Tabăra de pregătire a lotului național de informatică - juniori, Craiova 9-14 mai
Baraj 1
Descrierea soluțiilor

Comisia științifică

May 11, 2025

Problema 1. Rețete

Propusă de: prof. Emanuela Cerchez, Colegiul Național „Emil Racoviță" Iași
stud. Andrei Boacă, Facultatea de Informatică, Universitatea „Al. I. Cuza" Iași

Vom citi linie cu linie ingredientele din inventar. Pentru a extrage ingredientele, o soluție simplă este de a parcurge șirul de la final spre început:

- căutăm ultimul spațiu (acesta delimitează unitatea de măsură); pentru a nu avea probleme cu cantități exprimate în unități de măsură diferite, convertim l, dl, cl în ml, iar kg în g;
- căutăm următorul spațiu, acesta delimitează cantitatea;
- restul șirului reprezintă denumirea produsului.

Ingredientele din lista-inventar le vom memora într-o structură de date, împreună cu cantitățile în care sunt disponibile. Structura de date pe care o utilizați pentru memorarea ingredientelor influențează eficiența implementării.

Cerința 1

Vom citi succesiv rețetele și pentru fiecare rețetă verificăm dacă toate ingredientele specificate în rețeta respectivă apar în inventar într-o cantitate suficient de mare.

Cerința 2

Vom nota cu P numărul de rețete care pot fi preparate individual. Pentru cerința 2 se garantează că există cel mult 18 rețete care pot fi preparate individual ($P \leq 18$). Prin urmare, vor exista cel mult $C_{18}^{9} = 48\,620$ combinații fezabile.

Această observație ne sugerează că este o simplă problemă de generare de elemente combinatoriale, care poate fi abordată în diferite moduri. În funcție de modul de abordare și de implementare pot fi obținute diferite punctaje.

Soluția 1. Generare submulțimi

Se generează toate submulțimile de rețete și pentru fiecare submulțime se verifică dacă este fezabilă (adică dacă toate ingredientele care apar în rețetele respective există în inventar, iar suma cantităților necesare pentru fiecare ingredient este mai mică sau egală cu cantitatea disponibilă).

Această soluție are complexitate $O(2^P \cdot P \cdot I)$, unde I este numărul maxim de ingrediente din fiecare rețetă ($I \leq 100$), datorită faptului că pentru fiecare submulțime generată trebuie să verificăm dacă suma cantităților ingredientelor care apar în submulțime este disponibilă.

Soluția 2. Generare combinări

Se generează combinări (submulțimi de K rețete dintre cele P care pot fi preparate), în ordine descrescătoare după K) și se verifică, în același mod, pentru fiecare combinare dacă este fezabilă. La primul K pentru care există soluții fezabile ne oprim, întrucât acestea sunt combinații cu număr maxim de rețete. Această abordare este mai eficientă decât prima, dar nu obține 100 de puncte.

Soluția 3. Generare cod Gray

Pentru început vom „normaliza” șirurile de caractere, asociind fiecăruia câte un număr, pentru a putea scăpa de un eventual factor de log în implementarea ideii de mai jos. Normalizarea o vom face folosind un `std::map<std::string , int>` asociind fiecărui șir de caractere (ținut în memorie pe tipul de date `std::string`) neîntâlnit până la momentul curent un nou număr.

Vom genera submulțimile de rețete sub formă de măști pe biți în ordinea codului Gray. Codul $Gray(K)$ (codul $Gray$ pe K biți) se obține în felul următor:

1. $Gray(1)$ este 0,1.
2. $Gray(K)$ pentru $K > 1$ se obține din $Gray(K-1)$ completând codul $Gray(K-1)$ cu un bit cu valoarea 0, apoi concatenând cele 2^{K-1} soluții cu oglindirea codului $Gray(K-1)$ completată cu un bit cu valoarea 1.

De exemplu, $Gray(2)$ poate fi:
00
01
11
10

$Gray(3)$ poate fi:
000
001
011

010
110
111
101
100

Codul Gray poate fi calculat și direct cu formula `x ^ (x >> 1)`.

Avantajul folosirii acestui mod de generare a submulțimilor este dat de faptul că valorile de la două poziții vecine diferă prin exact un bit, motiv pentru care va trebui să actualizăm suma cantităților ingredientelor din rețetele submulțimii curente doar pentru rețeta care apare/dispare din submulțimea de la pasul anterior. Obținem astfel o complexitate $O(2^P \cdot I)$, care este suficientă pentru punctajul maxim.

