Societatea pentru Excelență și Performanță în Informatică

Articolul educațional al anului școlar 2020-2021

Abordarea problemelor ad-hoc

Cuvânt înainte

Nu este simplu să redactezi soluția unei probleme de informatică; autorii și cititorii confirmă deopotrivă. Aceștia din urmă ar putea acuza în mod particular faptul că deși află din lectura soluției ce idee ar fi trebuit să găsească în timpul concursului, nu sunt mult mai lămuriți *cum* ar fi putut să ajungă la ea în mod practic. Acest efect, ar argumenta în continuare cititorii noștri, e în special vizibil în cazul așa-numitelor probleme *ad-hoc*; probleme mai puțin tehnice, în care accentul cade nu pe cunoștințe, ci pe creativitate.

Ne-am regăsit și noi deseori printre acești cititori ipotetici, iar din acest motiv am scris acest articol. Articolul prezintă soluțiile a patru probleme "ad-hoc" (din punctul nostru de vedere) de dificultate variată (din punctul.. ați înțeles). Articolul este mai lung decât v-ați așteptat, fiindcă ne-am propus nu doar să descriem o soluție, ci să urmărim de asemenea firul de gândire care ne-a dus la ea. Acest fir poate fi întortocheat; ne poate duce prin alte variante ale problemei, ne poate duce chiar pe piste false inițial, dar în final, sperăm noi, ne va da o imagine mai veridică a procesului de rezolvare al unei probleme în practică.

Am descris ce ne propunem să menționăm și vrem să subliniem și ce *nu* ne propunem. Nu vrem să sugerăm că abordările prezentate în acest articol sunt singurele posibile pentru problemele în cauză (sau că sunt cele mai "eficiente" în vreun sens). Persoane diferite vor găsi soluții diferite, vor motiva diferit soluții aparent foarte similare sau nu își vor motiva soluțiile defel (uneori ideile "îți vin" și atât). Dar la finalul zilei credem că există multă valoare în a încerca să demistificăm acest proces atunci când este posibil și că soluțiile prezentate în acest articol vă pot servi ca punct de plecare în dezvoltarea propriilor voastre tehnici în acest sens.

Dorim să mulțumim tuturor celor care au contribuit la realizarea acestei lucrări, începând cu Alexandru Petrescu care a avut inițiativa de a crea "articolul bombă care să rupă norii". Mulțumim autorilor care și-au pus pe hârtie talentele algoritmice: Radu Muntean, cel ce a propus rezolvările ultimelor două cerințe, precum și Alexandru Băbălau și Bogdan Pop, care au alcătuit descrierile primelor două, respectiv a celei de-a doua probleme. Suntem recunoscători și coordonatoarei echipei, Amalia Rebegea, dar și celor care au revizuit și editat conținutul, aducându-l la forma în care îl vedeți astăzi: Mihai Calancea, Andra-Elena Mircea și Ștefan Manolache.

Vă urăm lectură plăcută!

Problema 1. Corect

Enunț

Numim **corect** un număr care se împarte exact la fiecare dintre cifrele sale ***nenule***. De exemplu, 102 este **corect**, deoarece se împarte exact atât la 1, cât și la 2. Pe de altă parte, 282 nu este **corect**, pentru că nu se împarte exact la 8.

Cerință

Se dă un număr natural `N` ($\texttt{N} \leq 10^{18}$). Se cere să se găsească **cel mai mic** număr natural `X` astfel încât $\texttt{N} \leq \texttt{X}$ și `X` este un număr **corect**.

Soluție

1. Verificarea unui număr

Pentru început, ne propunem să verificăm dacă un număr fixat `X` este sau nu **corect**.

Pentru acest lucru, vom parcurge pe rând cifrele lui `X`, începând cu cea mai din dreapta. Vom verifica, pentru fiecare cifră nenulă, dacă aceasta îl divide sau nu pe `X`. În cazul în care întâlnim o cifră la care `X` nu se împarte exact, concluzionăm că `X` nu este **corect** și oprim algoritmul. Pe de altă parte, dacă reușim să parcurgem toate cifrele lui `X` fără să ne oprim pe parcurs, rezultă că `X` este un număr **corect**.

De exemplu, pentru $\texttt{X} = 28704$ algoritmul ar arăta astfel: Parcurgem cifrele lui 28704 în ordine începând cu cea mai din dreapta:

- 28704 se împarte exact la 4, deci putem trece la următoarea cifră;
- 0 nu este o cifră nenulă, deci putem trece la următoarea cifră, fără să verificăm alte condiții;
- 28704 nu se împarte exact la 7, deci oprim algoritmul cu rezultatul că 28704 **nu** este un număr **corect**.

Să considerăm un alt exemplu, de data aceasta pentru $\texttt{X} = 2170$:

- 0 nu este o cifră nenulă, deci putem trece la următoarea cifră, fără să verificăm alte condiții;
- 2170 se împarte exact la 7, deci putem trece la următoarea cifră;
- 2170 se împarte exact la 1, deci putem trece la următoarea cifră;
- 2170 se împarte exact la 2, deci putem trece la următoarea cifră;
- am parcurs toate cifrele lui 2170 fără să ne oprim, de unde tragem concluzia că 2170 este un număr **corect**.

Pentru a implementa acest algoritm în C/C++, setăm la început o variabilă de tip flag `is_fair` pe valoarea **true**. Parcurgem, apoi, toate cifrele lui `X` și, în caz că `X` nu se împarte exact la una dintre ele, setăm `is_fair` pe **false**. La final, valoarea din variabila `is_fair` ne va indica dacă `X` este sau nu un număr **corect**.

```
bool is_fair = true;
long long x_copy = x; //avem nevoie de o copie a lui x pentru
                      //a-i parcurge cifrele
while (x_copy > 0 && is_fair) {
    int current_digit = x_copy % 10;
    if (current_digit > 0 && x % current_digit != 0) {
        is_fair = false;
    }
    x_copy = x_copy / 10;
}
```

2. Soluția Bruteforce

Soluția Bruteforce la care ne putem gândi acum este una simplă. Pornim cu `X` de la `N` și îl incrementăm cu câte 1, până când găsim primul număr **corect**, care va fi soluția problemei noastre (codul pentru această implementare îl găsiți mai jos, la secțiunea "Cod Sursă").

Numărul de pași efectuați în cadrul acestei soluții este `nr_incrementări * nr_cifre_x`.

3. Soluția eficientă

Vom demonstra că soluția Bruteforce efectuează un număr mic de pași și este, în realitate, una eficientă.

Observăm că un număr care se împarte la toate cifrele 1, 2, ..., 9 este întotdeauna **corect**. Un astfel de număr este un multiplu al **LCM**(1, 2,...,9), unde prin **LCM**[1] am notat *cel mai mic multiplu comun*. Dar, **LCM**(1, 2, ..., 9) = 2520. Rezultă că orice multiplu de 2520 este un număr **corect**. Astfel, algoritmul Bruteforce va face maxim 2520 de incrementări până când va ajunge la un multiplu de 2520, deci maxim 2520 de incrementări până va găsi primul număr **corect**.

Ținând cont de faptul că numerele pe care le parcurgem au maxim 19 cifre, numărul maxim de pași al algoritmului este 2520·19, care este suficient de mic.

