Utilizarea randomizării în probleme

Alexandru Luchianov

April 2022

1 Introducere

Algoritmii randomizați sunt o unealtă puternică în rezolvarea de probleme. Un exemplu arhicunoscut este algoritmul de sortare Quicksort. [1]

Alți algoritmi a căror performanță se bazează pe argumente probabilistice sunt Hashurile[2] si Treapurile[3]. Scopul acestui articol este de a prezenta și alte utilizări ale randomizării în programarea competitivă în afara celor menționate mai sus.

2 Consul

Enunț. Există un șir ascuns cu N elemente, dorim să găsim un element care apare de strict mai mult de $\frac{N}{3}$ ori. Avem la dispoziție 2 tipuri de operații. Putem să întrebăm de valoarea elementului de la o anumită poziție sau să întrebăm de câte ori apare o anumită valoare în șirul ascuns.

Problema originală se poate găsi la linkul: `oj.uz/problem/view/info1cup19_consul`

Solutie. O posibilă soluție ar consta în a întreba de valoarea fiecărei poziții și apoi a verifica dacă există vreun element care apare de cel puțin $\frac{N}{3}$ ori. Această soluție ne-ar garanta faptul că răspunsul nostru este corect, însă pentru a obține această garanție trebuie să folosim un număr extraordinar de mare de operații. Am putea încerca să sacrificăm din certitudinea asupra corectitudinii răspunsului nostru pentru a putea folosi mai puține operații.

O posibilă strategie ar fi să alegem un element arbitrar, să întrebăm de valoarea lui, iar apoi despre frecvența acestei valori. O astfel de strategie are o șansă de $\frac{1}{3}$ de a identifica un element care apare de cel puțin $\frac{N}{3}$ ori. Această șansă poate părea mică, însă putem pur și simplu să repetăm procedeul de multiple ori. Deoarece șansa ca o încercare să eșueze este de $\frac{2}{3}$, putem spune că șansa ca K astfel de încercări să eșueze este de $(\frac{2}{3})^K$. Dacă am face 10 astfel

[1] `ro.wikipedia.org/wiki/Quicksort`
[2] `infoarena.ro/problema/hashuri`
[3] `infoarena.ro/treapuri`

de încercări atunci șansa ca toate să eșueze ar fi de 0.02%. Acesta este un risc acceptabil.

3 Ghd

Enunț. Se dă un șir A de N ($N \leq 10^6$) elemente. Toate elementele sunt mai mici ca 10^{12}. Se definește *ghd*-ul șirului ca cel mai mare număr care divide cel puțin jumătate din elementele din șir. Să se determine *ghd*-ul acestui șir.

Problema originală se poate găsi la linkul: `codeforces.com/contest/364/problem/D`

Soluție. Un lucru care atrage atenția este faptul că *ghd*-ul trebuie să dividă cel puțin jumătate din elementele din șir. Asta înseamnă că dacă am alege un element arbitrar, A_k, din șir, atunci *ghd*-ul va fi unul din divizorii săi cu probabilitate de $\frac{1}{2}$. Dacă am încerca pe rând x numere alese arbitrar, atunci șansa ca *ghd-ul* să nu se găsească printre divizorii acestor numere ar fi de 2^{-x}. De exemplu, dacă am testa toți divizorii a doar 10 numere arbitrare alese din șir, șansa să nu găsim *ghd*-ul ar fi de doar $2^{-10} = \frac{1}{1024}$. Astfel șansa de a nu găsi $ghd - ul$ după ce am testat 10 numere este de cel mult 0.1%, adică șansa să îl găsim este de cel puțin 99.9%.

O limită superioară folosită în practică a numărului de divizori pentru numerele N ($N \leq 10^{18}$) este de $\mathcal{O}(N^{\frac{1}{3}})$. Astfel, putem verifica fiecare divizor al unui număr arbitrar din șir ($A_k \leq 10^{12}$) în mod brut în $\mathcal{O}(10^4 N)$. Cu această metodă, chiar și verificarea tuturor divizorilor unui singur număr ar dura mult prea mult.

Putem observa faptul că multe din numerele din șir sunt divizibile cu exact aceiași divizori ai lui A_k, astfel că ele sunt practic același număr din punctul nostru de vedere. Am vrea să găsim o transformare a șirului care să reflecte acest lucru.

