# RoAlgo Cup I - Descrierea soluțiilor

Comisia științifică

*Problema 1: Numere magice*

În această problem ni se cere să răspundem la mai multe întrebări de tipul: „Câte numere naturale din intervalul [st, dr] au 3 divizori impari?”.

**Subtask-ul 1:** Se aplică un raționament asemănător cu cel de la sume parțiale. Pentru fiecare număr de la 1 la 1.000 se precalculează câte numere naturale cu 3 divizori impari există mai mici sau egale decât el. Pentru a răspunde la întrebări, se scade din valoarea precalculată în $dr$, valoarea precalculată în $st - 1$.

**Subtask-ul 2:** Un număr cu 3 divizori impari se poate scrie sub forma $2^k \cdot p^2$, unde k este număr natural, iar p este un număr prim diferit de 2 (cei 3 divizori impari vor fi 1, p și $p^2$). Pentru a răspunde la o întrebare, vom calcula pentru fiecare k câte numere de forma $2^k \cdot p^2$ mai mici sau egale decât $dr$ și $st - 1$ există. Adică, câte numere prime mai mici sau egale decât $\sqrt{\frac{dr}{2^k}}$ și $\sqrt{\frac{st-1}{2^k}}$ există. Scăzând a doua valoarea din prima, vom obține răspunsul pentru un k fixat. Răspunsul final va fi suma pentru fiecare k. Pentru a calcula rapid câte numere prime mai mici sau egale decât o valoare există, vom putea precalcula rapid cu Ciurul lui Eratostene răspunsul pentru fiecare valoare de la 1 la $10^6$.

*Problema 2: Gomory și scândurile de lemn*

În această problem ni se dă un șir de lungime n, în care fiecare valoare este un număr natural cuprins între 1 și n, sau -1. Ni se cere să afișăm numărul de modalități de a înlocui valorile -1 cu numere naturale între 1 și n astfel încât să existe element majoritar în șir.

**Subtask-ul 1:** Pentru acest subtask este destul să generam toate șirurile posibile și să le număram pe cele care au element majoritar. Complexitate $O(n^n)$

**Subtask-ul 2:** Mai întâi vom alege valoarea elementului majoritar, fie ea $i$. Apoi, vom fixa de câte ori vrem să apară in sir, iar din moment ce vrem să fie majoritar, acest număr trebuie să fie mai mare sau egal decât $\left\lfloor \frac{n}{2} \right\rfloor + 1$. Restul pozițiilor le

putem completa cu oricare din cele n - 1 valori ramase. Astfel, răspunsul pentru $i$ va fi:

$$\sum_{j=\left\lfloor \frac{n}{2} \right\rfloor + 1}^{n} \binom{n}{j} \cdot (n-1)^{n-j}$$

Observăm că răspunsul va fi același pentru oricare $i$ de la 1 la n, deci rezultatul final va fi:

$$n \cdot \sum_{j=\left\lfloor \frac{n}{2} \right\rfloor + 1}^{n} \binom{n}{j} \cdot (n-1)^{n-j}$$

Complexitate $O(n)$

**Subtask-ul 3:** Vom nota numărul de apariții ale valorii -1 cu k. La fel ca în subtask-ul anterior, vom fixa mai întâi valoarea elementului majoritar, $i$. Apoi vom avea 2 cazuri:

1. $i$ este deja element majoritar: în acest caz, putem transforma fiecare -1 cu orice valoare, deci răspunsul va fi:

$$\sum_{j=0}^{k} \binom{k}{j} \cdot (n-1)^{k-j}$$

2. $i$ nu este element majoritar: în acest caz, va trebui mai întâi să ne asigurăm că $i$ apare de cel puțin $\left\lfloor \frac{n}{2} \right\rfloor + 1$ ori. Astfel, cel putin req $= \left\lfloor \frac{n}{2} \right\rfloor + 1 - f_i$, unde $f_i$ este numărul de apariții ale valorii $i$ în șir. Astfel, răspunsul va fi:

$$\sum_{j=req}^{k} \binom{k}{j} \cdot (n-1)^{k-j}$$

Dacă req > k, această sumă va da 0.

Pentru a afla rezultatul final, trebuie să adunăm răspunsurile pentru fiecare $i$ de la 1 la n.

