număra toate cazurile posibile, le păstrăm pe cele cu suma lungimilor intervalelor minimă ce au suma K și numărăm câte astfel de intervale exsită.

Pentru fiecare interval este nevoie de fixarea celor două capete, deci se va obține un algoritm cu complexitate $\mathcal{O}(N^{2S})$.

Această soluție obține 20 de puncte.

Subtask-ul 4. Pentru cazul în care avem un singur interval, putem să calculăm pentru fiecare capăt stânga al intervalului unde se va afla capătul dreapta astfel încât suma elementelor să fie K.

Pentru a realiza acest lucru se poate utiliza un algoritm de tip *two pointers* în complexitate temporală $\mathcal{O}(N)$.

Acest subtask obține 10 puncte.

Subtask-ul 5. Pentru cazul în care avem maxim două intervale, putem să procedăm astfel:

- setăm capetele celui de-al doilea interval pe care il notăm cu $(i, j), i \leq j$
- dacă notăm suma elementelor de pe acest interval cu s, atunci tot ce mai trebuie să facem este să numărăm câte intervale $(k, l), k \leq l < i$ de lungime minimă au suma $K - s$.
- pentru a putea calcula acest lucru vom crea un tablou unidimensional fr_x = numărul de intervale de sumă x de lungime minimă. Acum răspunsul pentru întrebarea de la pasul anterior se va afla în fr_{K-s}.
- când trecem de la i la $i+1$ cu capătul dreapta al intervalului al doilea, va trebui să adăugăm în fr toate intervalele nou apărute, adică toate intervalele de forma $(k, i), k \leq i$ astfel încât suma pe interval să fie $\leq K$.
- când calculăm răspunsul pentru un pas vom actualiza soluția cea mai bună (de lungime minimă si câte astfel de soluții există).

Această soluție are complexitate temporală de $\mathcal{O}(N^2)$.

Soluție oficială. Pentru a rezolva problema vom folosi tehnica *programării dinamice*.

Pentru a găsi o astfel de soluție va trebui să observăm care sunt parametrii de care depinde o posibilă soluție:

- care sunt elementele care s-au luat în considerare pentru soluția parțială;
- câte intervale am folosit pentru a crea soluția si care este starea ultimului interval (daca este încă deschis sau dacă îl considerăm terminat)l
- care este suma elementelor soluției.

Astfel, vom definii următorul tablou în care vom stoca atât lungimea minimă a unei soluții (*stare.lungime*) cât si numărul de astfel de soluții (*stare.cnt*) sub forma unei perechi de numere:

$dp_{i,j,k,l}$ = lungimea minimă a unei soluții și numărul de de astfel de soluții considerând doar elementele de pe pozițiile de la 1 la i ce au suma k și sunt împărțite în j intervale. Variabila l poate lua valorile 0 sau 1 și va reține dacă ultimul interval din cele j s-a terminat (0) sau încă il putem extinde (1).

Starea inițială a dinamicii este $dp_{0,0,0,0} = (0, 0)$, adică dacă nu considerăm niciun element, niciun interval, suma fiind 0, există 0 soluții, lungimea celei mai bune astfel de soluții fiind 0. Toate celelalte stări din matrice vor fi setate cu $(\infty, 0)$ pentru a semnala că nu există nicio astfel de soluție găsită încă.

Pentru a calcula răspunsul reuniunii a două stări din dinamică într-un mod simplu vom definii următoarea funcție:

Algorithm 1 combine

Intrare: două stări $stare_1, stare_2$
Ieșire: rezultatul combinării celor două stări

```
dacă stare_1.lungime < stare_2.lugime atunci
    return stare_1
altfel dacă stare_1.lungime > stare_2.lugime atunci
    return stare_2
altfel
    return (stare_1.lungime, (stare_1.cnt + stare_2.cnt) mod 10^9 + 7)
sfârșit dacă
```

În continuare vom analiza recurența acestei dinamici:

