TABĂRA DE PREGĂTIRE A LOTULUI NAȚIONAL DE INFORMATICĂ
CLUJ, 10-15 MAI 2024
BARAJ 2
DESCRIEREA SOLUȚIILOR

COMISIA ȘTIINȚIFICĂ

Problema 1: Copaci

Propusă de: lect. Paul Diac, Facultatea de Informatică, Universitatea Alexandru Ioan Cuza din Iași

Soluție $O(N^2 \times |S|)$. Pentru primul pas al plimbării lui Dan, putem construi o matrice de poziții accesibile $a[i][j]$ în care să reținem valoarea 1 pentru coordonate la care ar putea fi primul copac de la care pleacă Dan, și 0 în rest. Valorile 1 vor fi doar la coordonate la care copacii au cifra scrisă egală cu prima cifră $S[0]$. Pentru pasul al doilea, putem construi o nouă matrice similară, cu valori 1 la coordonatele la care Dan ar putea fi la al doilea pas. La aceste coordonate trebuie să fie copaci cu cifra scrisă următoare, adică $S[1]$, dar coordonatele să fie și vecine cu coordonate în care matricea $a[][]$ are valoarea 1. Astfel se completează plimbarea lui Dan cu al doilea pas în continuarea copacilor care ar fi putut fi primii în plimbare. Mai departe, pentru fiecare pas, putem costrui astfel o matrice de poziții accesibile, dar fiecare pas i se bazează doar pe matricea de la pasul anterior $i-1$ și cifra $S[i]$. Astfel, putem reține doar ultimele două matrici. Pentru simplificarea implementării putem construi o matrice $a[0..1][1..N][1..N]$ unde prima coordonată reprezintă paritatea pasului curent iar următoarele două corespund liniilor și coloanelelor pădurii. Paritatea se schimbă de la un pas la altul, astfel la pasul curent i construim accesibilitatea în matricea $a[i\%2][][]$ folosind matricea de la pasul anterior $a[!(i\%2)][][]$. Rezultatul va fi cea mai mare valoare i pentru care avem măcar un element nenul în matricea $a[i\%2][][]$. Parcurgând matricea la fiecare pas și iterand fiecare din cei 4 vecini pentru fiecare element, obținem o complexitate de $O(N^2 \times |S|)$ și aproximativ 60-70 puncte.

Soluție $O(N^2 \times |S| \ / \ 64)$. Putem compresa informația din matricea $a[][][]$ folosind numere întregi ai căror biți au valorile 0 și 1 cu aceeași semnificație ca în soluția de bază. Folosind tipul *long long*, putem grupa într-un singur element 64 de valori consecutive de pe o aceeași linie. Cifrele scrise pentru fiecare copac în parte sunt relevante și ele pentru fiecare coloană, dar le putem reprezenta tot prin mulțimi de biți, precalculate pentru fiecare cifră posibilă în parte. Pentru că avem doar zece cifre, memoria este suficientă. Vecinii de deasupra și de dedesubt se pot procesa printr-un *sau* pe mulțimi de biți: *or* adică operatorul |. Vecinii din stânga și dreapta strict din interiorul unei mulțimi de 64 biți se pot procesa folosind operatorii de shift-are pe biți $a[...][...][...] << 1$ și $a[...][...][...] >> 1$, în combinație cu *si* adică & pentru mulțimile de biți ale cifrei $S[i]$. Vecinii din capetele unei mulțimi de 64 biți trebuie tratați separat, este posibil ca un drum să se continue dintr-o coloană care este inclusă în gruparea din stânga sau dreapta grupării curente. Dar având doar două capete, sunt doar două operații în plus. Implementări care folosesc grupări de mai puțini biți sau *bitset* <> din STL ar putea obține punctaje parțiale sau maxime în funcție și de alte detalii de implementare.

Problema 2: PMO

Propusă de: Prof. Ciprian Chește, Liceul Tehnologic "Grigore C. Moisil", Buzău
Stud. Dumitru Ilie, Facultatea de Matematică-Informatică, Universitatea București

Soluția 1 - stud. Dumitru Ilie

Vom prezenta soluția pentru un singur X_i. Vom începe prin a factoriza numărul. Pentru a face asta rapid ne vom folosi de ciurul lui Eratostene pentru a găsi numerele prime mai mici decât $\sqrt{VAL\_MAX} \approx 32000$. Avem aproximativ 3400 de numere prime ce ne interesează.

