# Concursul Național de Programare "STELELE INFORMATICII"

*Mihai Stroe*

**Vă prezentăm în continuare soluțiile problemelor propuse spre rezolvare la Concursul național de programare "Stelele informaticii". Aceste soluții au fost realizate de redacția GInfo pe baza soluțiilor oficiale prezentate de către autorii problemelor.**

## Clasele a IX-a și a X-a

### P080301: Cuvinte

Problema se rezolvă folosind metoda *programării dinamice*.

Pornind de la sfârșitul șirului de numere, vom construi un tablou $A$, în care $A_i$ reprezintă numărul de subșiruri strict crescătoare de lungime maximă care încep cu elementul din poziția $i$. Acest tablou se construiește folosind rezolvarea problemei subșirului strict crescător de lungime maximă, deci se va construi în paralel un tablou $L$, unde $L_i$ reprezintă lungimea celui mai lung subșir strict crescător care începe pe poziția $i$.

Fie $MAX$ valoarea maximă din $L$.

Suma elementelor din $A$, corespunzătoare elementelor maxime din $L$, reprezintă numărul total de subșiruri strict crescătoare de lungime maximă. Ne interesează al $K$-lea astfel de subșir.

Fie $i_1$ poziția primei apariții a lui $MAX$ în $L$, $i_2$ poziția celei de-a doua apariții ș.a.m.d. Dacă $A_{i_1} \geq K$, atunci subșirul căutat este al $K$-lea subșir strict crescător maximal care începe pe poziția $i_1$. Dacă $A_{i_1} < K \leq A_{i_1} + A_{i_2}$, atunci subșirul căutat începe cu poziția $i_2$ etc.

După ce a fost găsit primul element al subșirului căutat, se determină al doilea element, apoi al treilea etc. Metoda este similară. Se impune următoarea observație: dacă, de exemplu, subșirul începe cu poziția $i_3$, se caută al $(K - A_{i_1} + A_{i_2})$ - lea subșir crescător maximal care începe după poziția $i_3$, are lungimea $MAX$ - 1, iar primul element al său este mai mare decât elementul de pe poziția $i_3$. Acest subșir se caută folosind același procedeu.

#### Analiza complexității

Operațiile de citire a datelor de intrare și scriere a rezultatelor au ordinul de complexitate $O(N)$.

Calculul elementelor tablourilor $A$ și $L$ are ordinul de complexitate $O(N^2)$; acesta este ordinul de complexitate al rezolvării clasice a problemei subșirului crescător de lungime maximă. Datorită limitelor din enunț și simplității implementării, acest algoritm este cea mai bună alegere în timp de concurs.

Recomandăm cititorului să încerce adaptarea algoritmului de complexitate $O(N \cdot \log N)$ pentru această problemă.

Determinarea celui de-al $K$-lea subșir pe baza tablourilor $A$ și $L$ are ordinul de complexitate $O(N)$.

În final, ordinul de complexitate al rezolvării pentru această problemă este $O(N^2)$.

### P080302: Timp

Pentru a rezolva această problemă pornim de la următoarea observație: la un moment dat, în clepsidră se află cel mult două cantități nenule de nisip, $X$ și $Y$.

Perechea $(X, Y)$ poate fi tratată similar cu perechea $(Y, X)$. Considerăm $X \leq Y$.

Din $X$ și $Y$ se va obține fie perechea $(X/2, Y + X/2)$, fie $(Y/2, X+Y/2)$, în funcție de sensul în care rotim clepsidra. Al doilea element este mai mare sau egal cu primul.

Considerând invers, din ce pereche se obține perechea $(A, B)$ cu $B \geq A$?

Simplu: din $(X, Y)$ se obține $(X/2, Y + X/2)$ sau $(Y/2, X + Y/2)$, deci elementul mai mic este unul din elementele din $(X, Y)$ împărțit la 2.

Atunci un element din perechea care a condus la $(A, B)$ este $2 \cdot A$ (cel mai mic element din $(A, B)$ înmulțit cu 2). Al doilea element este $A + B - 2 \cdot A$, adică $B - A$.