Complexitate $O(n^2)$

**Subtask-ul 4:** Uitându-ne la răspunsurile de la subtask-ul anterior, putem observa că expresia din interiorul sumei rămâne aceeași, indiferent de valoarea lui $i$, ci doar capetele se schimbă. Putem face următoarea sumă parțială:

$$sp[e] = \sum_{j=0}^{e} \binom{k}{j} \cdot (n-1)^{k-j} = sp[e-1] + \binom{e}{j} \cdot (n-1)^{e-j}$$

Astfel, putem afla răspunsul pentru fiecare *i* în *O(1)* cu *O(n)* precalculare.

Complexitate *O(n)*, dar se acceptă și soluțiile $O(n\sqrt{n})$.

*Problema 3: Conspirație*

În această problemă se dă un arbore cu n noduri. Fiecare nod are câte o culoare c. Inițial c[i]=i pentru toate nodurile. La citirea datelor se dau și cele n – 1 muchii într-o ordine care contează. Va urma un număr infinit de update-uri, unde fiecare update este reprezentat de o muchie din input. Primul update este pentru prima muchie din input, al doilea update pentru a doua muchie din input etc. Când se ajunge la operație n se începe din nou de la prima muchie. La un update ai o muchie și ești obligat sa schimbi fie culoarea nodului u în culoarea nodului v, fie invers. Trebuie să afișezi numărul minim de update-uri astfel încât să faci toate nodurile să aibă aceeași culoare.

*Observație principală:* Dacă dorim să aducem toate nodurile la aceeași culoare *i*, este optim să facem operațiile de update în felul următor:

1. Dacă ambele noduri au culoarea *i*, este irelevant ce facem
2. Dacă ambele noduri au culoarea diferită de *i*, este irelevant ce facem
3. Dacă unul dintre noduri are culoarea *i*, iar celălalt are culoarea diferită de *i*, atunci vom face ca ambele noduri să aibă culoarea *i*

Astfel, culoarea *i* se propagă încet în tot arborele.

**Subtask-ul 1:** Pentru acest subtask, este suficient să iterăm prin fiecare culoare posibilă și să vedem numărul de update-uri necesare pentru a aduce întregul arbore la acea culoare. Să presupunem că vrem să colorăm tot arborele în culoarea x. Pornim un DFS din nodul x (pe care îl considerăm rădăcină) și definim următorul dp:

- dp[i] = numărul minim de operații astfel încât să colorăm nodul *i* în culoarea x
- dp[x] = 0

Dacă definim id(u-v) ca fiind indicele muchiei u-v din input (indexate de la 0), pentru toți vecinii lui x: dp[vecin] = id(x-vecin) + 1. Pentru celelalte noduri, să notăm unul dintre ele *u*, fie *t* tatăl lui u și $t_2$ tatăl lui *t*:

- Dacă id($t_2$-t) < id(t-u): dp[u] = dp[t] + id(t-u) - id($t_2$-t)
- Dacă id($t_2$-t) > id(t-u): dp[u] = dp[t] + n - id($t_2$-t) + id(t-u)

Răspunsul pentru fiecare culoare va fi maximul dintre aceste valori. Culoarea cu cea mai mică valoare maximă ne va da răspunsul.

**Subtask-ul 2:** Pentru acest subtask, nu ne permitem să simulăm problema pentru fiecare culoare, deci ne trebuie o strategie mai eficientă.

Pentru a pune în valoare culoarea fiecărui nod în
parte fără a face efectiv n DFS-uri, putem folosi un DP cu rerooting. Deci, punem ca rădăcină nodul 1 și definim dp1[i] = numărul de zile necesare pentru a aduce nodul i și întreg subarborele său la culoarea tatălui lui i, acesta fiind t. Inițial, dp1[i] = id(i-t) + 1. Apoi, pentru fiecare fiu al lui i:

- Dacă id(i-t) < id(fiu-i): dp1[i] devine maximul dintre dp1[i] și dp1[fiu]
- Dacă id(i-t) > id(fiu-i): dp1[i] devine maximul dintre dp1[i] și dp1[fiu] + n – 1

De asemenea, definim dp2[i] = numărul de zile necesare pentru a aduce toate nodurile din arbore care nu se află în subarborele lui i la culoarea nodului i. Inițial, dp2[i] = id(i-t) + 1.