Am putea transforma șirul A într-un șir echivalent $B_i = gcd(A_i, A_K)$[4]. Pe acest șir nou B, se menține proprietatea necesară: Dacă avem $d \mid A_K$, atunci știm că $d \mid A_i$ este echivalent cu $d \mid B_i$. Prin această transformare am reușit să reducem numărul de elemente distincte din șir la doar $\mathcal{O}(A_k^{\frac{1}{3}})$. Calcularea șirului B_i se poate face în $\mathcal{O}(N \log)$, iar apoi verificarea fiecărui divizor se poate face în doar $\mathcal{O}(A_k^{\frac{2}{3}})$

In concluzie, putem găsi *ghd*-ul șirului cu o probabilitate de succes de 99.9% in $\mathcal{O}(10 * (N \log + A_k^{\frac{2}{3}}))$. Această complexitate este extrem de aproape de limita de timp, însă putem aplica diverse optimizări pentru a ne încadra mai lejer:

[4] gcd(a, b) = cel mai mare divizor comun al lui a și b

- Când calculăm câte elemente din șirul B sunt divizibile cu un anumit număr d, testăm doar elementele mai mari decât d.
- Când testăm divizorii unui număr, îi testăm numai pe cei mai mari decât *ghd*-ul curent.
- Optimizăm citirea șirului.

După aplicarea optimizărilor de mai sus, ar trebui ca programul să se încadreze în mod lejer în limita de timp.

4 Aplicații ale randomizării în geometrie

Enunț. Se dau N ($N \leq 2*10^5$) puncte în plan. Intr-o operație putem șterge 3 puncte care formează un triughi cu aria mai mare ca 0. Care este numărul minim de puncte cu care putem rămâne dacă aplicăm operația descrisă de multiple ori?

Problema originală a fost dată la Grand Prix of Beijing în cadrul celui de-al XXI-lea Open Cup.

Soluție. Deoarece mereu eliminăm câte 3 puncte este clar faptul că numărul rămas de puncte o să fie mai mare sau egal cu $N \mod 3$. Un alt lucru ușor de observat este faptul că dacă există o linie care conține K puncte, atunci noi vom putea elimina maxim $(N - K) * 3$ puncte deoarece la fiecare operație trebuie să selectăm cel puțin un punct din afara dreptei. Aceste observații ne oferă o limită inferioară cu privire la răspuns. Se poate intui sau demonstra ușor faptul că această limită inferioară se poate și atinge, însă, deoarece scopul articolului este partea de randomizare, vom lăsa acest lucru ca un exercițiu pentru cititor.

De aici, problema se poate reduce ușor la numărul maxim de puncte de pe o dreaptă. Aceasta problemă, în mod normal, este extrem de complicată, însă în cazul nostru, noi suntem interesați de numărul exact de puncte de pe această dreaptă doar dacă conține mai mult de $\frac{2N}{3}$ puncte, altfel numărul exact de puncte este irelevant. Acest detaliu poate părea nesemnificativ dintr-un punct de vedere determinist, însă putem să ne folosim de el prin puterea randomizării. Putem alege din set 2 puncte arbitrare, să calculăm dreapta dintre ele, iar apoi să evaluăm câte puncte sunt pe această dreaptă. La fiecare pas, avem o șansă de $\frac{4N}{9}$ să nimerim dreapta cu număr maxim de puncte. Aceasta este o șansă de aproape 45%. Dacă am repeta această procedură de 10 ori am obține rezultatul căutat cu o certitudine de 99.99%. Această garanție e mai mult decât suficientă.

Alte aplicații posibile ale randomizării în geometrie sunt găsirea celor mai apropiate 2 puncte în plan în timp așteptat liniar [5], găsirea cercului minim care include un set de puncte în timp așteptat liniar [6] sau verificare dacă o

[5] Mai multe despre acest algoritm se pot găsi în lucrarea "A Reliable Randomized Algorithm for the Closest-Pair Problem" aparținând lui Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen si Martti Penttonen

[6] `en.wikipedia.org/wiki/Smallest-circle_problem`

intersecție de semiplane este goală în timp așteptat liniar. [7]. Acești algoritmi sunt întâlniți extrem de rar în programarea competitivă și au de cele mai multe ori alternative deterministe care, chiar dacă au o complexitate teoretică mai proastă, pot merge suficient de rapid.

5 Maximul într-un șir

Enunț. Fie un șir ascuns de N elemente cu valori întregi cuprinse între 1 și V. In cadrul unei operații putem întreba dacă un anumit element din șirul ascuns este mai mic decât o anumită valoare. Se dorește găsirea maximului din șir într-un număr cât mai mic de operații.