Această observație conduce la următorul algoritm:
- se pleacă de la configurația finală a clepsidrei;
- se efectuează mutările inverse, ținând cont de regula "$(A, B)$ cu $A \leq B$ provine din $(2 \cdot A, B - A)$"; rotirea spre stânga, respectiv spre dreapta sunt date de poziția elementului mai mic din pereche;
- se reconstituie șirul de mutări pornind de la configurația inițială și se afișează.

**Analiza complexității**

Ordinul de complexitate al operației de citire a datelor de intrare este $O(1)$.

Se observă că după primul pas, cele două numere vor fi obligatoriu pare, după al doilea pas vor fi divizibile cu 4, iar după al $K$-lea pas, cu $2^K$. Deoarece suma lor este $N$, numărul de pași este $O(\log N)$.

Ordinul de complexitate al operației de scriere a rezultatelor este $O(\log N)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare pentru această problemă este $O(\log N)$.

## P080303: Vecini

Pentru a rezolva această problemă, la început, vom genera primele 8 linii:

```
1
1 1
1 0 1
1 1 1 1
1 0 0 0 1
1 1 0 0 1 1
1 0 1 0 1 0 1
1 1 1 1 1 1 1 1
```

Completând toate liniile cu zerouri până la cea mai apropiată putere a lui 2, se obține următorul tabel:

```
1
1 1
1 0 1 0
1 1 1 1
1 0 0 0 1 0 0 0
1 1 0 0 1 1 0 0
1 0 1 0 1 0 1 0
1 1 1 1 1 1 1 1
```

Se observă că jumătatea a doua a fiecărei linii este identică cu prima jumătate. Mai mult, elementele din prima jumătate a unui rând cu $2^K$ elemente (după completare) sunt identice cu elementele aflate cu $2^{K-1}$ rânduri mai sus (dacă am completa și rândurile respective cu suficiente zerouri).

Pentru a valida aceste observații se poate demonstra prin inducție că observațiile sunt valabile în tot tabelul; concurenții pot alege o altă variantă de validare generând mai multe linii din tabel prin metoda forței brute și testând această ipoteză.

În acest moment, ideea de rezolvare devine destul de clară:
- pentru a genera linia $N$ se generează și se dublează "prima jumătate". Am pus "prima jumătate" între ghilimele deoarece este vorba despre jumătatea după completarea până la o putere a lui 2;
- pentru a genera "prima jumătate" se apelează recursiv aceeași metodă, având ca parametru un nou $N$, care se obține scăzând din vechiul $N$ cea mai mare putere a lui 2 mai mică decât el.

Există și alte variante de rezolvare eficiente, toate exploatând proprietățile descrise mai sus.

**Analiza complexității**

Aproximăm superior pe $N$ cu cea mai mică putere a lui 2 mai mare sau egală cu $N$. Fie aceasta $P$.

Conceptual, sunt generate $O(P)$ elemente ($P$ pe linia $N$, $P / 2$ cu $P / 2$ linii mai sus etc., suma fiind aproximativ $2 \cdot P$). Practic, se poate folosi un singur vector.

Pentru fiecare element se fac calcule în $O(1)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare pentru această problemă este $O(N)$.

## P080304: Editor

Se citesc caracterele unul câte unul și se consideră o stivă, inițial goală. De fiecare dată când se întâlnește o paranteză (deschisă sau închisă, dreaptă sau rotundă), aceasta se introduce în stivă. Dacă se întâlnește caracterul "`*`", atunci se scoate caracterul din vârful stivei (dacă există un astfel de caracter). În final (la întâlnirea caracterului "`E`"), stiva va conține șirul afișat pe ecran.

Pentru a verifica dacă un șir este parantezat corect, acesta se parcurge de la stânga la dreapta și se consideră o a doua stivă, inițial vidă. Dacă se întâlnește o paranteză deschisă dreaptă sau rotundă, aceasta se introduce în vârful stivei. Dacă se întâlnește o paranteză închisă și caracterul din vârful stivei nu este o paranteză deschisă de același tip cu paranteza închisă (dreaptă sau rotundă), atunci se semnalează eroare. Altfel, paranteza deschisă din vârful stivei se elimină și se trece la următorul caracter. Dacă nu s-a semnalat eroare în timpul parcurgerii șirului și la final stiva este vidă, atunci șirul este parantezat corect.