Notăm tatăl lui t ca $t_2$. Dacă t > 1, dp2[i] va deveni maximul dintre dp2[i] și:

- Dacă id(i-t) < id(t- $t_2$), cu dp2[t]
- Dacă id(i-t) > id(t- $t_2$), cu dp2[t] + n - 1

De asemenea, pentru fiecare frate al lui i (fiu al aceluiași tată), dp2[i] va deveni maximul dintre dp2[i] și:

- Dacă id(i-t) < id(frate-t), cu dp1[frate]
- Dacă id(i-t) > id(frate-t), cu dp1[frate] + n - 1

Pentru a evita o complexitate *O(nr_fii$^2$)*, care poate duce la *O(n$^2$)*, putem procesa fiii în ordine crescătoare / descrescătoare a id-urilor id(fiu-t), pentru a preprocesa maximul valorilor dp1 pe prefix / sufix. Astfel, maximizările din partea fraților vor fi făcute în *O(1)*.

Răspunsul va fi min(max(dp2[nod], dp1[fiu] pentru orice fiu al lui nod)) pentru orice nod.

*Problema 4: Gomory, Hu și pietricelele colorate*

În această problemă trebuie să găsim valoarea așteptată (EV) a numărului de secvențe maximale cu elemente egale a unui șir, luând în considerare toate permutările cu repetiție ale acestuia.

**Subtask-ul 1:** În acest subtask este destul să generam toate permutările și să facem media numărului de secvențe maximale cu elemente egale.

Complexitate *O(n!)*

**Subtask-ul 2:** Din moment ce avem maxim două valori distincte, putem fixa numărul de secvențe maximale cu elemente egale și să numărăm permutările cu acel număr de secvențe folosind Stars and Bars. Pentru fiecare număr de secvențe fixat, adunăm într-o variabilă numărul de secvențe înmulțit cu numărul de permutări cu acel număr de secvențe, iar la sfârșit luăm media împărțind la numărul total de permutări.

Complexitate *O(n)*

**Subtask-ul 3:** Pentru fiecare k de la 1 la n, se calculează numărul de permutări în care numărul de secvențe maximale cu elemente egale este k și se face media pe baza acestora. Pentru a calcula rapid, se procedează asemănător ca la Șir Dacic (https://kilonova.ro/problems/611).

Complexitate $O(n^2)$

**Subtask-ul 4:** O primă observație este că numărul de secvențe maximale dintr-un șir este egal cu numărul de perechi de elemente consecutive distincte + 1.
Astfel, noi vrem de fapt să calculam 1 + EV(numărul de perechi de elemente consecutive distincte). Sau altfel spus, 1 + EV($\sum_{i=1}^{n-1}(a_i \neq a_{i+1})$), care este egal cu $1 + \sum_{i=1}^{n-1} EV(a_i \neq a_{i+1})$.
Din moment ce EV($a_i \neq a_{i+1}$) = EV($a_1 \neq a_2$), pentru oricare $1 \leq i < n$, rezultatul este egal cu: $1 + (n - 1) \cdot \text{EV}(a_1 \neq a_2)$. Dar, EV($a_1 \neq a_2$) = 1 - EV($a_1 = a_2$) = 1 – P($a_1 = a_2$).

Drept urmare, trebuie să calculăm probabilitatea ca 2 elemente vecine sa fie egale. Vom nota cu $f_i$ numărul de apariții ale valorii $i$ în șir. Astfel, avem:

nr_cazuri_favorabile = $\sum_{i=1}^{n} \binom{f_i}{2}$

nr_cazuri_totale = $\binom{n}{2}$

P($a_1 = a_2$) = nr_cazuri_favorabile / nr_cazuri_totale

Răspunsul final va fi: $1 + (n - 1) \cdot \left(1 - P(a_1 = a_2)\right)$.

Complexitate *O(n)*

*Problema 5: Grădinarul Hu*

În această problemă ni se dau două drepte într-un plan și trebuie să găsim o a treia dreaptă care împreună cu cele două drepte date să conțină k puncte laticeale (cu coordonate numere întregi). De asemenea, vârfurile triunghiului trebuie să fie puncte laticiale.

**Subtask-ul 1:** Pentru acest subtask este destul doar să alegem 2 puncte din plan, să verificăm dacă aparțin dreptelor, iar apoi să numărăm toate punctele laticiale din triunghiul pe care îl definesc. Din moment ce numărul de puncte de pe fiecare dreaptă este maxim n, vom avea maxim $n^2$ perechi de puncte valide. Astfel, ajungem la un algoritm de complexitate $O(n^4)$.

**Subtask-ul 2:** Vom optimiza modul în care număram punctele din soluția de mai sus. Știind ecuațiile celor 3 drepte ale triunghiului, putem afla pentru fiecare x de la 1 la n cate puncte de abscisa x in *O(1)*. Astfel, vom avea complexitate $O(n^3)$.