Soluție. Putem găsi valoarea exactă a unui element în aproximativ $\log V$ întrebări. După ce găsim valoarea fiecărui element putem identifica ușor și maximul. Aceasta este soluția standard care folosește $N \log V$ întrebări. Insă, se poate mai bine de atât, un posibil indiciu este faptul că generăm extrem de multă informație irelevantă precum valorile exacte ale fiecărui element.

O posibilă optimizare este ca, înainte de a căuta valoarea exactă a unui element, să utilizăm o întrebare pentru a determina dacă acest element este mai mare decât maximul găsit anterior. In cazul unui șir sortat, această optimizare este contraproductivă deoarece am irosi câte o întrebare la fiecare element. Pentru a scăpa de această vulnerabilitate putem încerca să parcurgem numerele într-un mod arbitrar. Incă există o șansă ca noi să parcurgem șirul în ordine sortată, însă aceasta șansă este extrem de mică și nesemnificativă.

Pentru a estima numărul de întrebări utilizat de această metodă, trebuie mai întâi să estimăm de cate ori o să întâlnim un element pe care este necesar să îl aflăm în mod explicit. Acest lucru este echivalent cu numărul de maxime pe prefix distincte în medie într-un șir arbitrar și se poate calcula in următorul fel:

Dacă notăm cu $E\left[changes\right]$ numărul de maxime pe prefix în medie și cu $p(i)$ șansa ca al i-lea element să fie maximul pe prefixul de lungime i, este ușor să ajungem la următoarea formulă:

$E\left[changes\right] = \sum_{i=1}^{N} p(i) = \sum_{i=1}^{N} \frac{1}{i} = H_n = \ln N$

La o primă vedere această formulă poate părea intimidantă, însă adevărul este că poate să fie interpretată într-un mod mult mai simplu prin cuvinte. Numărul așteptat de schimbări de maxim pe prefix este rescris ca suma probabilităților ca fiecare element să fie maxim pe prefixul său. Șansa ca al i-lea element să fie maxim pe prefixul de lungime i este chiar $\frac{1}{i}$ deoarece toate elementele de pe acest prefix au șanse egale să fie maximul. Ultimul pas constă în recunoașterea sumei $\frac{1}{1} + \frac{1}{2} \ldots \frac{1}{N}$ ca fiind al N-lea număr armonic. Acest număr poate să fie aproximat drept $\ln N$.

[7] Mai multe detalii despre acest subiect se pot găsi aici `petr-mitrichev.blogspot.com/2016/07/a-half-plane-week.html`

Numărul total de operații al metodei prezentate mai sus este estimat ca fiind aproximativ $N + \log V \ln N$. Această metodă de a estima numărul de maxime pe prefixe poate să fie utilizată și în alte probleme, precum următoarea.

6 Rucsac Random

Enunț Se dă un șir de N ($N \leq 10^4$) numere cuprinse între 1 si V ($V \leq 100$). La fiecare pas se pot alege 2 dintre acestea, să le eliminam din șir și să adăugăm înapoi în șir diferența lor în modul. Care este numărul minim cu care putem rămâne la finalul acestui procedeu?

Problema originală se poate găsi la `pbinfo.ro/probleme/3105/bete2`.

Soluție Soluția originală a problemei se bazează pe o serie de observații matematice, însă există o soluție mai ușoară bazată pe următorul algoritm randomizat.

Este ușor de demonstrat faptul că procedeul de mai sus este echivalent cu a atribui numerelor semnul + sau − cu condiția ca suma să nu fie negativă și apoi a le însuma. Această problemă seamănă extrem de mult cu problema clasică a rucsacului. [8]

Ca și în problema rucsacului am putea calcula rezultatul folosind programare dinamică. Mai întâi o să notăm cu v_i al i-lea element din șir. Apoi, o să definim $dp(i, j)$ ca 1 dacă există cel puțin o metodă de a alege semnele pentru primele i elemente astfel încât să obținem suma j. Dacă nu se poate obține această sumă, atunci o să îl definim ca 0. Aceasta formulare are o recurență extrem de simplă:

$dp(i, j) = dp(i - 1, j - v_i) \vee dp(i - 1, j + v_i)$

Operatorul $\vee$ reprezintă operația logică *sau*. Calcularea acestei dinamici ar dura $O(N^2V)$, ceea ce în cazul nostru este mult prea mult, așa că trebuie să apelăm la câteva trucuri.