**Analiza complexității**

Notăm cu $N$ numărul total de caractere și cu $T$ numărul total de linii prezente în fișierul de intrare. În aceste condiții, operația de citire a datelor are un ordin de complexitate

$O(N)$, iar operația de scriere a rezultatelor are ordinul de complexitate $O(T)$.

Fiecare din cei doi pași ai rezolvării are ordinul de complexitate $O(N)$, deoarece pentru fiecare caracter din șir se execută o secvență de operații de complexitate $O(1)$.

În concluzie, ordinul total de complexitate al algoritmului de rezolvare a acestei probleme este $O(N)$.

## P080305: Numere

Practic, fiecare mutare duce la schimbarea parității vecinilor poziției în care s-a efectuat mutarea, respectiv la schimbarea / menținerea parității poziției în care s-a efectuat mutarea, în funcție de numărul de vecini.

În continuare vom lucra, pentru fiecare poziție, cu restul valorii corespunzătoare la împărțirea cu 2. Problema cere să aducem toate aceste resturi fie la valoarea `0`, fie la valoarea 1 și să alegem varianta cu număr minim de mutări. Vom trata primul caz, cel de-al doilea rezolvându-se în mod similar.

Se observa că $M$ și $N$ sunt mai mici sau egale cu 12. De asemenea, o a doua mutare într-o anumită poziție anulează efectul primei mutări în poziția respectivă, ceea ce înseamnă că în fiecare poziție se va face cel mult o mutare.

Observațiile de mai sus conduc la următorul algoritm de rezolvare:

- Se generează toate cele maxim $2^N$ modalități de a efectua mutări în prima linie (se mută / nu se mută în prima poziție, a doua etc.). Am scris maxim $2^N$ modalități deoarece în anumite poziții nu se pot efectua mutări.
- Pentru fiecare astfel de modalitate, se examinează, pe rând, necesitatea de a muta în pozițiile din rândul 2, 3, …, $M$. Dacă suntem în poziția $(i, j)$ și elementul de pe poziția $(i - 1, j)$ nu este adus la valoarea `0`, este obligatoriu să mutăm în $(i, j)$. În caz contrar este interzis să mutăm în $(i, j)$, deoarece nu putem afecta elementul $(i - 1, j)$ cu alte mutări în linia $i$ sau mai jos.

Sunt efectuate mutările obligatorii, ceea ce conduce la modificarea elementelor din matrice. Dacă o mutare obligatorie este împiedicată de faptul că în poziția respectivă nu se pot efectua mutări, înseamnă că varianta curentă (dată de mutările de pe prima linie) nu conduce la soluție.

După examinarea obligativității mutărilor în linia $M$ se verifică dacă toate elementele din linia $M$ au valoarea `0`. Dacă da, varianta curentă a condus la soluție, deci se verifică dacă aceasta este mai bună decât soluția optimă obținută anterior.

### Analiza complexității

Ordinul de complexitate al operației de citire a datelor de intrare este $O(M \cdot N)$.

Sunt maxim $2^N$ variante de a muta pe prima linie; pentru fiecare astfel de variantă, verificarea obligativității și efectuarea mutărilor se realizează în $O(M \cdot N)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare pentru această problemă este $O(M \cdot N)$.

## P080306: Oo

Se construiesc trei tablouri unidimensionale (conceptual; practic, se va folosi unul singur):

- $A_1$ – elementul de pe poziția $i$ reprezintă numărul maxim de ouă adunate până în sectorul $i$, dacă se adună ouăle din primele două sectoare, luate împreună;
- $A_2$ – elementul de pe poziția $i$ reprezintă numărul maxim de ouă adunate până în sectorul $i$, dacă se adună ouăle din sectorul al doilea, împreună cu cele din al treilea;
- $A_3$ – elementul de pe poziția $i$ reprezintă numărul maxim de ouă adunate până în sectorul $i$, dacă se adună ouăle din primul și ultimul sector, luate împreună.