[1] **LCM** este acronimul de la Least Common Multiple

Cod Sursă

```
int main() {
    long long n, x;
    cin >> n;
    x = n;
    bool found = false;
    while (!found) {
        bool is_fair = true;
        long long x_copy = x; //avem nevoie de o copie a lui x pentru
                              //a-i parcurge cifrele
        while (x_copy > 0 && is_fair) {
            int current_digit = x_copy % 10;
            if (current_digit > 0 && x % current_digit != 0) {
                is_fair = false;
            }
            x_copy = x_copy / 10;
        }
        if (is_fair) {
            found = true;
        } else {
            x = x + 1;
        }
    }
    cout << x << "\n";
    return 0;
}
```

Surse se pot trimite la `https://codeforces.com/problemset/problem/1411/B`

Problema 2. Suma

Enunț

Se dă o matrice de mărime $N(\leq 1000)$ pe $M(\leq 1000)$ care conține numere întregi. Se poate efectua următoarea operație: se aleg două celule adiacente (care au o latură comună) și se înmulțesc valorile lor cu -1.

Cerință

Fie X suma valorilor din matrice. Care este valoarea maximă pe care o poate lua X dacă putem efectua oricâte (inclusiv 0) operații de tipul celei descrise mai sus?

Soluție

Adesea, rezolvarea unei probleme complicate poate fi începută prin rezolvarea unui caz particular, mai simplu, care apoi să fie generalizat. Deseori, în cazul problemelor ce au o matrice ca date de intrare, un caz care se dovedește util este cel în care matricea are o singură linie. Astfel, vom încerca să începem cu rezolvarea cazului $N = 1$ (matricea devine un vector).

1. Rezolvarea problemei pe vector

Exemple

Putem începe prin a analiza manual câteva cazuri pe care ulterior să le generalizăm.

$$e_1 = [1, 1, 1]$$

Pentru exemplul e_1, observăm că toate numerele sunt pozitive, deci nu putem crește suma prin a înmulți un număr cu -1.

$$e_2 = [1, -1, -1]$$

Pentru exemplul e_2, putem înmulți valorile de pe pozițiile 2 și 3 cu -1 folosind o operație, și vom obține un vector identic cu toate numerele pozitive, a cărui sumă nu mai poate crește. Obținem astfel suma 3.

$$e_3 = [-2, 1, 1, -2]$$

Pentru exemplul e_3, nu există o singură operație care să facă toate numerele pozitive. Încercăm să găsim o metodă pentru a schimba doar valorile de -2. Observăm că o astfel de strategie poate fi alegerea perechilor de pe pozițiile $(1, 2), (2, 3), (3, 4)$:

Constatăm că pozițiile 2 și 3 sunt înmulțite cu -1 de câte două ori, în final valorile lor fiind identice cu cele inițiale. Astfel, obținem suma $2 + 1 + 1 + 2 = 6$.

$$e_4 = [-1, -1, -2]$$

Să încercăm o serie de operații pe acest vector. De exemplu:

- efectuăm o operație pe pozițiile $(1, 2)$. Vectorul devine [1, 1, -2]
- efectuăm o operație pe pozițiile $(2, 3)$. Vectorul devine [1, -1, 2]
- efectuăm o operație pe pozițiile $(1, 2)$. Vectorul devine [-1, 1, 2]
- efectuăm o operație pe pozițiile $(1, 2)$. Vectorul devine [1, -1, 2]

Observăm că orice operații am efectua, cel puțin un număr rămâne negativ. Putem explica acest fenomen observând faptul că aplicarea unei operații nu schimbă produsul numerelor, acesta rămânând constant. Anume, dacă el e negativ la început, vă rămâne negativ pe parcursul tuturor operațiilor, deci în vector vom avea cel puțin o valoare negativă. Așadar, este optim să rămânem cu vectorul final [-1, 1, 2], pentru a obține suma 2.

Generalizarea operației

În exemplul e_3, observăm că putem înmulți două numere neadiacente cu -1, fără să modificăm alte valori din vector. Încercăm să demonstrăm că acest lucru este posibil pentru orice pereche de numere dintr-un vector oarecare.

O observație care poate simplifica demonstrația este că două operații efectuate asupra aceleiași celule nu schimbă semnul valorii din celula respectivă. De exemplu, dacă avem un vector cu minim 3 valori, efectuând operațiile $(1, 2)$ și $(2, 3)$, celulele 1 și 3 vor fi schimbate, dar celula 2 va rămâne la fel.

Astfel, pentru a modifica exact două celule i și j, $i < j$, și a le lăsa pe celelalte neschimbate, trebuie ca acestea să apară într-un număr impar de operații, în timp ce toate celelalte celule să apară într-un număr par de operații. Vom încerca să găsim o construcție care să îndeplinească acest criteriu.

O astfel de construcție se dovedește a fi una simplă: dacă luăm perechile $(i, i + 1), (i + 1, i + 2), ...(j - 1, j)$, atunci toate pozițiile, diferite de i și j, vor apărea de două ori, în timp ce i și j vor apărea exact o dată în operații (orice k, $i < k < j$, va apărea doar în perechile $(k - 1, k)$ și $(k, k + 1)$).

Așadar, am obținut o metodă de a crea o super-operație (înmulțirea cu -1 a oricăror două celule), care este mai generală decât operația inițială și poate fi simulată prin operații simple.

Deseori, în rezolvarea problemelor cu operații, este folositor să aducem operațiile într-o formă mai ușor de analizat, dar cu un efect echivalent (cum este în acest caz transformarea unui set de operații într-o super-operație).

Soluția finală pe vector

Acum, putem încerca să rezolvăm problema folosind doar super-operații. Observați că orice operație poate fi privită ca o super-operație, prin urmare nu pierdem soluții dacă folosim doar super-operații.

Fie I numărul de numere negative din vector. Ideal, am vrea ca I să devină egal cu 0 (dacă nu avem numere negative, nu putem îmbunătăți suma, prin urmare aceasta este maximă).

Observăm că 0 este un număr particular - indiferent câte operații îi sunt aplicate, valoarea sa nu se schimbă (în general, astfel de cazuri particulare trebuie tratate cu grijă). Astfel, analizând, distingem următoarele cazuri:

1) Dacă avem o valoare egală cu 0 în vector, putem să aplicăm succesiv super-operația pe toate perechile formate dintr-o celulă cu valoare negativă și celula de valoare 0. Astfel, toate numerele negative își schimbă semnul, prin urmare I devine 0.

2) Dacă nu avem valori egale cu 0, observăm că, la o super-operație, I rămâne neschimbat, crește cu 2 sau scade cu 2, prin urmare nu își schimbă paritatea:

Demonstrație: Fie a un vector, în care vrem să aplicăm super-operația pe celulele i și j. Avem următoarele cazuri:

- $a(i) < 0$, $a(j) < 0$: I devine $I - 2$
- $a(i) < 0$, $a(j) > 0$: I rămâne neschimbat
- $a(i) > 0$, $a(j) < 0$: I rămâne neschimbat
- $a(i) > 0$, $a(j) > 0$: I devine $I + 2$.

Astfel, dacă nu avem valori de 0 în vector, putem obține $I = 0$ doar dacă I-ul inițial este par. Cum? Putem împărți numerele negative în perechi disjuncte pe care efectuăm super-operația. Practic, în cazul I par, știm că putem ajunge la $I = 0$ prin super-operații, deci răspunsul final este suma modulelor numerelor din matrice.