Soluția 4. Backtracking optimizat

O abordare de tip backtracking optimizat pentru generarea soluțiilor poate obține, de asemenea, punctaj maxim.

Problema 2. Tort

Propusă de: instr. Cristian Frâncu, Nerdvana București

Observație: Orice grosime $2^n \times a + m \times b$ poate fi obținută prin îndoiri și ungeri cu unt, deoarece putem să efectuăm n îndoiri și apoi m ungeri.

Cerința 1

- Vom încerca toate variantele pentru n, deoarece 2^n va depăși rapid c.
- Pentru un n dat calculăm m minim astfel încât $2^n \times a + m \times b \geq c$.
- Astfel:

  $$m = (c - 2^n \times a)/b$$

  dacă împărțirea este exactă sau

  $$m = (c - 2^n \times a)/b + 1$$

  dacă împărțirea este cu rest.
- Astfel, formula finală a lui m este

  $$m = (c - 2^n \times a + b - 1)/b$$

- Grosimea tortului va fi:

$$g = 2^n \times a + m \times b$$

Înlocuind:

$$g = 2^n \times a + (c - 2^n \times a + b - 1)/b \times b$$

Algoritmul pentru determinarea grosimii minime a tortului este:

```
1: citește a, b, c
2: grosime_minima ← ∞
3: n ← 0
4: while 2^n × a ≤ c do
5:     g ← 2^n × a + (c − 2^n × a + b − 1)/b × b
6:     if g < grosime_minima then
7:         grosime_minima ← g
8:     end if
9:     n ← n + 1
10: end while
11: afișează grosime_minima
```

Complexitatea este $O(\log c)$ ca timp și $O(1)$ memorie.

Cerința 2

Observație: O combinație $2^n \times a + m \times b$ se obține în număr minim de operații astfel:

- Avem n îndoiri.
- Vrem să obținem cele m straturi de unt din cât mai puține aplicări.
- Pentru aceasta trebuie să „strecurăm” puterile lui 2 din m printre îndoiri.
- Este posibil ca n să fie prea mic, caz în care vom adăuga mai multe puteri ale lui 2 din m la început.

Cu alte cuvinte, pentru o combinație $2^n \times a + m \times b$ vom obține numărul minim de operații astfel:

- La început vom unge k straturi de unt, unde $k = m/2^n$.
- Apoi la fiecare dublare vom unge un singur strat doar dacă puterea corespunzătoare a lui 2 din m există în m (are bit 1 în reprezentarea binară a lui m).

- Informatic spus, numărul minim de mutări este:

$$n + m/2^n + \text{popcount}(m \mod 2^n)$$

  unde `popcount(x)` este numărul de biți 1 din reprezentarea binară a lui x.

Algoritmul pentru determinarea numărului minim de operații este:

```
1:  citește a, b, c
2:  grosime_minima ← ∞
3:  n ← 0
4:  while 2^n × a ≤ c do
5:      m ← (c − 2^n × a + b − 1)/b
6:      g ← 2^n × a + m × b
7:      if g < grosime_minima then
8:          grosime_minima ← g
9:          nrop_minim ← n + m/2^n + popcount(m mod 2^n)
10:     else
11:         if g = grosime_minima then
12:             operatii = n + m/2^n + popcount(m mod 2^n)
13:             if operatii < nrop_minim then
14:                 nrop_minim ← operatii
15:             end if
16:         end if
17:     end if
18:     n ← n + 1
19: end while
20: afișează nrop_minim
```

Complexitatea este $O(\log c \times p)$ ca timp și O(1) memorie, unde p este timpul de calcul al funcției `popcount(x)`. Cu o metodă brută p este $O(\log x)$, dar există și metode mai rapide ce duc la timp $O(\log \log x)$.

Observăm că putem folosi această implementare pentru a rezolva și prima cerință.

Anexă

Algoritmul de mai sus, implementat eficient, se va încadra în timp, dar la limită, existând riscul să depășim timpul cu o implementare mai puțin eficientă „de concurs”. Ce putem face?