Evident, în primul caz nu se pot lua ouăle din ultimul sector etc.

Valorile din aceste tablouri se pot calcula pe baza unor recurențe simple, ceea ce conduce la un algoritm liniar.

În final se selectează maximul dintre aceste valori, determinate corect.

### Analiza complexității

Ordinul de complexitate al operației de citire a datelor de intrare este $O(N)$, iar cel al operației de scriere a rezultatelor este $O(1)$.

Calculul valorilor din tablouri are ordinul de complexitate $O(N)$.

În concluzie, ordinul total de complexitate al algoritmului de rezolvare a acestei probleme este $O(N)$.

# Clasele a XI-a și a XII-a

## P080307: Arbori

Problema se rezolvă prin metoda *programării dinamice*.

Construim o matrice $A$ cu $M$ linii și $K + 1$ coloane (numerotate de la `0` la $K$), unde $A_{i,j}$ reprezintă efortul total minim necesar pentru a obține un subarbore pornind din nodul $i$ al primului arbore, izomorf cu un subarbore pornind din nodul corespunzător al celui de-al doilea arbore, care să conțină exact $j$ frunze.

Efortul este dat de formula $1000 \cdot nrCrengiTaiate + nrFloriRupte$. În acest mod se elimină `if`-uri complexe de genul: "dacă numărul de crengi este mai mic decât optimul până la momentul respectiv, sau este egal, dar numărul de flori este mai mic".

O parte din elemente sunt egale cu $+\infty$ (o valoare maximă fixată); pentru acestea nu se pot obține subarbori izomorfi cu $j$ noduri.

Pentru a obține un subarbore cu $j$ noduri, pornind din nodul $i$, avem următoarele opțiuni:

- lăsam $f$ flori în nodul $i$ (trebuie să putem lăsa $f$ flori și în nodul izomorf cu $i$ din celălalt subarbore); $f$ este contorul într-un ciclu `for`;
- în continuare, fie nu păstrăm nici o creangă din $i$ (deci tăiem între `0` și 4 crengi, deoarece vom tăia numai crengi care există, din primul și din cel de-al doilea arbore), fie păstrăm numai creanga stângă (dacă există în ambii arbori), fie

numai pe cea dreaptă (dacă există), fie pe amândouă (dacă există în ambii arbori);
- pentru ultimul caz, din cele $j - f$ flori, o parte ($st$) sunt în stânga, iar $j - f - st$ în dreapta.

Nu am descris toate condițiile care se pun pentru a nu complica prezentarea.

Elementele matricei se calculează de la frunze spre rădăcină. Reconstituirea soluției se poate face fie pe baza valorilor din matrice, fie stocând informații suplimentare.

Se pot face numeroase optimizări. De exemplu, pentru fiecare nod $i$ se poate determina numărul maxim de flori care poate fi obținut într-un subarbore cu rădăcina $i$, astfel încât să existe subarborele izomorf corespunzător în cel de-al doilea arbore. Aceste valori influențează limitele `for`-urilor care apar la calculul elementelor.

Implementarea și tratarea tuturor cazurilor necesită o atenție mărită; totuși, ideea de rezolvare nu este foarte complicată, deci problema se poate rezolva în mai puțin de 90 de minute, incluzând testarea și depanarea.

### Analiza complexității

Operația de citire a datelor de intrare are ordinul de complexitate $O(M + N)$.

Calculul unuia dintre cele maxim $M \cdot K$ elemente ale matricei are ordinul de complexitate $O(11 \cdot K) = O(K)$. Aici, 11 vine de la numărul de flori pe care îl lăsăm în nodul respectiv, iar $K$ de la posibilitatea de a păstra atât subarborele stâng, cât și subarborele drept, ceea ce duce la împărțirea florilor între cei doi subarbori.

În concluzie, ordinul de complexitate al algoritmului de rezolvare pentru această problemă este $O(M \cdot K^2)$.

În practică, numărul fix de operații de la fiecare pas este destul de mare (apar foarte multe condiții), dar nu toți cei $M \cdot K^2$ pași amintiți sunt efectuați. De exemplu, pentru nodurile care au un singur descendent se execută $O(11)$ operații în loc de $O(11 \cdot K)$. În orice caz, algoritmul se încadrează în timp.