Dacă I este impar, nu putem scăpa de toate numerele negative din matrice, indiferent de super-operațiile pe care le-am efectua (0 este număr par, iar I va fi mereu impar). În acest caz, o idee ar fi să găsim soluția optimă (care inevitabil conține un număr impar de numere negative) și să demonstrăm că o putem construi oricând folosind super-operații. Observăm că, în cel mai bun caz, vectorul nostru final ar trebui să conțină un singur număr negativ, iar acel număr să fie cel cu valoarea absolută cea mai mică. Să demonstrăm că putem ajunge mereu la o astfel de configurație a vectorului. O posibilă secvență de super-operații este următoarea:

- Grupăm numerele negative din vectorul inițial două câte două și aplicăm super-operația pe aceste perechi. Deoarece I este inițial impar, vom rămâne cu exact un număr negativ după acest pas;

- Dacă numărul negativ rămas are valoarea absolută minimă din vector, am obținut configurația dorită (de exemplu rămânem cu vectorul $[2, 3, -1]$, iar rezultatul nu mai poate fi îmbunătățit). Dacă nu, efectuăm super-operația pe numărul negativ rămas și pe numărul cu valoarea absolută minimă din vector și ajungem, astfel, la rezultatul dorit. Un exemplu pentru a doua situație este următorul: avem vectorul $[1, -3, -3, -2]$ și îl transformăm în $[1, 3, 3, -2]$ cu operația $(2, 3)$. Avem deja un singur număr negativ, dar observăm că este optim să ducem vectorul la starea $[-1, 3, 3, 2]$ prin super-operația $(1, 4)$.

În concluzie, soluția pe vector este următoarea:

- Dacă vectorul conține cel puțin un 0, rezultatul este suma valorilor absolute ale elementelor din vector;
- Dacă vectorul nu conține niciun 0 și avem un număr par de numere negative, rezultatul este suma valorilor absolute ale elementelor din vector;
- Dacă vectorul nu conține niciun 0 și avem un număr impar de numere negative, rezultatul se obține prin însumarea valorilor absolute ale elementelor din vector, cu excepția valorii absolute minime, pe care trebuie să o scădem.

2. Generalizare pe matrice

Așa cum se întâmplă de multe ori, observăm că soluția pe vector poate fi extinsă pe matrice. Astfel, putem folosi și în acest caz ideea de super-operație.

Pentru un vector, o super-operație între celulele i și j, $i < j$, este construită aplicând operația din enunț pe câte două poziții adiacente pe intervalul de la i la j (adică pe pozițiile $(i, i+1), (i+1, i+2)...(j-1, j)$). Asemănător, pe o matrice putem construi super-operația pentru două celule (i_1, j_1), (i_2, j_2) aplicând operația din enunț pe câte două poziții adiacente care fac parte dintr-un *drum* între cele două celule. Explicația este aceeași ca în cazul anterior: cu excepția celulelor (i_1, j_1), (i_2, j_2), toate celulele de pe *drum* sunt înmulțite cu -1 de două ori, și deci rămân neschimbate.

-1	4	-2	6	5	3
15	-14	12	-23	-10	-2
-16	7	18	-9	-12	5
4	-2	9	11	-3	1
13	-16	-6	9	19	20

În exemplul de mai sus, am construit o super-operație pentru celulele $(1, 4)$ și $(5, 6)$. Super-operația va fi formată din următoarele operații simple: [(1,4), (1,5)], [(1,5), (2,5)], [(2,5), (3,5)], [(3,5),(4,5)], [(4,5),(4,6)], [(4,6),(5,6)] (grupuri de câte două celule adiacente de

pe un *drum* de la $(1, 4)$ la $(5, 6)$). Astfel, observăm că numerele 6 și 20 pe care dorim să efectuăm super-operația vor fi înmulțite cu -1 o singură dată, deci își vor schimba semnul. Pe de altă parte, restul numerelor de pe *drum*, anume (5, -10, -12, -3, 1), vor fi înmulțite de două ori cu -1, prin urmare vor rămâne neschimbate.

Așadar, orice super-operație poate fi construită dintr-o secvență de operații inițiale și pe o matrice. Din acest punct, putem rezolva problema la fel cum am rezolvat-o și pe vector, deoarece, în continuare, soluția tratează numerele ca făcând parte dintr-o mulțime neordonată și nu ține cont de modul în care acestea sunt aranjate (în formă de vector sau matrice).

Complexitatea finală a algoritmului este $O(N \cdot M)$

3. Implementare

Pentru a implementa acest algoritm în C/C++, vom împărți codul în două etape:

- O parcurgere a matricei prin care vom calcula toate valorile de care avem nevoie, și anume:
  - o variabilă `exists_0`, de tip **boolean**, care să ne spună dacă există sau nu o valoare egală cu 0 în matrice. Pentru a o calcula, o setăm inițial pe **false**. Dacă pe măsură ce parcurgem matricea găsim o valoare de 0, setăm variabila pe **true**;
  - o variabilă `negative_count`, care să rețină câte numere negative sunt în matrice;
  - o variabilă `sum_absolute_value`, care să rețină suma valorilor absolute din matrice;
  - o variabilă `min_absolute_value`, care să rețină minimul valorilor absolute din matrice.

  Această etapă poate fi realizată încă de la citire.
- Încadrarea într-unul din cele trei cazuri prezentate mai sus și găsirea rezultatului:
  - Dacă `exists_0` este **true**, afișăm `sum_absolute_value`;
  - Dacă `exists_0` este **false** și `negative_count % 2 == 0`, afișăm `sum_absolute_value`;
  - Dacă `exists_0` este **false** și `negative_count% 2 == 1`,
    afișăm `sum_absolute_value - 2 * min_absolute_value`.

Cod Sursă

```
int main() {
    bool exists_0 = false;
    long long sum_absolute_value = 0;
    int min_absolute_value = 1000000001; //o valoare care este
                                         //suficient de mare
    int negative_count = 0;
    int N, M;
    cin >> N >> M;
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= M; j++) {
            int elem;
            cin >> elem;
            int absolute_value; //valoarea absoluta a lui elem
            if (elem == 0) {
                absolute_value = 0;
                exists_0 = true;
            } else if (elem < 0) {
                absolute_value = -elem;
                negative_count += 1;
            } else {
                absolute_value = elem;
            }
            sum_absolute_value += absolute_value;
            if(min_absolute_value > absolute_value)
                min_absolute_value = absolute_value;
        }
    }

    if (exists_0) {
        cout << sum_absolute_value << "\n";
    }
    if (!exists_0 && negative_count % 2 == 0) {
        cout << sum_absolute_value << "\n";
    }
    if (!exists_0 && negative_count % 2 == 1) {
        cout << sum_absolute_value - 2 * min_absolute_value<< "\n";
    }
    return 0;
}
```

Surse se pot trimite la `https://codeforces.com/problemset/problem/1447/B`

Problema 3. Bossime

Enunț

Se dau două numere A și B. Să se determine un x natural astfel încât cifrele numărului $A + x$ să fie o permutare a cifrelor lui $B + x$.

Restricții

- $1 \leq A, B \leq 10^9$
- Răspunsul afișat se cere a fi $\leq 10^{17}$ și se garantează că, în caz că perechea admite soluție, mereu va exista cel puțin o soluție sub această limită;
- Se admite orice răspuns corect ce se încadrează în limitele specificate;
- NU se admit "leading zeroes" în scrierea soluțiilor $A + x$ și $B + x$, de exemplu 105, 015.

Exemple

- Pentru perechea $\{13; 58\}$, un răspuns este $x = 3 \rightarrow \{16; 61\}$;
- Pentru perechea $\{13;\ 24\}$, nu există răspuns;
- Pentru perechea $\{153;\ 270\}$, un răspuns este $x = 50 \rightarrow \{203; 320\}$.

Abordare de rezolvare

În continuare vom prezenta o modalitate de abordare a acestei probleme. Vom trece prin întregul proces de gândire, integrând în raționament inclusiv unele idei intermediare, la care însă vom renunța mai târziu. În final, soluția va rezulta din observațiile făcute pe parcursul descrierii ce urmează.