O observație importantă este faptul că 2 se comportă la fel ca 3, 5, 7, 11, și orice alt număr prim, având o singură partiție multiplicativ ordonată. Mai mult, $2 \cdot 3 = 6$ se comportă la fel ca $2 \cdot 5 = 10$, $2 \cdot 7 = 14$, $3 \cdot 5 = 15$, și orice alt produs de două numere prime, având 3 partiții multiplicativ ordonate. Acest fapt se poate extinde. În general pentru un număr $p_1^{e_1} \cdot p_2^{e_2} \cdot p_3^{e_3} \cdot \ldots \cdot p_k^{e_k}$ (unde p_i este număr prim) nu ne interesează decât exponenții numerelor prime, e_1, e_2, e_3, ..., e_k. Mai mult, orice rearanjare a acestor exponenți oferă același număr de partiții, deci le putem rearanja într-un mod convenabil, mai exact descrescător. De exemplu, numărul $2^1 \cdot 7^5 \cdot 11^2 = 4067294$ va fi redus la o listă de 3 numere, mai exact $[5, 2, 1]$.

Datorită observației precedente putem calcula o singură dată răspunsul pentru toate configurațiile de exponenți și apoi obține în timp foarte bun numărul de partiții folosindu-ne de procesarea anterioară. Trebuie sa avem grijă ca numărul de configurații să nu fie prea mare, încât să nu depășim limita de timp doar preprocesând sau limita de memorie din cauza numărului mare de date necesare. Într-adevar numărul configurațiilor este scăzut, aproximativ 1300 (Determinarea numărului exact este un exercitiu interesant pe care vă sugerăm să îl încercați). Pentru a obține prima dată numărul de partiții al unei configurații putem folosi metoda backtracking pentru a itera prin toate configurațiile "incluse" în configurația curentă și să adăugăm numărul de partiții obținut prin eliminarea acelor elemente. De exemplu, configurația $[5, 2, 1]$ poate fi "ruptă" în:

- $[5, 2, 1] - [1, 0, 0] = [4, 2, 1]$;
- $[5, 2, 1] - [2, 0, 0] = [3, 2, 1]$;
- $[5, 2, 1] - [3, 0, 0] = [2, 2, 1]$;
- $[5, 2, 1] - [4, 0, 0] = [1, 2, 1] = [2, 1, 1]$ (prin convenție le ordonăm descrescător);
- $[5, 2, 1] - [5, 0, 0] = [2, 1, 0] = [2, 1]$ (zerourile nu ne interesează);
- $[5, 2, 1] - [0, 1, 0] = [5, 1, 1]$;
- $[5, 2, 1] - [1, 1, 0] = [4, 1, 1]$;
- $[5, 2, 1] - [2, 1, 0] = [3, 1, 1]$;
- $[5, 2, 1] - [3, 1, 0] = [2, 1, 1]$;
- $[5, 2, 1] - [4, 1, 0] = [1, 1, 1]$;
- $[5, 2, 1] - [5, 1, 0] = [1, 1]$;
- $[5, 2, 1] - [0, 2, 0] = [5, 1]$;
- $[5, 2, 1] - [1, 2, 0] = [4, 1]$;
- $[5, 2, 1] - [2, 2, 0] = [3, 1]$;
- $[5, 2, 1] - [3, 2, 0] = [2, 1]$;
- $[5, 2, 1] - [4, 2, 0] = [1, 1]$;
- $[5, 2, 1] - [5, 2, 0] = [1]$;
- $[5, 2, 1] - [0, 0, 1] = [5, 2]$;
- $[5, 2, 1] - [1, 0, 1] = [4, 2]$;
- $[5, 2, 1] - [2, 0, 1] = [3, 2]$;
- $[5, 2, 1] - [3, 0, 1] = [2, 2]$;
- $[5, 2, 1] - [4, 0, 1] = [2, 1]$;
- $[5, 2, 1] - [5, 0, 1] = [2]$;
- $[5, 2, 1] - [0, 1, 1] = [5, 1]$;
- $[5, 2, 1] - [1, 1, 1] = [4, 1]$;
- $[5, 2, 1] - [2, 1, 1] = [3, 1]$;
- $[5, 2, 1] - [3, 1, 1] = [2, 1]$;