**Subtask-ul 3:** Vom optimiza din nou soluția de la subtask-ul 1. Folosind Teorema lui Pick (https://en.wikipedia.org/wiki/Pick%27s_theorem), putem calcula răspunsul pentru orice triunghi cu vârfurile în puncte laticiale în *O(1)*.

$$\text{Puncte din triunghi} = \frac{\text{Arie} + \text{Numărul de puncte de pe laturi}}{2} + 1$$

Numărul de puncte de pe latură a triunghiului delimitată de punctele ($x_1$, $y_1$) și ($x_2$, $y_2$) este |cmmdc(x1 - x2, y1 - y2)|.

Astfel, vom reduce complexitatea la $O(n^2)$.

**Subtask-ul 4:** Să notăm dreptele date ca d1 și d2. Pe acestea vom alege 2 puncte: p1, respectiv p2. Putem observa că dacă fixăm pe p1, numărul de puncte din

triunghi crește odată cu îndepărtarea lui p2 de intersecție. Astfel, putem folosi un algoritm de tip Two Pointers pentru a alfa răspunsul în *O(n)*. Se acceptă și soluții care folosesc căutarea binară în *O(n log n)*.

*Problema 6: Subșiruri importante*

Problema ne cere ca pentru un șir de lungime n să afișăm câte submulțimi există în care diferența maximă dintre oricare două elemente să fie ≤ k și diferența maximă dintre oricare două poziții să fie ≤ d.

**Subtask-ul 1:** Pentru acest subtask este suficient să facem un brut pe măști de biți, verificând câte submulțimi sunt bune.

**Subtask-ul 2:** În acest subtask, singura restricție care contează este k. Putem sorta valorile în ordine crescătoare și folosi algoritmul Two Pointers pentru a determina pentru fiecare valoare câte valori există în șir în intervalul [val - k, val] (exceptând valoarea respectivă și cele egale care au indici mai mari în șirul sortat, pentru a nu face overcounting). Fie numărul de valori x. Contribuția acestui element la răspuns este $2^x$.

**Subtask-urile 3,4:** Vom menține ideea de Two Pointers după valori. Astfel, vom activa treptat valorile în ordine crescătoare, dezactivând elementele care ajung să aibă valoarea mai mică decât valoarea nou adăugată (fie ea *val*) - k. Înainte să activăm un element de pe poziția p, îi calculăm contribuția în felul următor: pentru fiecare poziție *poz* de la p - d la p - 1, unde se află un element activat, există $2^x$ subșiruri în care elementul de pe poziția poz este cel mai din stânga și cel mai mare element ca valoare este *val*, unde x este numărul de elemente activate din intervalul [poz + 1, poz + d].

**Subtask-ul 5:** Pentru a optimiza problema, vom defini șirul dp: dp[i] = $2^x$, unde x este numărul de elemente activate din intervalul [i+1, i + d] dacă
elementul de pe poziția i este activat, iar dp[i] = 0 dacă elementul de pe poziția i nu este activat.

Vom crea un AINT pe acest șir. Astfel, operațiile devin:

1. Contribuție: Sumă pe interval
2. Activare element: Înmulțire cu 2 pe interval / Schimbare element
3. Dezactivare element: Împărțire la 2 pe interval (Înmulțire cu 500.000.004) / Schimbare element

*Problema 7: Criptomonede*

În această problemă pornim de la un șir $a$ de lungime n, în care fiecare element are valoarea 0. Pe acest șir se vor aplica k operații de incrementare asupra unor poziții. La a $i$-a operație ni se dă un subșir $S_i$, de dimensiune $c_i$, din care trebuie să alegem exact $p_i$ elemente. Pentru fiecare element x din cele $p_i$ elemente alese, vom incrementa $a_x$ cu 1. Trebuie să afișăm suma produselor elementelor șirului, pentru fiecare modalitate de a alege operațiile de incrementare.

**Subtask-ul 1:** Se vor genera cu backtracking toate modalitățile de a efectua cele k operații și se va calcula produsul elementelor șirului $a$ pentru fiecare dintre acestea. La final se afișează suma produselor.