Punct de plecare

Întotdeauna, primul și cel mai important pas este să citim cu atenție enunțul și abia apoi să ne apucăm de rezolvarea cerinței. În citirea enunțului trebuie luate în seamă orice fel de restricții ce par "nenaturale". În speța noastră nu avem nicio restricție așa-zis "nenaturală" de care să ne putem agăța ca punct de plecare, așa că vom continua cu sublinierea unei interpretări la prima vedere a acestor restricții:

- Un răspuns $\leq 10^{17}$ sugerează indirect că soluția trebuie găsită folosind proprietăți ale numerelor și nu simulare/verificare exhaustivă. Totuși, există și șansa ca această valoare să nu aibă o relevanță esențială și să existe mereu și soluții cu x mic.
- Nu se admit "leading zeroes" pare acel tip periculos de restricție la care trebuie neapărat să ne întoarcem, după ce avem conturată o soluție completă și ne pregătim de implementare.

Pașii următori

Ce proprietăți matematice (ce pot fi implementate în cod) au două numere ce reprezintă două permutări ale acelorași cifre?

- Niciuna nu pare evidentă, așa că încercăm să introducem niște notații $A + x = A^*$ și $B + x = B^*$, în speranța că vom reuși să ajungem la vreo observație. Aceste două ecuații conțin trei necunoscute: x, A^*, B^*, unde între A^* și B^* există proprietatea permutării cifrelor. Pare oarecum delicat să determinăm un triplet valid, atâta timp cât încă nu știm să formalizăm algoritmic dependența dintre A^* și B^*. Fiind blocați în acest punct, încercăm să schimbăm abordarea de tip determinist (utilizarea parametrilor pentru a restrânge domeniul posibil al soluțiilor la câteva valori printre care putem căuta soluția) într-o alta, de tip constructiv (pornind de la datele problemei, încercăm să construim pas cu pas o soluție).

- Să încercăm să ne axăm pe o operație "mai simplă", inclusă în operația complexă din enunț, pentru a câștiga mai mult control asupra reprezentării procesului constructiv. Prin operație simplă ne referim la tipuri specifice de permutări (e.g.[1] inversiuni intre poziții date, permutări circulare etc). Motivația din spate constă în "speranța" ca operația complexă din enunț să conțină un grad foarte mare de inutilitate, caz în care am putea încerca o simplificare a ei și verifica dacă această simplificare este suficientă pentru a obține soluții. În situația ideală, simplificarea returnează soluție întotdeauna când aceasta există.

- Să pornim de la presupunerea că A^* e identic cu B^*, în afară de două cifre care sunt inversate. Dar care două cifre ar fi cele mai potrivite pentru a găsi o soluție în cât mai multe situații? Începem prin a verifica (pe hârtie) cum se comportă o inversiune între prima și ultima cifră. Luăm un exemplu de dimensiune rezonabilă și simulăm această construcție. Dacă vedem că pe un exemplu aleatoriu găsim soluție, atunci o să începem să formalizăm reprezentări pentru cazurile generale.

  Reamintim că $A + x = A^*$ și $B + x = B^*$. Fie exemplul $A^* = \overline{abcde}$, $B^* = \overline{ebcda}$ (inversiune între prima și ultima cifră). $A + x = \overline{abcde}$; $B + x = \overline{ebcda}$.

$$x = -A + \overline{abcde} = -A + 10000 \cdot \overline{a} + \overline{e} + \overline{bcd0}$$
$$x = -B + \overline{ebcda} = -B + 10000 \cdot \overline{e} + \overline{a} + \overline{bcd0}$$

  Deci:

$$-A + 10000 \cdot \overline{a} + \overline{e} + \overline{bcd0} = -B + 10000 \cdot \overline{e} + \overline{a} + \overline{bcd0}$$

  Eliminăm $\overline{bcd0}$:

$$-A + 10000 \cdot \overline{a} + \overline{e} = -B + 10000 \cdot \overline{e} + \overline{a}$$

$$9999(\overline{a} - \overline{e}) = A - B$$

[1] exempli gratia

$A-B$ este dat în enunț, iar $(a-e) \in \{-9, -8, ..+9\}$. Practic, putem varia valorile celor două variabile a și e și să găsim numai 19 valori posibile pe care le poate lua partea stângă a ecuației. Dar fiind doar 19 posibilități, sunt slabe șansele ca printre acestea să se afle exact valoarea lui $A - B$ (partea dreaptă a ecuației). Am ales precedent să inversăm prima și ultima cifră, dar, intuitiv, și dacă am alege alte două poziții, procedura ar fi oarecum similară și am rămâne tot cu două variabile/cifre în partea stângă a ecuației. Astfel, cel mai probabil nu am depăși 100 de valori posibile pentru $A - B$ (numărul 100 venind de la numărul de posibilități de a fixa simultan cele două cifre din partea stângă a ecuației). Dacă ne gândim să variem oricare două poziții de inversat și apoi să variem și valorile celor două cifre implicate, probabil obținem tot ceva mic, în jur de $17^2 * 100$ valori posibile pentru $A-B$ (17 reprezintă numărul maxim de cifre pe care le poate avea soluția $x \leq 10^{17}$). Dar deja pare că ne complicăm, așa că ar fi bine să căutăm o altă construcție.

Rămânând pe ideea precedentă, încercăm să găsim permutări specifice pe care să le folosim ca și operații "mai simple". Să încercăm acum o rotație a cifrelor: $A^* = \overline{abcde}$, $B^* = \overline{eabcd}$. Pentru a simplifica ecuațiile de mai jos, să analizăm cazul mic în care A^* șiB^* au doar 5 cifre:

- $A + x = A^*$ și $B + x = B^*$ implică $A + x = \overline{abcde}$; $B + x = \overline{eabcd}$;

$$A + x = \overline{abcde} = \overline{e} + 10 * \overline{abcd}$$
$$B + x = \overline{eabcd} = 10^4 * \overline{e} + \overline{abcd}$$

$$-A + \overline{e} + 10 * \overline{abcd} = -B + 10^4 * \overline{e} + \overline{abcd}$$

$$9 * \overline{abcd} = A - B + 9999 * \overline{e}$$

$$\overline{abcd} = \frac{A - B}{9} + 1111 * \overline{e}$$

Din ultima ecuație observăm că putem găsi soluție pentru cazurile când $A-B$ se divide cu 9 ($\overline{abcd}$ și $\overline{e}$ sunt doar părți din soluția $x = \overline{abcde} - A$). Fără să restrângem generalitatea, vom presupune că $A > B$, anume că diferența lor este pozitivă. În comparație cu metoda precedentă, numărul de perechi (A, B) pentru care putem returna soluție s-a mărit considerabil. Merită acum să fie analizată posibilitatea ca nu cumva această soluție să se încadreze în categoria celor cu "leading zeroes", care sunt nu sunt valide. Reamintim că am simplificat calculele prin analiza cazului când A^* șiB^* au numai 5 cifre. Aplicând aceeași procedură, pe forma generală de n cifre am obține:

$$\overline{w_1 w_2 w_3 .. w_{n-1}} = \frac{A - B}{9} + 11..1 * \overline{w_n}$$

cu $n-1$ cifre de 1 în numărul 11..1.

Dacă $A-B$ are strict mai puține cifre decât lungimea lui $\overline{w_1w_2w_3..w_{n-1}}$, atunci pentru un $\overline{w_n}$ mic, e.g. $\overline{w_n}=1$, suma din dreapta va păstra lungimea de $n-1$ cifre. Fiind permisă returnarea soluțiilor de până la 17 cifre și având diferența maximă $A-B$ de ordinul 10^9, putem obține soluție pentru orice $A-B$ divizibil cu 9. O bună practică în momentul obținerii unei rezolvări la o problemă este verificarea de mână a exemplelor din enunț. Vedem că ambele teste care admit soluție au diferența divizibilă cu 9 și începem să prindem curaj în a încerca să demonstrăm că divizibilitatea numărului $A-B$ cu 9 este criteriul de decizie pentru admitere de soluție sau nu.