## P080308: Sum

Problema admite o rezolvare de complexitate $O(N)$, dar și alte rezolvări de complexitate $O(N \cdot \log N)$ care merită menționate, deoarece pot fi folosite în alte probleme.

Prezentăm mai întâi rezolvarea liniară.

Construim un tablou $P$ cu semnificația că $P_i$ = *poziția de sfârșit a celei mai scurte secvențe, care începe în $i$ și are suma elementelor strict pozitivă.*

Construcția acestui tablou se face de la dreapta spre stânga, folosind elementele tocmai calculate. Secvența $(i, P_i)$ se poate numi atomică.

Se observă că dacă avem o secvență curentă care se termină în poziția $i$ și vrem să mărim această secvență spre dreapta (cu scopul de a crește suma elementelor) atunci trebuie să sărim direct în $P_i$, pentru că altfel i-am scădea suma (deoarece am adăuga o cantitate negativă). Se încearcă într-un fel împărțirea șirului inițial în secvențe strict pozitive, însă această împărțire nu se face numai asupra șirului inițial, ci a oricărei secvențe.

Deși aparent construcția acestui tablou are ordinul de complexitate $O(N^2)$, o analiză amortizată ne arată că este vorba de $O(N)$.

Cu ajutorul tabloului $P$, ne "plimbăm" cu doi *pointer*-i (indici) în șirul inițial (limita stângă și limita dreaptă a secvenței căutate, inițial plecând din stânga șirului), având grijă ca diferența dintre cei doi să fie între $L$ și $U$. În căutarea secvenței *pointer*-ii sar numai peste secvențe atomice (din $i$ se sare în $P_i$). Fiecare secvență căutată se compară cu maximul până în acel moment. Astfel și această parte de căutare are ordinul de complexitate $O(N)$.

Recomandăm cititorului să analizeze comportamentul algoritmului pe un exemplu.

Vom prezenta în continuare două variante de rezolvare care au ordinul de complexitate $O(N \cdot \log N)$.

### Varianta cu heap-uri

Se folosește un *max-heap* cu $U - L + 1$ elemente. La fiecare pas $i$, în acest *heap* se află sumele secvențelor de elemente din șir, cu lungimea cuprinsă între $L$ și $U$, care se termină pe poziția $i$.

De fapt, în *heap* nu se memorează efectiv sumele, ci pozițiile de început ale acestor secvențe, astfel încât sumele să se afle în relația impusă de *max-heap*.

Avansarea la pasul $i + 1$ are următorul efect:
- toate secvențele din *heap* devin mai lungi cu un element. Aceasta nu modifică relația de *max-heap* dintre ele, pentru că la fiecare se adaugă aceeași cantitate;
- cea mai lungă secvență va avea lungimea $U + 1$ (dacă $i > U$), deci va trebui scoasă din *heap*;
- apare o nouă secvență de lungime $L$, care va trebui introdusă în *heap*.

Suma unei secvențe (cunoscând pozițiile de început și sfârșit) poate fi calculată folosind vectorul sumelor parțiale. Dacă șirul dat este $A$, atunci definim $S_i = A_1 + A_2 + ... + A_i$, $S_0 = 0$. Atunci, suma elementelor dintre pozitiile $i$ și $j$ este $S_j - S_{i-1}$.

Pentru localizarea secvențelor care trebuie scoase din *heap* se păstrează o informație auxiliară. De asemenea, se observă că va fi scos un element aflat pe o poziție oarecare, nu neapărat din vârful *heap*-ului.

Evident, la fiecare pas maximul este comparat cu soluția optimă de până atunci, și eventual aceasta este actualizată.

Recomandăm implementarea acestei soluții, ca un exercițiu util pentru lucrul cu *heap*-uri.

La fiecare pas, se fac cel mult două operații cu *heap*-uri, care pot fi realizate cu complexitate *logaritmică*. Ordinul de complexitate al algoritmului este $O(N \cdot \log (U - L))$, adică $O(N \cdot \log N)$ pentru cazul cel mai defavorabil.