Să observăm că fișierul de intrare va fi foarte mare, trei milioane de numere. Precum știm (sau nu :-) citirea cu *streams* din C++ sau cu `fscanf` în C este destul de lentă. Putem citi mai rapid aceste numere folosind o combinație de funcție de bibliotecă `fread()` și prelucrare a caracterelor, numită „citire rapidă” sau „parsing”, în limbajul olimpicilor.

Iată mai jos un exemplu de cod ce citește rapid un întreg **`unsigned long long`**.

Listing 1 Exemplu de citire rapidă

```
#define BUFSIZE (128 * 1024)

FILE *fin, *fout;
int rpos = BUFSIZE - 1; char rbuf[BUFSIZE];

static inline char readChar() {
  if ( !(rpos = (rpos + 1) & (BUFSIZE - 1)) )
    fread( rbuf, 1, BUFSIZE, fin );
  return rbuf[rpos];
}

unsigned long long readInt() {
  int ch;
  unsigned long long res = 0;

  while ( isspace( ch = readChar() ) );
  do
    res = 10 * res + ch - '0';
  while ( isdigit( ch = readChar() ) );

  return res;
}
```

Această citire poate de fi de două până la patru ori mai rapidă decât citirea standard.

Problema 3. Zid

Propusă de: Prof. Ionel-Vasile Piț-Rada, Colegiul Național Traian, Drobeta Turnu Severin

Multe rezolvări pentru cerința 1 pornesc de la următoarea observație. Oricând avem $h_i > h_{i+1}$, cumva trebuie să-l aducem pe h_{i+1} la nivelul lui h_i (cel puțin). Nu ajută să creștem perechea (h_i, h_{i+1}), căci diferența dintre ele se va păstra. De aceea, este necesar să creștem perechea (h_{i+1}, h_{i+2}) cu valoarea $h_i - h_{i+1}$.

Aceste creșteri sunt strict necesare: nu putem să egalizăm h_i și h_{i+1} fără ele. Deci, dacă problema are soluție, orice algoritm care aplică astfel de creșteri va ajunge la soluție.

Subtaskul 1

Fie p și q, $p < q$, pozițiile celor două înălțimi 1 și fie $A = [p+1, q-1]$ și $B = [q+1, p-1]$ intervalele (circulare) dintre ele. Apar trei posibilități.

1. Dacă N este impar, atunci exact unul dintre A și B are lungime pară, să spunem B. Atunci pe B îl aducem la înălțimea 1 cu creșteri din două în două poziții. Pe A îl umplem similar

și va rămîne o înălțime 0, să zicem pe poziția $p+1$. Mai creștem o dată perechea $(p, p+1)$ și obținem un zid cu o înălțime 2 și restul 1. Numărul de înălțimi 1 este par, deci putem completa zidul la înălțime 2.

2. Dacă N este par, iar A și B au lungimi pare, atunci putem completa zidul la înălțime 1.

3. Dacă N este par, iar A și B au lungimi impare, atunci putem crește perechi din A și din B pînă rămîn doar două înălțimi de 0, să zicem la pozițiile $p+1$ și $q+1$. Dacă acum creștem perechile $(p, p+1)$ și $(q, q+1)$, obținem un zid cu înălțimi 2 pe pozițiile p și q și 1 în rest. Adică am ajuns la problema originală crescută cu o cărămidă pe toate pozițiile. Dar poate există alte creșteri care duc la o soluție? Răspunsul este că nu, deoarece orice creștere are loc pe o poziție pară și o poziție impară. Suma pe pozițiile de aceeași paritate cu p și q va fi mereu cu 2 mai mare decît suma pe pozițiile de paritate opusă. Problema nu are soluție decît dacă eliminăm cei doi de 1 inițiali.

Reținem de aici observația-cheie că, dacă N este par, atunci problema are soluție doar dacă suma pe pozițiile pare este egală cu suma pe pozițiile impare.

Subtaskul 2

Dacă zidul este nedescrescător, iar problema are soluție, atunci singurul mod în care N poate fi par este că perechile de poziții 1-2, 3-4, 5-6 etc. au înălțimi egale. Deci putem crește aceste perechi pentru a aduce zidul la înălțimea h_N.

Dacă N este impar, putem parcurge zidul de la N la 1. Creștem fiecare înălțime h_i la valoarea h_N, crescînd corespunzător și poziția h_{i-1}. La final, cînd creștem h_1 la înălțimea h_N, h_N va crește și el pînă la o nouă înălțime H. Acum, avem o poziție de înălțime H și un număr par $(N-1)$ de poziții de înălțime h_1, pe care le putem crește în perechi pînă la H. Așadar, răspunsul este H, iar complexitatea este $O(N)$.