Urmează să demonstrăm că dacă diferența $A-B$ nu este multiplu de 9, atunci cu siguranță nu există nicio soluție validă:

- Ipoteza este extrem de naturală și putem reveni la ideea inițială de analiză a proprietăților matematice ce există între două numere formate din aceleași cifre, doar că permutate. Luăm două numere formate din aceleași cifre, scădem un x mic din ambele ($x=3$ pare suficient de bun) și testăm dacă într-adevăr diferența dintre numere rămâne multiplu de 9. Considerând 402 și 240, scădem 3 și obținem 399 și 237. Diferența este într-adevăr 162, multiplu de 9. *În timp ce efectuăm acești pași, realizăm că sunt inutili: scădem, de fapt, același x din două numere de la care ne interesează numai diferența.* Așa că, pentru demonstrația propusă, este suficient să arătăm că diferența dintre oricare două numere formate din aceleași cifre este multiplu de 9. Pare că trebuie să descompunem diferența pe cifre:

  - fie a, $\overline{b}$, $\overline{c}$, $\overline{d}$ cifrele din care sunt compuse A^* și B^*;
  - fie cifra a pe pozițiile p_1 și p_2 în A^*, respectiv B^*.

  Atunci, fiecare din aceste patru cifre va avea un anumit coeficient în descompunerea diferenței A^*-B^*:

$$A^*-B^*=k_aa+k_b\overline{b}+k_c\overline{c}+k_d\overline{d}$$

  Cifra a va avea coeficientul $k_a=10^{p_1}-10^{p_2}$ care are forma $k_a=\pm 9..90..0$, deci multiplu de 9. Identic, toți coeficienții vor fi multipli de 9, așadar întregul A^*-B^* va fi sumă de multipli de 9. Observând că diferența rămâne constantă la adunarea/scăderea aceluiași x din A și din B, am demonstrat că dacă $A-B$ nu este multiplu de 9, atunci nu există nicio pereche $A^*=A+x$, $B^*=B+x$ cu același set de cifre.

Implementare

Conform pașilor de mai sus, tot ce avem de făcut este să considerăm un nou exemplu $A+x=A^*$ și $B+x=B^*$ cu $A^*=\overline{w_1w_2w_3..w_n}$ și să aplicăm formula $\overline{w_1w_2w_3..w_{n-1}}=\frac{A-B}{9}+11..1*\overline{w_n}$. Trebuie să fixăm doar valoarea ultimei cifre $\overline{w_n}$ și numărul de cifre n. Conform explicațiilor anterioare:

- $\overline{w_1 w_2 w_3..w_n}$ poate avea orice număr de cifre atâta timp cât sunt mai multe decât numărul cifrelor lui $A - B$;
- ultima cifră a lui A^* poate fi setată liniștit la $\overline{w_n} = 1$.

Pentru reconstrucție rescriem $A^* = 10\overline{w_1 w_2 w_3..w_{n-1}} + \overline{w_n}$ și $x = A^* - A$. Pentru a acoperi toate cazurile de numere ce pot apărea în input, considerăm $n = 11$, deoarece diferența maximă din input are 9 cifre.

În continuare avem:

$$\overline{w_1 w_2 w_3..w_{10}} = \frac{A - B}{9} + 1.111.111.111 \cdot \overline{w_{11}}$$

Dar $\overline{w_{11}} = 1$:

$$\overline{w_1 w_2 w_3..w_{10}} = \frac{A - B}{9} + 1.111.111.111$$

$$A^* = \overline{w_1 w_2 w_3..w_{11}} = 10\left(\frac{A - B}{9} + 1.111.111.111\right) + \overline{w_{11}}$$

$$A^* = \overline{w_1 w_2 w_3..w_{11}} = 10\left(\frac{A - B}{9} + 1.111.111.111\right) + 1$$

$$x = A^* - A = 10\left(\frac{A - B}{9} + 1.111.111.111\right) + 1 - A$$

- Amintim că am presupus $A > B$ și că diferența $A - B$ e pozitivă. Având în vedere soluția finală, vom afirma că x rămâne pozitiv, chiar dacă $A < B$.

  Demonstrație: observăm că $\frac{|A-B|}{9} \leq 111.111.111$.
  Atunci $-\frac{|A-B|}{9} + 1.111.111.111 \geq 10^9 \geq A$, deci $x \geq 0$.

Cod Sursă

```
int main() {
    int num_pairs;
    cin >> num_pairs;
    const int ADD = 1111111111;
    for (int i = 0; i < num_pairs; i++) {
        int a, b;
        cin >> a >> b;
        if ((a - b) % 9) {
            cout << "Imposibil\n";
            continue;
        }
        cout << 1LL * ((a - b) / 9 + ADD) * 10 + 1 - a << "\n";
    }
    return 0;
}
```

Surse se pot trimite la `https://infoarena.ro/problema/bossime`

Problema 4. Caraibe

Enunț

Cei N pirați de pe Perla Neagră au făcut recent o captură foarte importantă: un cufăr cu 10.000.000.000 (zece miliarde) de bănuți. Acum piraților au de rezolvat o problemă și mai dificilă: cum să împartă banii.

Pentru împărțire, piraților se așază în linie. Primul pirat va propune o schemă de împărțire a banilor. Dacă un anumit număr de pirați nu sunt de acord cu această schemă, piratul va fi aruncat peste bord, apoi următorul pirat va propune o schemă de împărțire, și tot așa. Piraților sunt foarte inteligenți: un pirat este de acord cu o schemă de împărțire doar dacă aceasta îi aduce un avantaj strict (cel puțin un bănuț) față de ce ar obține votând împotriva schemei. Pentru că acționează numai pe baze rationale, piraților sunt și foarte previzibili. Cu alte cuvinte, un pirat poate anticipa decizia altor pirați pentru a lua o decizie proprie (aceasta înseamnă și că dacă un pirat are mai multe posibilități de a alege o schemă de împărțire, ceilalți pirați știu ce variantă ar alege).

Depinzând de caracteristicile fiecărui pirat (forță, popularitate), numărul de pirați care trebuie să fie de acord cu schema lui pentru a nu fi aruncat peste bord variază. Să zicem că pentru piratul i ($1 \leq i < N$) acest număr este A_i. Dacă piratul i propune o schemă, știm că toți piraților până la $i-1$ au fost aruncați deja peste bord. În afară de piratul i, mai există $N-i$ pirați. Dacă cel puțin A_i dintre aceștia sunt de acord cu schema piratului i, comoara va fi împărțită după această schemă. Altfel, piratul i va fi aruncat peste bord, și piratul i+1 va propune o schemă. Pentru orice i, avem $0 \leq A_i < N-i$. Datorită acestei condiții $A_{N-1} = 0$, iar A_N nu este definit (pentru că piratul N este ultimul).

Cerință

Primul pirat din linie dorește să propună o schemă de împărțire a banilor astfel încât să nu fie aruncat peste bord, iar el să primească cât mai mulți bănuți. Determinați suma maximă pe care o poate primi. Se garantează că există o schemă pe care o poate propune primul pirat, astfel încât să nu fie aruncat peste bord.

Restricții

- $2 \leq N \leq 65.000$;
- $0 \leq A_i < N-i$.

Explicarea neclarităților din cerință

Pentru un elev nefamiliarizat cu teoria jocurilor, unele sintagme precum *acționează numai pe baze rationale* ori *piratii sunt și foarte previzibili* pot fi confuze. În enunț sunt descrise regulile unui joc destul de complex, după care este introdus paragraful (util de recitit chiar acum):

Pentru că acționează numai pe baze raționale, piratii sunt și foarte previzibili. Cu alte cuvinte, un pirat poate anticipa decizia altor pirați pentru a lua o decizie proprie (aceasta înseamnă și că dacă un pirat are mai multe posibilități de a alege o schemă de împărțire, ceilalți pirați știu ce variantă ar alege).