- $[5,2,1] - [4,1,1] = [1,1]$;
- $[5,2,1] - [5,1,1] = [1]$;
- $[5,2,1] - [0,2,1] = [5]$;
- $[5,2,1] - [1,2,1] = [4]$;
- $[5,2,1] - [2,2,1] = [3]$;
- $[5,2,1] - [3,2,1] = [2]$;
- $[5,2,1] - [4,2,1] = [1]$;
- $[5,2,1] - [5,2,1] = []$;

Soluția 2 - prof. Ciprian Chește

Se știe că numărul de soluții ordonate ale ecuației $x_1 \cdot x_2 \cdot ... \cdot x_n = T$ este egal cu

$$\prod_{i=1}^{k} \binom{\alpha_i + n - 1}{n-1} \quad (1)$$

unde $\alpha_i, 1 \leq i \leq k$ reprezintă exponenții descompunerii numărului T, în factori primi.

Un număr T poate fi descompus ca partiție multiplicativă ordonată în produs de $1, 2, 3, ..., \alpha_1 + \alpha_2 + ... + \alpha_k$ termeni. Așadar putem aplica relația (1) pentru toate valorile lui n cuprinse între 1 și $\alpha_1 + \alpha_2 + ... + \alpha_k$. Trebuie însă avut în vedere că în relația (1) sunt incluse și soluțiile care-l conțin pe 1 ca factor, ceea ce înseamnă că acestea trebuie scăzute din numărul total calculat. Soluțiile cel conțin pe 1 ca factor și au k termeni pot fi calculate combinatoric în funcție de soluțiile anterioare, adică cele cu $k-1$ termeni.

Exemplu: $T = 12 = 2^2 3^1$.

Sunt 8 partiții multiplicative ordonate ale lui 12:

$12, 2 \cdot 6, 6 \cdot 2, 3 \cdot 4, 4 \cdot 3, 2 \cdot 2 \cdot 3, 2 \cdot 3 \cdot 2, 3 \cdot 2 \cdot 2$.

- Dacă 12 ar fi scris ca produs de 1 termen atunci relația (1) furnizează $\binom{2+1-1}{1-1} \cdot \binom{1+1-1}{1-1} = \binom{2}{0} \cdot \binom{1}{0} = 1$ soluție.
- Dacă 12 ar fi scris ca produs de 2 termeni atunci relația (1) furnizează $\binom{2+2-1}{2-1} \cdot \binom{1+2-1}{2-1} = \binom{3}{1} \cdot \binom{2}{1} = 6$ soluții. Din acest număr trebuie scăzute $1 \cdot \binom{2}{1} = 2$ deoarece există o singură soluție și 1 poate fi așezat la această soluție în $\binom{2}{1}$ poziții. Deci pentru acest caz avem $6 - 2 = 4$ soluții.
- Dacă 12 ar fi scris ca produs de 3 termeni atunci relația (1) furnizează $\binom{2+3-1}{3-1} \cdot \binom{1+3-1}{3-1} = \binom{4}{2} \cdot \binom{3}{2} = 18$ soluții. Din acest număr trebuie scăzute $1 \cdot \binom{3}{2}$ deoarece există o singură soluție cu un singur factor și 2 de 1 pot fi așezați în poziții și să mai scădem $4 \cdot \binom{3}{1}$ deoarece există 4 soluții scrise ca produs de 2 factori și un 1 poate fi așezat în $\binom{3}{1}$ poziții. Deci pentru acest caz avem $18 - 3 - 12 = 3$ soluții.

În total avem $1 + 4 + 3 = 8$ soluții.

Soluția 3 - prof. Adrian Panaete

Spunem că două numere sunt echivalente dacă au același număr de divizori primi și multiseturile exponenților pentru cele două numere coincid.

Observație: Dacă două numere A și B sunt echivalente atunci acestea vor avea exact același număr de descompuneri distincte.

Demonstrație: Într-adevăr, dacă $A = a_1^{E_1} * a_2^{E_2} * ... * a_m^{E_m}$ cu a_i numere prime distincte putem scrie $B = b_1^{E_1} * b_2^{E_2} * ... * b_k^{E_m}$ unde b_i sunt eventual alte numere prime.

Considerând o descompunere oarecare $A = A_1 * A_2 * .. * A_p$ și fiecare factor A_j se poate scrie că $A_j = a_1^{e_1} * a_2^{e_2} * ... * a_m^{e_m}$. Corespunzător acelui factor putem construi pentru B factorul $B_j = b_1^{e_1} * b_2^{e_2} * ... * b_m^{e_m}$ și astfel avem o descompunere $B = B_1 * B_2 * ... * B_p$ care corespunde în mod unic descompunerii alese pentru A.