Subtaskurile 3 și 4

Putem aplica același principiu și la un vector de formă oarecare. În mod repetat, căutăm minimul y. Fie x și z vecinii săi cu $x \geq z$. Atunci îl aducem pe y la înălțimea lui x, crescînd perechea (y, z) cu valoarea $x - y$.

Remarcăm că aceste soluții fac efort proporțional cu înălțimea finală a zidului. Această înălțime poate fi $N/2$, de exemplu pentru vectorul $1010\ldots101$.

Pentru subtaskul 3 este suficientă o implementare în $O(\log N)$ per creștere. Putem menține un `std::set` cu coloanele ordonate după înălțime. La înălțimi egale preferăm coloana cu vecinul cel mai înalt, ca să ne asigurăm că nu alegem o coloană de mijloc dintr-un platou de valori egale. Este nevoie de atenție la implementare, întrucît, la creșterea unei perechi (a, b), atît elementele, cît și ceilalți doi vecini ai lor își schimbă criteriul de ordonare, deci toate patru trebuie șterse și reinserate în structură. Complexitatea este $O(\text{hmin} \cdot N \cdot \log N)$.

Putem îmbunătăți constanta acestei implementări, păstrînd complexitatea asimptotică, dacă grupăm în permanență platourile de coloane egale. Pentru a aduce un triplet x_1, x_2, x_3 cu $x_1 \geq x_2 \leq x_3$ la aceeași înălțime vom face următoarele operații:

- Aducem x_1 și x_3 la aceeași înălțime prin creșterea celui mai mic dintre ele, apelând o operație de incrementare care să cuprindă atât elementul mai mic, cât și x_2.
- Alternăm între operația (x_1, x_2) și operația (x_2, x_3) pentru a aduce x_2 la nivel cu x_1 și x_3.

Acum în loc de trei coloane egale putem păstra una singură, deoarece pe celelalte două le vom crește împreună.

Pentru a trece și subtaskul 4, putem reduce timpul de găsire a minimului la $O(1)$ dacă menținem pentru fiecare înălțime o listă înlănțuită cu coloanele de acea înălțime. Complexitatea este $O(\text{hmin} \cdot N)$.

Subtaskul 5

Notăm cu x_k valoarea care se adaugă la turnurile h_k și h_{k+1}. Deoarece toate turnurile vor avea aceeași înălțime vom avea:

$$\begin{aligned} & h_1 + x_N + x_1 \\ = {} & h_2 + x_1 + x_2 \\ = {} & h_3 + x_2 + x_3 = \ldots \\ = {} & h_{N-1} + x_{N-2} + x_{N-1} \\ = {} & h_N + x_{N-1} + x_N \end{aligned} \tag{1}$$

Dacă pentru (1) avem o soluție $x = (x_1, x_2, \ldots, x_N)$ care va produce înălțimea finală H, atunci adăugând / scăzând o valoare arbitrară d din toate valorile lui x vom obține o altă soluție $y = (x_1 + d, x_2 + d, \ldots, x_N + d)$ care va produce înălțimea finală $H + 2 \cdot d$.

Cazul N impar

Reducând fiecare dintre egalitățile din (1) obținem relațiile:

$$\begin{aligned} x_1 &= h_N - h_1 + x_{N-1} \\ x_2 &= h_1 - h_2 + x_N \\ x_3 &= h_2 - h_3 + x_1 \\ x_4 &= h_3 - h_4 + x_2 \\ &\ldots \\ x_{N-1} &= h_{N-2} - h_{N-1} + x_{N-3} \\ x_N &= h_{N-1} - h_N + x_{N-2} \end{aligned} \tag{2}$$

Fixăm $x_1 = 0$, din care rezultă x_3, apoi $x_5, \ldots, x_N$ (N impar), apoi $x_2, x_4, \ldots, x_{N-1}$ ($N - 1$ par). Nu putem accepta valori negative, astfel că vom calcula $x_{min} = \min\{x_k \mid 1 < k \leq N\}$ și vom scădea valoarea x_{min} din toate valorile x_k, $1 \leq k \leq N$. Această soluție va produce înălțimea minimă $h_{min} = h_1 + x_N + x_1$.