Primul truc la care putem apela se bazează faptul că $dp(i, j)$ poate lua doar valorile 0 și 1. Putem reține în memoria calculatorului fiecare linie $dp(i)$ drept un șir de biți și să lucrăm direct cu aceste șiruri de biți. Astfel recurența se transformă în :

$dp(i) = dp(i - 1) \ll v_i \vee dp(i - 1) \gg v_i$

Operatorii $\ll$ si $\gg$ reprezintă operația de shiftare pe biți. Este ușor de văzut de ce aceste 2 recurențe sunt echivalente. Un avantaj al acestei reprezentări este faptul că putem lucra cu tot șirul în același timp și astfel putem face mai multe lucruri în paralel. Pentru a implementa această idee am putea reține fiecare linie drept o serie de variabile int fiecare de cate 32 de biți. Astfel, am putea să lucram cu câte 32 de biți în același timp și să obținem o îmbunătățire a timpului de execuție cu un factor de $\frac{1}{32}$.

Desigur, este inutil să reinventăm acest lucru deoarece există deja o librărie în C++ numită bitset.[9]

[8] infoarena.ro/problema/rucsac

[9] en.cppreference.com/w/cpp/utility/bitset

O altă optimizare posibilă, de data aceasta bazată pe randomizare, ar fi să permutăm aleator șirul de început iar apoi să impunem restricția ca suma pe fiecare prefix din soluția optimă să se încadreze într-un anumit interval ales de noi $[-K, K]$. Astfel, numărul din stări din dinamica poate să fie redus la $N * K$. Acest K îl putem alege cât de mare posibil cât să ne încadrăm în limita de timp. Se poate găsi în mod matematic un K cu o probabilitate mare de succes folosind limite Chernoff, se poate demonstra ca acest K este de ordinul lui $V\sqrt{N}$. Lăsăm această demonstrație ca un exercițiu pentru cititor.

7 Mici detalii de limbaj

Majoritatea limbajelor de programare conțin metode de a genera numere pseudo-random. De exemplu, in C++ există funcția numită *rand*()[10] care ne permite să generăm numere pseudo-random. In ciuda popularității acestei metode, aceasta are dezavantaje clare atunci când este comparată cu funcțiile disponibile în librăria random.[11]

Un prim punct slab este faptul că funcția *rand*() generează un număr întreg random între 0 și RAND_MAX. Acest lucru pare o mică inconvenientă la prima vedere, însă RAND_MAX este garantat să fie minim 32767. Aceasta este o valoare neașteptat de mică și acest lucru poate cauza probleme în cazul în care nu suntem conștienți de această limită. Desigur, dacă suntem atenți, acest lucru este doar o inconvenientă, însă alternativele din librăria random ne-ar permite să generăm numere în orice interval dorim.

Un alt avantaj al librăriei random peste varianta standard cu *rand*() sunt numeroasele distribuții posibile. Un exemplu de distribuție folositoare este distribuția uniformă pe numere reale sau distribuția geometrică.

Un lucru pe care îl au totuși în comun funcția *rand*() și librăria *random* este faptul că ambele sunt generatoare de numere pseudo-random. Un număr pseudo-random nu este un număr cu adevărat random deoarece așa ceva ar fi extrem de dificil de generat, posibil chiar imposibil, ci un număr generat folosind un procedeu matematic complex care pare suficient de "random". Pentru a determina dacă un astfel de procedeu matematic îndeplinește această condiție, acesta este supus unei serii de teste. Din acest lucru este ușor de dedus faptul că o parte din generatoarele de numere pseudo-random sunt considerate mai slabe, functia rand() fiind una dintre acestea. Acest fapt este aproape irelevant în contextul programării competitive, ci este doar o dovadă în plus a vulnerabilității funcției *rand*().

Un ultim lucru care trebuie luat în considerare este faptul că aceste funcții de random trebuie inițializate cu un anumit număr relativ random. De cele mai multe ori este suficient să inițializăm folosind timpul curent sau să folosim adresa în memorie a unei variabile nou alocate. Mai multe detalii se pot găsi la linkul codeforces.com/blog/entry/61587.

[10] cplusplus.com/reference/cstdlib/rand/
[11] www.cplusplus.com/reference/random/

8 Probleme suplimentare

- infoarena.ro/problema/partition
- codeforces.com/problemset/problem/739/E
- codeforces.com/contest/1101/problem/F
- atcoder.jp/contests/abc234/tasks/abc234_h