În particular pentru un număr oarecare A putem asocia cea mai mică valoare echivalentă cu A $CA = Canonic(A)$ astfel : fie E_1, E_2,..., E_m aleși în ordine descrescătoare $CA = p_1^{E_1} * p_2^{E_2} * ... * p_m^{E_m}$ unde $p_1, p_2, p_3... = 2, 3, 5, ...$ = șirul numerelor prime.

Observație: $m \leq 9$ deci echivalentul minim va avea cel mult factorii primi 2, 3, 5, 7, 11, 13, 17, 19, 23 și mai mult se poate dovedi că exponenții corespunzători sunt respectiv cel mult 29, 11, 6, 3, 2, 2, 1, 1, 1.

În final se poate astfel dovedi că nu există decât un număr relativ mic de numere canonice (puțin peste 1000) și orice alt număr va fi echivalent cu unul dintre aceste numere.

Acest lucru ne conduce spre următorul algoritm. - se factorizează numărul pe care vrem să îl rezolvăm - se ordonează descrescător exponenții - se contruiește numărul canonic minim echivalent cu numărul inițial - se rezolvă numărul canonic.

Cum rezolvăm valorile canonice? Formula de rezolvare este valabilă pentru orice număr (nu numai pentru valorile canonice).

Fie A un număr. Notăm $sol[A]$ = numărul de descompuneri pentru A. În primul rând, dacă A are un singur factor prim atunci avem caz particular și se poate observă că $sol[A]$ este o putere de 2. De asemenea, dacă A este număr liber de pătrate putem trata particular problema cu un backtracking (există de fapt doar 9 astfel de numere canonice). Ca o observăție - soluția pentru numerele libere de pătrate sunt numere cunoscute sub denumirea de numere Fubini(irelevant pentru problema noastră)

Fie o descompunere $A = A_1 * A_2 * .. * A_m$. Pentru A_1 fixat se realizează toate descompunerile numărului $A_2 * A_3 * ... * A_m = \frac{A}{A_1}$ în număr de $sol[\frac{A}{A_1}]$. Se observă că astfel $sol[A]$ se obține ca sumă din $sol[\frac{A}{A_1}]$ pentru orice A_1 divizor diferit de 1 al lui A în particular pentru că formula să funcționeze se va lua $sol[1] = 1$ deoarece pentru $A_1 = A$ descompunerea va conține doar un singur termen. Această idee ne va conduce la un algoritm în care pentru a avea soluția pentru un număr A avem nevoie de soluțiile pentru fiecare divizor al lui A.

Cum putem optimiza acest algoritm ? În primul rând când rezolvăm un număr A vom avea de rezolvat toți divizorii lui A ceea ce ne arată că de fapt ar trebui să recalculăm același lucru de mai multe ori. Pentru a rezolva această situație vom memoiza orice valoare pe care o rezolvăm la un moment dat astfel încât când ajungem în rezolvare la un număr canonic care a fost rezolvat deja să nu fim nevoiți să recalculăm. Memoizarea ne reduce mult din operațiile pe care le avem de efectuat, dar tot trebuie să parcurgem toți divizorii numărului de rezolvat pentru fiecare număr canonic care nu a fost încă rezolvat.

Putem totuși să reducem numărul de divizori pe care trebuie să îl accesăm dacă urmărim cu atenție cum se formează mulțimea divizorului unui număr. Astfel, dacă alegem un divizor foarte mare (obținut de exemplu prin împărțirea lui a cu un factor prim A_i) mulțimea divizorilor acestui număr se regăsește în mulțimea divizorilor lui A, astfel dacă în loc să adunăm la $sol[A]$ valoarea $sol[\frac{A}{7}A_i]$ adunăm $2 * sol[\frac{A}{7}A_i]$ vom adună de fapt nu numai pentru divizorul $\frac{A}{A_i}$ ci pentru toți divizorii acestuia care la rândul lor sunt divizori pentru A, deci trebuie să apară în soluție.(Ideea este aproximativă ... calculul trebuie făcut cu atenție). În final, ținând cont că același lucru de va întâmplă pentru toți factorii primi A_i, dacă adunăm $2 * sol[\frac{A}{A_i}]$ avem $sol[A]$ la care se adaugă de mai multe ori valori $sol[D]$, unde D pentru divizorii $\frac{A}{A_i}$. Analizând cu atenție cum putem elimina ce avem în plus se observă că după ce adunăm termenii $2 * sol[\frac{A}{A_i}]$ va trebui să scădem $2 * sol[\frac{A}{A_i * A_j}]$ unde A_i, A_j sunt doi factori primi ai lui A, apoi să adunăm $2 * sol[\frac{A}{\text{produs de 3 factori primi distincți ai lui A}}]$, apoi să scădem $2 * sol[\frac{A}{\text{produs de 3 factori primi distincți ai lui A}}]$ și așa mai departe.