- dacă se închide intervalul curent:
$$dp_{i,j,k,0} = combine(dp_{i,j,k,0}, dp_{i-1,j,k,1})$$
- dacă se păstrează închis intervalul curent:
$$dp_{i,j,k,0} = combine(dp_{i,j,k,0}, dp_{i-1,j,k,0})$$
- dacă se continuă cu intervalul curent (se adaugă 1 element la lungime):
$$dp_{i,j,k,1} = combine(dp_{i,j,k,1}, (dp_{i-1,j,k-v_i,0}.lungime + 1, dp_{i-1,j,k-v_i,0}.cnt))$$
- dacă se deschide un nou interval (se adaugă 1 element la lungime):
$$dp_{i,j,k,1} = combine(dp_{i,j,k,1}, (dp_{i-1,j-1,k-v_i,0}.lungime + 1, dp_{i-1,j-1,k-v_i,0}.cnt))$$
$$dp_{i,j,k,1} = combine(dp_{i,j,k,1}, (dp_{i-1,j,k-v_i,0}.lungime + 1, dp_{i-1,j,k-v_i,0}.cnt))$$

Răspunsul va fi reprezentat de reuniunea soluțiilor din stările de forma $dp_{N,j,K,l}$ pentru orice $1 \le j \le S$ și $l \in \{0, 1\}$, întrucât vrem să luăm în considerare toate cele N elemente, iar suma soluției să fie exact K.

Complexitatea temporală a acestei soluții este $\mathcal{O}(N \cdot S \cdot K)$ și obține scorul maxim.

Ne mai rămâne o singură problemă de rezolvat. Complexitatea spațială a acestei soluții este $\mathcal{O}(N \cdot S \cdot K)$, care pentru testele maximale va depăși limita de memorie pentru problemă. Însă putem reduce memoria făcând următoarea observație:

Toate stările $dp_{i,j,k,l}$, pot fi calculate utilizând doar stări de forma $dp_{i-1,j',k',l'}$, deci putem să reținem doar ultimele două *rânduri* din matrice pentru a calcula răspunsul. Astfel, complexitatea spațială este redusă la $\mathcal{O}(S \cdot K)$.

Echipa

Problemele pentru această etapă au fost pregătite de:

- prof. Szabó Zoltan - Inspectoratul Școlar Județean Mureș
- prof. Boca Alina Gabriela – Colegiul Național de Informatică ,,Tudor Vianu" București
- prof. Popescu Carmen - Colegiul Național ,,Gheorghe Lazăr" Sibiu
- prof. Moț Nistor - Școala Gimnazială ,,Dr. Luca" Brăila
- Stud. Tulbă-Lecu Theodor-Gabriel - Universitatea Politehnica din București
- Stud. Cotor Andrei - Universitatea ,,Babeș-Bolyai" Cluj-Napoca
- Stud. Dăscălescu Ștefan Cosmin - Universitatea din București
- Stud. Popa Bogdan-Ioan - Universitatea din București
- Stud. Râpeanu George-Alexandru - Universitatea ,,Babeș-Bolyai" Cluj-Napoca
- Stud. Coroian David-Nicolae - Delft University of Technology
- Stud. Popescu Ioan - Universitatea Politehnica din București
- Asist. Drd. Andrei-Costin Constantinescu - ETH Zürich

Appendix A. Demonstrații problema munte

Lema 1: Vrem să demonstrăm că PMML respectă propriatatea: $a_i < a_{n-i+1}\ \forall\ i \in 1, 2, \ldots, k$ și arată astfel:

$$\begin{array}{ccccccccccccc} a_1 & < & a_2 & < & \ldots & < & a_{n-vf+1} & < & \ldots & < & a_{k-1} & < & a_k \\ \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & \wedge \\ a_n & < & a_{n-1} & < & \ldots & < & a_{vf} & > & \ldots & > & a_{n-k+2} & > & a_{n-k+1} \end{array}$$

Demonstrație:

Vom presupune că există PMML cu un $i \in \{1, 2, \ldots, k\}$ astfel încât $a_i > a_{n-i+1}$ și $a_j < a_{n-j+1}\ \forall\ 1 \le j < i$.