Paragraful subliniază faptul că situația în care toți piratii joacă optim formează un punct de echilibru al jocului în sine. Prin punct de echilibru ne referim mai exact la faptul că există o înșiruire de acțiuni (mutări pentru fiecare jucător) ce maximizează profitul tuturor piraților simultan. Ca o paralelă pentru conceptul de "punct de echilibru" ne putem gândi la jocul de X și 0, unde, dacă ambii jucători joacă optim, există o înșiruire de mutări în urma căreia niciunul nu va pierde. Numai că, la X și 0, outputul este ternar - *win/lose/draw* - în timp ce la pirați outputul este mult mai complicat. Revenind la paragraful de mai sus, se menționează și că jocul admite un unic punct de echilibru, deoarece fiecare pirat cunoaște strategia optimă a oricărui alt pirat.

Pe scurt, deși se dau regulile unui joc complex, pe noi ne interesează doar acea unică înșiruire de acțiuni ce maximizează simultan profitul tuturor piraților. Cerința este să înțelegem în profunzime regulile jocului (inclusiv să ne convingem că există această unică strategie perfectă) și să simulăm, parțial sau complet, aceste acțiuni, până obținem profitul primului pirat.

Restricțiile problemei

Să începem procesul de rezolvare cu puțină analiză pe text:

- Suma totală de 10.000.000.000 este prea mare ca să iterăm pe ea. Cu siguranță soluția nu va avea complexitate liniară în numărul de bănuți. De fapt, probabil complexitatea soluției nu depinde deloc de numărul de bănuți.
- $N \leq 65.000$ sigur cere o complexitate mult mai bună de $O(n^2)$.
- Pentru toate testele: A_n nu este definit și $A_{n-1} = 0$. Această restricție pare destul de nenaturală, așa că este foarte potrivită ca punct de plecare în analiza schemei de joc.

Pași de rezolvare

a) Enunțul garantează că piratul 1 va avea o strategie în care oferta lui este acceptată. Astfel, intuim că orice pirat, în cazul în care ar ajunge să propună, ar avea o strategie prin care oferta sa să fie acceptată. Această presupunere ne-ar ajuta să dezvoltăm o intuiție mai largă asupra problemei, așa că putem porni de la ea și încerca să o confirmăm pe parcursul

raționamentului. Dacă vom observa la un moment dat că presupunerea nu stă în picioare, atunci, din păcate, va trebui să reluăm gândirea problemei de la zero.

b) Revenim asupra analizei restricției $A_{n-1} = 0$. Dacă, prin absurd, am avea $A_{n-1} = 1$, atunci piratul $n-1$ ar avea nevoie de votul ultimului pirat. Dar ultimul pirat ar fi perfect conștient că, dacă refuză, atunci ajunge să păstreze toți banii. Așa că piratul n nu ar avea niciun interes să discute cu piratul $n-1$.

c) Pasul următor ar fi să vedem dacă putem generaliza această observație în ceva de tipul *piratul i nu va obține niciodată votul piratului i+1*. Intuiția ne spune că, deoarece există foarte mulți bănuți în total, tactica oricărui pirat va fi să cumpere strictul necesar de voturi și apoi să păstreze toți bănuții rămași. Astfel, de departe cel mai câștigat va fi mereu piratul care face propunerea. În aceste condiții, votul piratului $i+1$ va fi imposibil de obținut pentru o ofertă propusă de i.

d) Să analizăm acum pe ce se bazează piratul i când propune o ofertă. Cu siguranță va lua în calcul oferta pe care $i+1$ o va face în caz că acesta ajunge la putere. Dar reciproca nu este adevarata, oferta $i+1$ nu depinde în niciun fel de oferta i, căci, dacă $i+1$ ajunge să propună o împărțire a banilor, atunci piratul i a fost deja eliminat. Astfel, fiecare pirat e nevoit să ia în considerare strategiile acelora care urmează după el, dar nu trebuie să se gândească la strategiile celor deja eliminați.

e) Având acestea în vedere, un pirat i trebuie să se asigure că nu va fi refuzat și, pentru aceasta, va trebui să propună o ofertă "mai bună" decât oricare dintre ofertele optime propuse de $i+1$, $i+2$, .., $n-1$ (reamintim că este subliniat în cerință faptul că există o unică strategie optimă pentru fiecare pirat în parte, iar piratul i poate prezice ce vor propune toți ceilalți). Prin "mai bună" înțelegem că poate obține voturile a A_i pirați și astfel oferta să-i fie acceptată, iar următorii pirați să nu aibă șansa de a mai propune. Însă, deoarece rezolvarea noastră este de la coadă la cap, $i+1$ și-a făcut deja strategia de joc și s-a asigurat că propunerea lui este "mai bună" decât $i+2$, $i+3$ etc. Cu alte cuvinte, putem spune că am găsit o operație de comparare între oferte. Conform enunțului, această comparație este strictă: oferta i trebuie să fie strict mai avantajoasă decât orice alternativă pentru toți pirații care își dau votul lui i.

f) Tocmai am arătat că:

- oferta i depinde de ofertele $i+1$, $i+2$ etc.
- oferta $i+1$ depinde de ofertele $i+2$, $i+3$ etc.
- la momentul în care $i+1$ și-a construit strategia, acesta s-a asigurat deja că oferta îi va fi acceptată și atunci $i+2$, $i+3$ etc. nu mai sunt relevante.

Considerând aceste trei observații, reiese că este suficient ca oferta i să fie construită numai pe baza ofertei $i+1$. Astfel i va supralicita oferta $i+1$ până va obține numărul necesar de voturi.

g) Așadar, planul de rezolvare este ca întâi să construim oferta lui $n-1$, apoi $n-2$ și, în final, oferta lui 1 și să afișăm răspunsul cerut de problemă.

h) Să analizăm oferta făcută de $n-2$, unde au rămas numai 3 pirați. Avem observația de mai sus care ne spune că i nu poate obține votul lui $i+1$. Astfel, dacă $A_{n-2}=1$, singurul pirat cu care $n-2$ poate negocia este n și acestuia îi promite exact 1 bănuț. Un singur bănuț este suficient, deoarece $A_{n-1}=0$, adică piratul n nu va primi nimic dacă $n-1$ face împărțirea. Introducem notația $S_j^i =$ ***numărul de bănuți pe care îi primește piratul*** j ***în oferta făcută de*** i. Marcăm $S_i^i=\infty$ ca fiind suma totală din care scădem sumele pe care piratul i promite să le ofere celorlalți:

$A_{n-2}=1$	$A_{n-1}=0$	$A_n=-$
$S_{n-2}^{n-2}=\infty$	$S_{n-1}^{n-2}=0$	$S_n^{n-2}=1$

Iar în cazul în care $A_{n-2}=0$:

$A_{n-2}=0$	$A_{n-1}=0$	$A_n=-$
$S_{n-2}^{n-2}=\infty$	$S_{n-1}^{n-2}=0$	$S_n^{n-2}=0$

i) Construim acum oferta $n-3$, notată S^{n-3}. Pentru un singur vot piratul cu indicele $n-3$ va apela la piratul $n-1$ ce a fost ignorat în oferta S^{n-2}. Pentru două voturi i se vor acorda alți $S_n^{n-3}=2$ bănuți și piratului n, supralicitând $S_n^{n-2}=1$:

$A_{n-3}=2$	$A_{n-2}=1$	$A_{n-1}=0$	$A_n=-$
$S_{n-3}^{n-3}=\infty$	$S_{n-2}^{n-3}=0$	$S_{n-1}^{n-3}=1$	$S_n^{n-3}=2$

$A_{n-3}=1$	$A_{n-2}=1$	$A_{n-1}=0$	$A_n=-$
$S_{n-3}^{n-3}=\infty$	$S_{n-2}^{n-3}=0$	$S_{n-1}^{n-3}=1$	$S_n^{n-3}=0$

$A_{n-3}=0$	$A_{n-2}=1$	$A_{n-1}=0$	$A_n=-$
$S_{n-3}^{n-3}=\infty$	$S_{n-2}^{n-3}=0$	$S_{n-1}^{n-3}=0$	$S_n^{n-3}=0$

j) Soluția pare că s-a conturat din exemplele precedente. Să încercăm acum să compunem oferta S^{n-4} bazându-ne numai pe S^{n-3} cu $A_{n-3}=2$, $A_{n-2}=1$, $A_{n-1}=0$. Cel mai ieftin vot ar fi cel al piratului $n-2$: $S_{n-2}^{n-4}=1$ bănuț (deoarece $S_{n-2}^{n-3}=0$). În caz de două voturi, o să avem $S_{n-1}^{n-4}=2$ (deoarece $S_{n-1}^{n-3}=1$), iar în caz de trei voturi: $S_n^{n-4}=3$.