Cu alte cuvinte, pentru a calcula $sol[A]$:

- se adună $2 * sol[\frac{A}{2}]$, $2 * sol[\frac{A}{3}]$...$2 * sol[\frac{A}{23}]$ (numai pentru factorii primi ai lui A)

- se scade $2 * sol[\frac{A}{6}]$, $2 * sol[\frac{A}{15}]$...$2 * sol[\frac{A}{19*23}]$ ($sol[A/(p * q)]$ dar numai pentru p și q factori primi pentru A)

Și așa mai departe.

Această soluție va trebui să acceseze pentru un număr A cu M factori primi distincți doar 2^M divizori (ceea ce înseamnă mult mai puțin decât numărul total de divizori ai lui A)

Am obținut astfel un algoritm similar ca idee cu principiul includerii și excluderii (PINEX).

În final mai putem aminti de o optimizare a factorizarii numerelor care poate fi aplicată nu doar la această soluție ci în general la orice factorizare a numerelor (inclusiv la soluțiile anterioare). Optimizarea nu trebuie aplicată pentru a obține punctajul maxim dar poate ameliora foarte mult timpul de execuție.

FACTORIZARE RAPIDĂ PENTRU NUMERE PÂNĂ LA 1000000000

.

Se aplică Ciurul lui Eratostene până la $R = \sqrt{1000000000} = 32622$.

La marcajul numerelor în ciur vom folosi marcajul: Un factori prim al lui i (în loc să marcăm cu 1).

Vom construi în paralel șirul numerelor prime până la R (de fapt vom vedea ulterior că nu ne sunt necesare decât cele până la 1000).

Odată făcute precalculările putem factoriza numerele folosind succesiv următoarele idei:

- Cazul 1: Factorizarea numerelor $X \leq R$

  Deoarece pentru fiecare valoare $X \leq R$ cunoaștem un factor prim $P[X]$ determinăm exponentul lui $P[X]$ și simultan simplificăm numărul prin $P[X]$.

  Repetăm cu valoarea simplificată a lui X până când X ajunge la valoarea 1. Numărul de pași în acest caz este egal cu suma exponenților factorilor primi ai lui X (supraestimat 29+11+6+3+2+2+1+1+1)
- CAZUL 2: Factorizarea numerelor X > R

  Pasul 1: Folosind numerele prime ≤ 1000 în total 162 numere) se calculează factorii primi mai mici decât 1000 cu exponenții lor și evident se simplifică X prin acești factori primi.

  Pasul 2: Se aplică dacă numărul avea factori primi ≥ 1000 ceea ce e echivalent cu faptul că X nu a devenit 1 în pasul anterior. Evident că valoarea rămasă X va mai conține doar factori primi > 1000 deci numărul factorilor primi distincți sau nedistincți este cel mult 2 adică $X = p$ sau $X = p^2$ sau $X = p * q$ cu p, q numere prime și $p, q > 1000$
  - subcazul 2.1: dacă $X < 1000000$ atunci X nu poate să aibă doi factori < 1000 deci mai avem un singur factor prim cu exponentul 1
  - subcazul 2.2: dacă $X \geq 100000$ și X este pătrat perfect -> $X = p^2$ deci mai avem un factor prim cu exponentul 2
  - subcazul 2.3: se aplică dacă numărul X nu a fost factorizat încă. Se observă că acum suntem strict în una din două situații - $X = p$ sau $X = p * q$ cu p și q numere prime $p, q > 1000$ (în cazul $X = p$, $p > 1000000$). Pentru a distinge între cele două situații aplicăm testul de primalitate Rabin-Miller. Deoarece $X \leq 1000000000$ este suficient să aplicăm testul doar pentru bazele 2, 3, 5, 7 pentru că testul devine determinist pentru numere ≤ 1000000:
    * dacă testul validează că X este număr prim atunci mai avem un factor prim cu exponentul 1
    * dacă testul spune că X este compozit atunci mai avem încă doi factori primi cu exponentul 1.