Cazul N par

Relațiile (2) determină două grupuri, ecuațiile cu necunoscute cu indici impari (3) și cele cu necunoscute cu indici pari (4):

$$\begin{aligned} x_1 &= h_N - h_1 + x_{N-1} \\ x_3 &= h_2 - h_3 + x_1 \\ x_{N-1} &= h_{N-2} - h_{N-1} + x_{N-3} \end{aligned} \tag{3}$$

$$\begin{aligned} x_2 &= h_1 - h_2 + x_N \\ x_4 &= h_3 - h_4 + x_2 \\ x_N &= h_{N-1} - h_N + x_{N-2} \end{aligned} \tag{4}$$

Se observă că, dacă adunăm toate relațiile din grupul (3), atunci toate variabilele x se reduc, rezultând egalitatea:

$$h_1 + h_3 + \cdots + h_{N-1} = h_2 + h_4 + \cdots + h_N \tag{5}$$

Observăm din nou că există soluție doar dacă datele de intrare respectă egalitatea (5). Dacă nu, atunci problema ne cere să eliminăm un număr minim de cărămizi. Așadar, răspunsul este dat de diferența în valoare absolută dintre cele două sume.

Rezolvăm separat fiecare grup de relații inițializând $x_1 = 0$ și respectiv $x_2 = 0$. Pentru (3) calculăm $x_{min1} = \min\{x_k \mid 1 \le k \le N, k \text{ impar}\}$ și apoi scădem x_{min1} din fiecare x_k, k impar. Pentru (4) calculăm $x_{min2} = \min\{x_k \mid 1 \le k \le N, k \text{ par}\}$ și apoi scădem x_{min2} din fiecare x_k, k par. Această soluție va produce înălțimea minimă $h_{min} = h_1 + x_N + x_1$.

Complexitate $O(N)$.

Subtaskul 5, soluția 2

Dacă problema are soluție, există și o soluție greedy. Parcurgem zidul de la 1 la N și, oricînd avem $h_i > h_{i+1}$ creștem perechea (h_i, h_{i+1}) cu valoarea $h_i - h_{i+1}$. Astfel obținem un vector ordonat și reducem problema la subtaskul 2.

De aceea sînt suficiente două treceri prin vector. Complexitatea este $O(N)$.

Echipa

Problemele pentru această etapă au fost pregătite de:

- Prof. Adrian Panaete, Colegiul Național „A.T. Laurian” Botoșani
- Prof. Ciprian Chește, Liceul Tehnologic „Grigore C. Moisil” Buzău
- Instr. Cristian Frâncu, Nerdvana București
- Instr. Cătălin Frâncu, Nerdvana București

- Stud. Andrei Boacă, Facultatea de Informatică, Universitatea „Alexandru Ioan Cuza”, Iași
- Prof. Mihai Bunget, Colegiul Național Tudor Vladimirescu, Târgu-Jiu
- Prof. Gheorghe-Eugen Nodea, Centrul Județean de Excelență Gorj, Târgu-Jiu
- Prof. Emanuela Cerchez, Colegiul Național „Emil Racoviță” Iași
- Stud. Alin Răileanu, Facultatea de Informatică, Universitatea „Alexandru Ioan Cuza”, Iași
- Stud. Victor Botnaru, Facultatea de Automatică și Calculatoare, Universitatea Națională de Știință și Tehnologie Politehnica București
- Prof. Ionel-Vasile Piț-Rada, Colegiul Național Traian, Drobeta Turnu Severin
- Stud. Răzvan Alexandru Rotaru, Facultatea de Informatică, Universitatea „Alexandru Ioan Cuza”, Iași
- Stud. Rareș-Andrei Cotoi, Universitatea Babeș-Bolyai, Cluj, Facultatea de Matematică și Informatică
- Stud. Giulian Buzatu, Facultatea de Matematică-Informatică, Universitatea București
- Prof. Dan Pracsiu, Liceul Teoretic Emil Racoviță, Vaslui
- Prof. Marinel Șerban, Colegiul Național „Emil Racoviță” Iași
- Stud. Petruț-Rareș Gheorghieș, Facultatea de Automatică și Calculatoare, Universitatea Națională de Știință și Tehnologie Politehnica București
- Stud. Ioan-Cristian Pop, Facultatea de Automatică și Calculatoare, Universitatea Națională de Știință și Tehnologie Politehnica București