**Subtask-urile 2,3,4:** În loc să ne gândim ca la un șir în care incrementăm unele elemente, vom reformula problema astfel încât elementele șirului să fie mulțimi, inițial vide, iar în loc să adăugăm 1 la un element din șir, vom adăuga un element în mulțime (nu este important ce element). La final ni se cere să calculăm produsul cardinalelor mulțimilor și să afișăm suma pentru toate modalitățile de a efectua cele k operații. Astfel, produsul cardinalelor mulțimilor are aceeași valoare cu numărul de modalități în care putem alege din fiecare mulțime câte un element.
Problema se poate rezolva folosind programare dinamică pe biți. Vom defini următoarea dinamică dp[i][mask] ca fiind numărul de modalități de a alege elemente din mulțimile date de masca mask în primele i operații. Tranziția va fi următoarea:

$$dp[i][mask] = \sum_{S\ inclus\ în\ S_i\ și\ în\ mask} dp[i\ -\ 1][mask\ \wedge\ S] \cdot \binom{c_i\ -\ popcount(S)}{p_i\ -\ popcount(S)}$$

Diferența dintre subtask-uri este dată de modalitatea de parcurgere a submăștilor lui $S_i$. Pentru subtask-ul 2, este destul să enumerați submăștile formate dintr-un singur bit, din moment ce $p_i = 1$. Pentru subtask-ul 3 se acceptă soluții de complexitate $O(4^n \cdot k)$, iar pentru subtask-ul 4 se cere o enumerare rapidă a submăștilor cu un algoritm final de complexitate $O(3^n \cdot k)$

*Problema 8: Advanced Persistent Threat*

Problema spune că într-un graf s-a ales un arbore parțial ascuns care are muchiile orientate de la sursă la fii, iar concurentul trebuie să găsească rădăcina arborelui ascuns folosind cât mai puține întrebări de tipul: „În ce direcție este orientată muchia x – y în arborele ascuns?"

Restricțiile și modalitatea de generare a testelor – care pot fi văzute în enunțul problemei – au fost alese astfel încât să se evite soluția în care se alege un nod random și se pun întrebări pe toți vecinii săi, iar dacă nodul nu are sursă în arborele ascuns, atunci el este rădăcina, altfel continuăm cu nodul sursă până ce ajungem la rădăcină.

Există idei diverse care nu ar trebui să ia un punctaj prea mare. Câteva dintre ele includ: alegerea aleatorie a muchiilor asupra cărora se pun întrebări, atribuirea fiecărei muchii din graf o probabilitate ca ea să facă parte din arborele parțial și să alegem mereu să punem întrebări pe cea cu probabilitatea cea mai mare, reducerea grafului la un arbore și după de aplicat metoda eficientă din primul subtask, o combinație dintre a alege muchii random și de a pune întrebări pe poduri când acestea apar etc.

O primă observație care se poate face este că în orice tăietură a grafului există cel puțin o muchie din arborele parțial ascuns.

O a doua observație este următoarea: dacă găsim o tăietură în graf în care fiecare muchie dintr-o parte în cealaltă care aparțin și arborelui ascuns sunt orientate în aceeași direcție, atunci putem fi siguri că rădăcina nu se află într-o parte a tăieturii, și putem reduce graful la partea din tăietură în care se află rădăcina.

O soluție care obține în jur de 10 – 50 de puncte, depinzând de implementare, alege două noduri aleatorii între care se găsește tăietura minimă și se pun întrebări toate muchiile din tăietură asupra cărora încă nu s-au pus întrebări. După aceste întrebări, verificăm dacă există o tăietură a grafului în care toate muchiile sunt orientate într-o singură direcție cu DFS. Se repetă acest algoritm până se găsește rădăcina.

Soluția de 100 de puncte se folosește de un Gomory-Hu Tree pentru a alege la fiecare pas tăietura optimă din punct de vedere a numărului maxim de noduri rămase într-o parte a tăieturii, dacă se întâmplă ca aceasta să conțină muchii în

arbore orientate într-o singură direcție. De asemenea, se ignoră tăieturile în care deja s-au găsit muchii în arbore orientate în ambele părți.

Soluția comisiei conține mai mulți euristici, care s-ar putea să trebuiască să fie implementați de concurent în caz de termenul de indulgență nu este îndeajuns pentru a compensa. Două exemple sunt: atribuirea unei probabilități fiecărei muchii din tăietură de a fi în arborele ascuns și de a o alege mereu pe cea cu probabilitatea cea mai mare pentru a se pune întrebarea, oprirea algoritmului când se găsește un nod despre care putem fi siguri că este rădăcina (s-au pus întrebări pe toate muchiile incidente și nu s-a găsit sursa în arbore), eliminarea muchiilor redundante (despre care putem fi siguri că nu se află în arborele parțial).