În cazul problemei noastre nu avem nevoie efectiv de factorizare ci doar de exponenții fiecărui factor prim. Astfel în ultimul caz analizat nu avem efectiv nevoie de valorile p și q din scrierea $X = p * q$ ci doar de informația că avem doi exponenți 1.

Dacă în alt context(la o altă problema) chiar am dori factorizarea completă, observăm că singura nedeterminare a algoritmului este necunoașterea celor doi factori p și q de la cazul menționat. În acest caz putem determina unul dintre factori aplicând numărului X algoritmul Pollard's RHO.

O ultimă optimizare se referă la divizorii numerelor canonice pe care le întâlnim pe parcursul rezolvării. Acești divizori conțin sigur factori primi ≤ 23 doar că nu știm dacă exponenții sunt ordonați descrescător.

Pentru a avea formă canonică la aceste numere este suficient să iterăm printre cele 9 valori prime de la 2 la 23 și să determinăm exponenții acestora. Prin reordonarea descrescătoare a

acestor exponenți putem canoniza și acești divizori astfel că în memoizarea aplicată în mod efectiv acel număr apropiat de 1000 de valori de valori distincte ale soluției.

Problema 3: Pokemoni

Propusă de: prof. Gheorghe-Eugen Nodea

Analizăm inițial traseul ales de pokemonul Fire, traseu care influențează traseul pokemonului Water.

Soluție 1 - $O(P^2)$ - 50p

Reținem cele P coordonate ale punctelor speciale în vectorul Ps. Sortăm crescător după coordonata x, iar în caz de egalitate după coordonata y punctele speciale.

Problema se reduce la determinarea lungimii subșirului crescător maximal (LIS).

Pentru a determina lungimea maximă a unui subșir crescător al lui Ps, vom construi un șir suplimentar L, cu proprietatea că $L[i]$ este lungimea maximă a unui subșir care începe în $Ps[i]$. Atunci lungimea maximă LEN a unui subșir crescător va fi cel mai mare element din vectorul L.

Vom reține în vectorii $st[]$, respectiv $dr[]$ cea mai din stânga, respectiv dreapta poziție a fiecărei lungimii $L[i]$ găsită în șirul Ps, cu $1 \leq i \leq LEN$.

Determinarea traseului minim lexicografic ales de Fire, notat cu T, presupune parcurgem vectorul L, de la stânga la dreapta, plecând de la poziția $st[LEN]$ către poziia $dr[LEN]$.

Dacă $L[i] = LEN - 1$ iar $Ps[i].x \geq T[LEN].x$ și $Ps[i].y \geq T[LEN].y$ atunci adăugăm la traseu punctul $Ps[i]$, iar $LEN = LEN - 1$. Această parcurgere se efectuează în $O(N)$ amortizat.

La final, se elimină traseul pokemonului Fire din vectorul punctelor speciale Ps.

Se aplică același algoritm pentru traseul pokemonului Water.

Soluție 2 - $O(N \times M)$ - 65-85p

Pentru a determina traseul parcurs de Fire putem reține într-o matrice de tip bool punctele speciale de pe hartă: $Sp[x][y] = 0$ sau 1, după cum punctul (x, y) este punct special sau nu.

$best[x][y]$ = numărul maxim de puncte ce se vizitează pe un traseu optim de la $(0, 0)$ la (x, y)

$best[0][0] = Sp[0][0]$

$best[x][0] = Sp[x][0] + best[x-1][0]$, pentru $x > 0$

$best[0][y] = Sp[0][y] + best[0][y-1]$, pentru $y > 0$

$best[x][y] = Sp[x][y] + max(best[x-1][y], best[x][y-1])$, pentru $x, y > 0$

Rezultatul va fi furnizat de $best[x][y]$. Din modul cum se construiește matricea $best$ se obține traseul minim lexicografic prin marcarea punctelor speciale vizitate.

Evident, se poate comprima matricea c astfel încât să se rețină liniile și coloanele care conțin puncte speciale. Această observație aduce puncte în plus.