$A_{n-4}=1$	$A_{n-3}=2$	$A_{n-2}=1$	$A_{n-1}=0$	$A_n=-$
$S_{n-4}^{n-4}=\infty$	$S_{n-3}^{n-4}=0$	$S_{n-2}^{n-4}=1$	$S_{n-1}^{n-4}=0$	$S_n^{n-4}=0$

$A_{n-4}=2$	$A_{n-3}=2$	$A_{n-2}=1$	$A_{n-1}=0$	$A_n=-$
$S_{n-4}^{n-4}=\infty$	$S_{n-3}^{n-4}=0$	$S_{n-2}^{n-4}=1$	$S_{n-1}^{n-4}=2$	$S_n^{n-4}=0$

k) Observăm că update-ul de ofertă pentru piratul i reprezintă numai supralicitarea cu 1 bănuț ofertei precedente: $S_j^i = 1 + S_j^{i+1}$ pentru cei A_i pirați care au pretențiile financiare cele mai mici. Acești A_i pirați îi asigură lui i voturile de care are nevoie ca să îi fie acceptată propunerea, în rest, toți ceilalți primesc 0 bănuți în cadrul ofertei.

Gândirea implementării

a) Avem ideea soluției, urmează să găsim un mod eficient de a o implementa. Dacă am implementa update-urile brut, am obține o complexitate de $O(n^2)$. Așadar, căutăm un mod de a reprezenta ofertele care să permită o complexitate mai bună.

b) Observăm că ar fi bine să grupăm toți pirații care au S_j^i egal. Astfel, notăm P_f^i ***numărul de pirați care primesc exact*** f ***bănuți pe oferta*** S^i. Folosind noua notație, supralicitările din update se pot rescrie mai frumos ca $S_j^i = 1 + S_j^{i+1} \rightarrow P_f^i = P_{f-1}^{i+1}$.

c) Motivația din spatele introducerii acestui P este micșorarea numărului de operații din cadrul unui update de ofertă: în loc să incrementăm, pe rând, costul tuturor voturilor cu același preț, putem incrementa costul grupei P_f^i o singură dată.

d) Presupunem că deja am efectuat $P_f^i = P_{f-1}^{i+1}$. În acest moment, toți pirații, în afară de $i+1$, votează oferta. Dar probabil nu avem nevoie de așa de multe voturi, așa că le putem elimina pe cele mai scumpe până rămânem cu strictul necesar de A_i. Iterăm descrescător după f toate P_f^i-urile nenule și mutăm pirații ale căror voturi nu sunt necesare în P_0^i. Observăm că va exista maxim un singur P_f^i ce va fi mutat doar parțial în P_0^i (deoarece este posibil ca să avem nevoie de numai o parte din voturile piraților care primesc f bănuți).

e) Datorită faptelor că:

- Update-ul $P_f^i = P_{f-1}^{i+1}$ poate refolosi vectorul deja existent P^{i+1} ce trebuie doar mutat o poziție la dreapta,
- Se modifică în $P_f^i = 0$ numai f-urile din capătul cu elementul maxim,
- Se micșorează P_f^i-ul nenul, unde f este maximul după eliminările precedente,
- Se introduce P_0^i la capătul cu elementul minim,

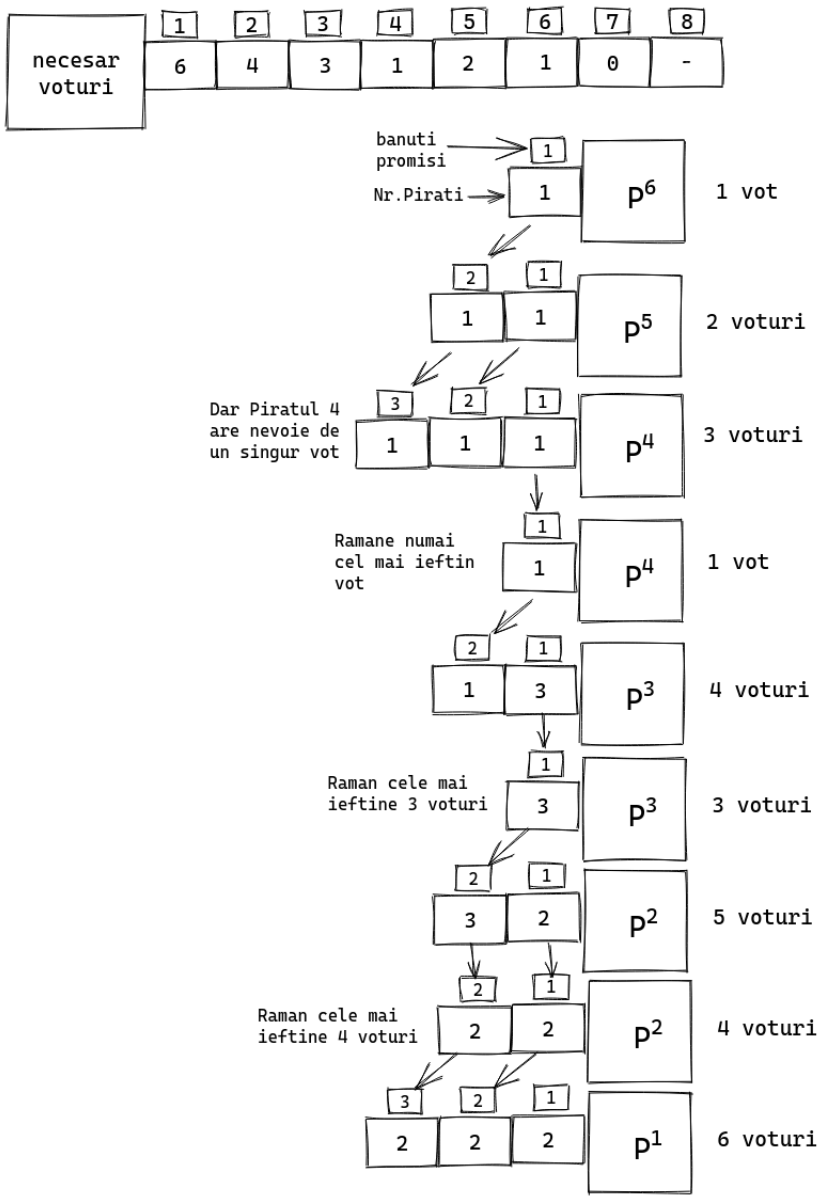
rezolvarea pare să aibă complexitate $O(n)$ dacă folosim structura de date numită *coadă*.

f) Dimensiunea lui P_0^i este singura care variază la fiecare pas. Putem să introducem o variabilă care să contorizeze numărul de pirați ce votează oferta la un moment de timp (variabila *total_pirati_in_coada* din codul de mai jos). Pentru P_0^i, scădem *total_pirati_in_coada* din numărul total de pirați la acel moment.