### Varianta bazată pe programare dinamică

Se calculează, pentru fiecare poziție $i$, cea mai mare sumă a unei secvențe de lungime cel mult $1, 2, 4, ... 2^K$ începând din

poziția respectivă. Fie această informație $MAX[K, i]$. Notăm cu $LUNG[K, i]$ lungimea acestei secvențe.

Inițial $MAX[0, i]$ este elementul de pe poziția $i$ din șir.

La trecerea de la pasul $K$ la pasul $K + 1$, secvența de sumă maximă, de lungime cel mult $2^{K+1}$, care începe de pe poziția $i$, este fie secvența de lungime cel mult $2^K$ (cu suma $MAX[K, i]$), fie o secvență mai lungă. Singura secvență candidată este cea formată din cele $2^K$ elemente începând de pe poziția $i$, la care alipim încă $LUNG[K, i + 2^K]$ elemente care dau cea mai mare sumă a unei secvențe imediat următoare, de lungime maxim $2^K$.

Calculul valorilor din $MAX$ și $LUNG$ pentru un $K$ fixat are deci ordinul de complexitate $O(N)$, ceea ce înseamnă că determinarea tuturor valorilor va avea ordinul de complexitate $O(N \cdot \log N)$.

Cum folosim aceste valori?

Se observă că o secvență de lungime cel puțin $L$ și cel mult $U$ începe cu o secvență de lungime $L$, la care se alipesc la dreapta cel mult $U - L$ elemente. Fie $P$ cea mai mare putere a lui 2 mai mică decât $U - L$.

Atunci, secvența de sumă maximă, de lungime cel puțin $U$ și cel mult $L$, care începe cu $L$ elemente din care ultimul se află pe poziția $i - 1$, se încadrează în unul din următoarele cazuri:

- are exact lungimea $L$;
- are lungimea cuprinsă între $L$ și $L + P$, caz în care se adaugă la cele $L$ elemente o secvență de lungime $LUNG[P, i]$;
- are lungimea mai mare de $L + P$. Fie $D = U - L$. Atunci secvența se termină pe una din ultimele $P$ poziții din cele $U$, deci se adaugă $D - P$ elemente la primele $i$, după care urmează încă $LUNG[P, i + D - P]$ elemente.

Se ia varianta cea mai bună din aceste cazuri. Calculele necesare au ordinul total de complexitate $O(N)$.

Se impun următoarele observații:

- la fel ca la varianta cu *heap*-uri, se va folosi vectorul sumelor parțiale;
- se poate renunța la valorile din $MAX$; se lucrează numai cu $LUNG$ și sumele parțiale;
- din matricea $LUNG$ se vor folosi numai ultimele două linii; linia $K + 1$ se construiește pe baza liniei $K$, după care se renunță la linia $K$;
- pentru a rezolva problema, indicele $K$ al pasului pentru calcularea valorilor din $LUNG$ merge până la $P$, deci sunt $\log (U - L)$ pași și nu $\log N$ pași (valorile care ar fi obținute la pașii următori nu ne interesează).

În concluzie, ordinul de complexitate al algoritmului este $O(N \cdot \log (U - L))$, adică $\boldsymbol{O(N \cdot \log N)}$ pentru cazul cel mai defavorabil.

## P080309: Taxi

Această problemă nu este dificilă; dificultatea ei constă în a nu pierde din vedere vreun caz.

Notăm cu $dx$ diferența pe $x$ între cele două taxiuri și cu $dy$ diferența pe $y$.

Avem următoarele cazuri:

- $dx = 0$, $dy = 0$ (taxiurile coincid): toți vor fi nehotărâți;
- $dx > dy$: sunt $B + 1$ oameni nehotărâți;
- $dx < dy$: sunt $A + 1$ oameni nehotărâți;
- $dx = dy$: aici este un caz special. Pe diagonala dintre taxiuri vor fi oameni nehotărâți, însă vor exista, de asemenea, și două dreptunghiuri în care toți oamenii sunt nehotărâți. Să presupunem că taxiurile sunt colțurile unui pătrat. Dreptunghiurile (cu oameni nehotărâți) vor avea un colț în colțurile neocupate (de taxiuri) ale pătratului, iar altul în colțul orașului.