Soluție 3 - $O(P \times log(P))$ - 100p

Sortăm punctele ca în Soluția 1. Determinăm subșirul crescător maximal, folosindu-ne de căutare binară. Pentru aceasta, definim doi vectori auxiliari b și dp, unde:

- $b[i]$ - reprezintă un element al șirului de puncte în care se termină un subșir crescător maximal de lungime i
- $dp[i]$ - lungimea celui mai lung subșir crescător maximal care se termină cu elementul din șirul de puncte aflat la poziția i

Din moment ce șirul b va fi întotdeauna crescător, căutăm binar valoarea lui $dp[i]$, și vom actualiza valoarea lui $b[dp[i]]$ cu punctul curent. În funcția de comparare, va trebui ca ambele coordonate să fie mai mici decât cele ale punctului curent.

Pentru reconstituirea traseului, iterăm de la finalul vectorului dp. Dacă cumva valoarea la un anumit indice i este egală cu lungimea maximă a unui traseu obținut, vom lua numărul de pe poziția respectivă, scădem 1 din lungimea traseului, și continuă iterația.

După determinarea traseului lui Fire, se elimină punctele din șirul dat, și se repetă algoritmul pentru Water.

Însă, prin această metodă, obținem șirul maxim lexicografic, din cauza reconstituirii acestuia. Pentru a evita această problemă, vom sorta inițial șirul în ordine descrescătoare din punct de vedere lexicografic, și vom determina subșirul descrescător maximal.

Soluție 4 - $O(P \times log(P))$ - 100p

Sortăm punctele în ordine lexicografică. Apoi, pentru fiecare punct cu coordonatele (x_i, y_i), vrem să determinăm care este cel mai bun punct (x_p, y_p) cu $x_p \leq x_i$ și $y_p \leq y_i$ cu lungimea drumului maximă. Diferit față de soluția precedentă, putem să căutăm valoarea maximă a unei lungimi de drum de pe intervalul $[1, y_i]$ întrucât procesăm punctele în ordine lexicografică (deci liniile nu pot fi restricții).

Ne dorim să construim un vector v unde $v[j]$ reprezintă lungimea maximă a unui drum care are ultimul punct situat pe coloana j.

Așadar, fiecare punct (x_i, y_i) va avea două tipuri de operații:

(1) Query: care punct are drumul maxim cu $y_p \leq y_i$. (deci maximul dintre $v[1], v[2], \ldots, v[j]$
(2) Update: toate punctele de pe intervalul $[y_p, m]$ se pot lipi de punctul curent, deci se va modifica $v[j]$. Cu alte cuvinte, maximul de pe intervalul $v[j], v[j+1], \ldots, v[m]$ va fi influențat de $v[j]$

Observăm ca această problemă se poate rezolva cu arbori de intervale, dar și cu arbori indexați binari (întrucât query-urile sunt strict pe intervale cu capătul stânga 1).

Pentru a genera drumul minim lexicografic, în arbore vom ține și indicele punctului cu lungimea maximă și comparăm pe caz de egalitate.

Acestă soluție funcționează numai pentru valori scăzute ale lui N.

Echipa

Problemele pentru această etapă au fost pregătite de:

- Prof. Emanuela Cerchez, Colegiul Național "Emil Racoviță" Iași
- Prof. Adrian Panaete, Colegiul Naţional "August Treboniu Laurian", Botoşani
- Prof. Ciprian Chește, Liceul Tehnologic "Grigore C. Moisil", Buzău
- Prof. Ionel-Vasile Piț-Rada, Colegiul Naţional "Traian", Drobeta Turnu Severin
- Prof. Mihai Bunget, Colegiul Național "Tudor Vladimirescu", Târgu Jiu
- Lector dr. Paul Diac, Facultatea de Informatică, Universitatea "Alexandru Ioan Cuza" Iaşi
- Prof. Dan Pracsiu, Liceul Teoretic "Emil Racoviță", Vaslui
- Prof. Gheorghe Eugen Nodea, Centrul Județean de Excelență Gorj, Târgu Jiu
- Prof. Daniela Elena Lica, Centrul Județean de Excelență Prahova, Ploiești
- Stud. Ioan-Cristian Pop, Universitatea Politehnica București
- Stud. Dumitru Ilie, Facultatea de Matematică-Informatică, Universitatea București
- Stud. Giulian Buzatu, Facultatea de Matematică-Informatică, Universitatea București
- Stud. Mircea Măierean, Facultatea de Matematică și Informatică, Universitatea "Babeș-Bolyai", Cluj-Napoca