Pentru $i > n - vf + 1$, intrucât subșirul $a_{n-vf+1}, \ldots, a_{vf+1}$ este sortat crescător nu poate exista un i, astfel încât $a_i > a_{n-i+1}$ și să fie respectată proprietatea de munte.

Pentru $1 \le i < n - vf + 1$ premutarea arată astfel:

$$\begin{array}{ccccccccccccccccccc} a_1 & < & a_2 & < & \ldots & < & a_{i-1} & < & a_i & < & \ldots & < & a_{n-vf+1} & < & \ldots & < & a_{k-1} & < & a_k \\ \wedge & & \wedge & & \wedge & & \wedge & & \vee & & ? & & ? & & \wedge & & \wedge & & \wedge \\ a_n & < & a_{n-1} & < & \ldots & < & a_{n-i} & < & a_{n-i+1} & < & \ldots & < & a_{vf} & > & \ldots & > & a_{n-k+2} & > & a_{n-k+1} \end{array}$$

Din figura anterioară putem observa că $a_i > a_{n-i+1} > a_{n-i}$ și $a_{n-i+1} > a_{n-i} > a_{i-1}$, deci putem inversa a_i cu a_{n-i+1} folosind o operație de swap si se va păstra proprietatea de munte, iar pentru a pătsra proprietatea de munte și pentru restul sirului, vom aplica swap si pe toate perechile de la $i + 1$ la k asfel:

$$\begin{array}{ccccccccccccccccccc} a_1 & < & a_2 & < & \ldots & < & a_{i-1} & < & a_{n-i+1} & < & \ldots & < & a_{vf} & > & \ldots & > & a_{n-k+2} & > & a_{n-k+1} \\ \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & ? & & ? & & \vee & & \vee & & \vee \\ a_n & < & a_{n-1} & < & \ldots & < & a_{n-i} & < & a_i & < & \ldots & < & a_{n-vf+1} & < & \ldots & < & a_{k-1} & < & a_k \end{array}$$

Noua permutare obținută este tot o permutare munte și este mai mică lexicografic decât permutarea inițială pentru că $a_i > a_{n-i+1}$, deci permutarea inițială nu putea fi PMML.

Corolar 1: Utilizând algoritmul de la Lema 1: de mai multe ori, putem demonstra prin inducție matematică că orice permutare munte, poate fi transformată într-o PMML utilizând doar operații de tip swap , deoarece la fiecare pas al algoritmului, prefixul de perechi $a_i > a_{n-i+1}$ crește în lungime cu cel puțin 1.

Mai mult, întrucât știm că pentru orice permutare posibilă a datelor de intrare a problemei există cel putin o permutare munte ce poate fi obținută, atunci putem ajunge la PMML în următorul mod:

- realizăm operații de swap pentru toate perechile cu $a_i > a_{n-i+1}$, unde $i \in \{1, 2, \ldots, k\}$ – acest lucru ne garantează proprietatea de minimalitate lexicografică
- sortăm crescător folosind operații de `double_swap` elementele de la a_1 la a_k. Datorită faptului că se garantează existența a cel puțin unei permutări munte, celelalte elemente vor fi sortate sortate crescător de la a_{n-k+1} la a_{vf} și descrescător de la a_{vf} la a_n . – acest lucru ne garantează proprietatea de munte

Complexitatea acestui algoritm este dată de sortarea numerelor pereche. Luând în considerare faptul că sirul este o permutare, numerele sunt cuprinse între 1 și n, astfel se pot obține complexitățile:

- $\mathcal{O}(n^2)$, dacă se folosesc algoritmi suboptimi precum Bubble Sort.
- $\mathcal{O}(n \cdot \log(n))$, dacă se folosesc algoritmi optimi de sortare prin comparare precum Merge Sort.
- $\mathcal{O}(n)$ dacă se folosește Count Sort.