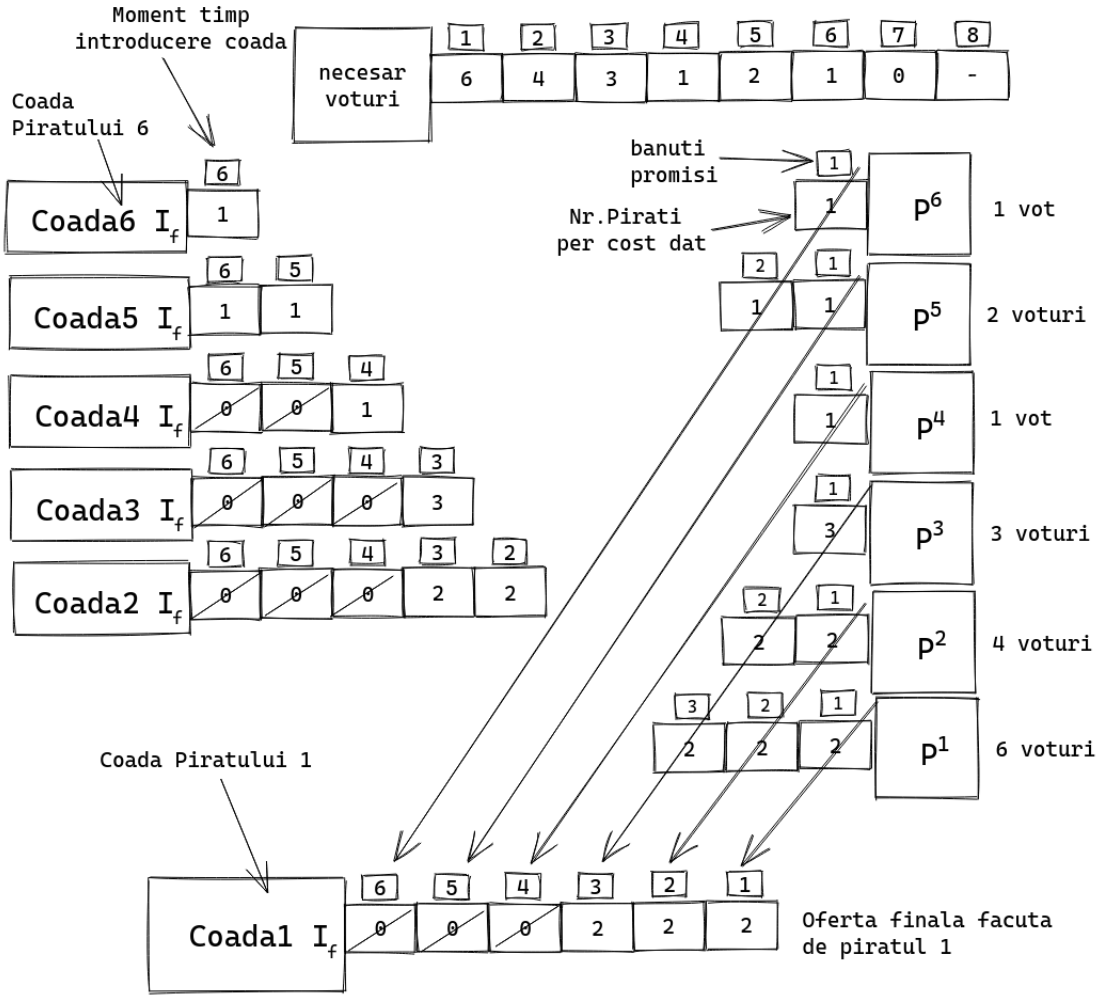
g) Observăm că, în general, $P_f^i = P_{f-1}^{i+1} = P_{f-2}^{i+2} = ... = P_1^{i+f-1}$. Putem să ne folosim de acest fapt ca să reducem complexitatea soluției. Introducem o variabilă I_i ce reprezintă numărul de pirați care primesc exact 1 bănuț în oferta făcută de i. Deoarece câștigul promis fiecărui pirat crește cu 1 la fiecare update, știm că toți pirații din grupa P_f^i au fost introduși la oferta $i+f-1$, când li s-a promis 1 bănuț. Astfel, $P_f^i = P_{f-1}^{i+1} = ... = P_1^{i+f-1} = I_{i+f-1}$, iar

în loc să actualizăm P-urile la fiecare iterație, shiftăm la dreapta I-urile în coadă, fiindcă ele rămân în aceeași ordine.

Schițăm ideea ilustrată în punctul de mai sus:



Ilustrăm mai jos cum arată aceste transformări, de la oferta ultimului până la cea a primului pirat, când iau forma pe care o vor lua în cod, într-o coadă:



h) O mică problemă a cozii din *std* :: *queue* este că aceasta nu permite modificări de elemente la capătul de *pop*(). Totuși, noi avem nevoie să modificăm și acel unic I_f ce își micșorează valoarea doar parțial. O variantă ar fi să folosim *std::deque*, dar acesta este foarte lent și ar fi bine să-l evităm dacă nu avem nevoie de întreaga lui putere. O abordare simplă ar fi să simulăm coada cu un vector normal: insert-ul să îl facem cu *push_back()*, iar *pop()*-ul prin incrementarea unei variabile numită *stanga_coada* (vezi codul de mai jos). Atenție la folosirea acestei tehnici, deoarece vectorul ar putea ajunge la dimensiuni extrem de mari în cazul a mai mult de *un* insert per update.

i) Rămâne doar să justificăm că această implementare rulează în $O(n)$. Deși este posibil ca la un update să ștergem multe grupuri nenule I_f, numărul total de grupuri șterse nu poate depăși numărul total de grupuri introduse. Reamintim faptul că la fiecare update introducem un singur grup: I_i (adică precedentul P_0^i), deci în total vor fi cel mult n inserări de-a lungul întregului joc. Din acest motiv complexitatea este $O(n)$ amortizat.

Cod Sursă

```
struct cluster_pirati {
    int num_pirati;
    int indice_inserare;
};
int main() {
    int n;
    cin >> n;
    vector<int> reputatie(n);
    for (int i = 0; i < n - 2; ++i) {
        cin >> reputatie[i];
    }
    int stanga_coada = 0;
    int total_pirati_in_coada = 0;
    vector<cluster_pirati> q;
    for (int i = n - 3; i >= 0; i--) {
        int num_pirati_de_ignorat = n - 2 - i - reputatie[i];
        while (q.size() > stanga_coada &&
               q[stanga_coada].num_pirati <= num_pirati_de_ignorat) {
            num_pirati_de_ignorat -= q[stanga_coada].num_pirati;
            total_pirati_in_coada -= q[stanga_coada].num_pirati;
            ++stanga_coada;
        }
        if (q.size() > stanga_coada &&
               q[stanga_coada].num_pirati > num_pirati_de_ignorat) {
            q[stanga_coada].num_pirati -= num_pirati_de_ignorat;
            total_pirati_in_coada -= num_pirati_de_ignorat;
            num_pirati_de_ignorat = 0;
        }
        if (reputatie[i] > total_pirati_in_coada) {
            q.push_back({reputatie[i] - total_pirati_in_coada, i});
            total_pirati_in_coada = reputatie[i];
        }
    }
    long long suma_primita = 10000000000;
    for (int i = q.size() - 1; i >= stanga_coada; i--) {
        suma_primita -= (q[i].indice_inserare + 1) * q[i].num_pirati;
    }
    cout << suma_primita << "\n";
    return 0;
}
```

Surse se pot trimite la `https://infoarena.ro/problema/caraibe`