### Analiza complexității

Ordinul de complexitate al operațiilor de citire a datelor și scriere a rezultatelor este $O(1)$.

Ordinul de complexitate al rezolvării unui caz de test este $\boldsymbol{O(1)}$, deoarece identificarea cazurilor și calculele aferente necesită un număr de operații mic, limitat superior de o constantă.

## P080310: Expr

După citirea datelor, se sortează toate numerele din mulțimi. Pe parcursul rezolvării, un element va fi înlocuit prin indicele său din vectorul sortat; pentru aceasta se folosește căutarea binară.

Rezolvarea problemei se bazează pe evaluarea expresiilor cu ajutorul gramaticilor. Gramatica este structurată pe două niveluri: *expresie* și *operand*. Expresia este formată din mai mulți *operanzi* între care există operațiile specificate, iar operandul poate fi o mulțime sau altă expresie (dacă se întâlnește paranteza deschisă).

Pentru fiecare dintre cele două elemente ale gramaticii se va scrie o funcție recursivă. Aceasta returnează o mulțime, reprezentată printr-o structură de 1000 de octeți, suficienți pentru totalul de 8000 de numere distincte din expresie (din considerente de eficiență se poate prefera folosirea unei structuri de 1024 octeți). Această structură poate fi alcătuită din elemente de un octet (tipul *byte* în *Pascal*); se pot alege variante pe doi, respectiv patru octeți.

Pentru mai multe detalii vom analiza exemplul din enunț: `{1,2,3,4}%({1,2,3,4}*{}+{1,2}+{5,6}-{1})`

Inițial se apelează funcția *expresie*().

Expresia conține un operand, urmată eventual de un operator și de o altă expresie. Se va apela deci funcția *operand*(), se va citi un operator (dacă nu s-a terminat șirul), apoi se va apela funcția *expresie*(). Rezultatul apelării ei și rezultatul apelării funcției *operand*() vor servi ca operanzi operatorului respectiv.

Operandul inițial este dat de mulțimea `{1,2,3,4}`. Se citește apoi `%` și se apelează *expresie*().

Aceasta întâlnește un prim operand dat de paranteza deschisă, deci se va apela din nou *expresie*(), după care se va citi o paranteză închisă. Această nouă apelare va întoarce rezultatul `{1,2,3,4}*{}+{1,2}+{5,6}-{1}` (evident, prin noi apelări de *operand*() și *expresie*()). Fișierul se ter-

mină, deci se iese din funcțiile recursive și se calculează și celelalte rezultate.

Invităm cititorul să realizeze o implementare a acestei rezolvări. Este un exercițiu util de programare, testare și depanare (dacă este cazul).

Precizăm că, în cazul în care s-ar fi introdus priorități ale operatorilor, gramatica ar fi fost mai complicată.

### Analiza complexității

Fie $L$ numărul de caractere din expresie, $N$ numărul total de numere din mulțimile din expresie, $D$ numărul de numere distincte și $T$ numărul de operații.

Operația de citire a datelor de intrare are ordinul de complexitate $O(L)$.

Sortarea celor $N$ numere are ordinul de complexitate $O(N \cdot \log N)$.

Pe parcursul rezolvării, fiecare număr este transformat în indicele său din vectorul sortat, ceea ce duce la ordinul de complexitate cu $O(N \cdot \log D)$. Deoarece $D \le N$, acest termen poate fi ignorat.

Efectuarea fiecăreia din cele $T$ operații necesită $O(D)$ operații elementare. Pentru acestea se folosesc operațiile *OR*, *AND*, *XOR* pe octeții care compun structura.

În concluzie, ordinul de complexitate al algoritmului este $O(L + N \cdot \log N + T \cdot D)$. Relația pare complicată, dar ea ne spune că rezolvarea se va încadra în timp.

## P080311: Ciclu

Rezolvarea se bazează pe o legătura subtilă dintre ciclul minim și ciclul mediu minim într-un graf care admite arce de cost pozitiv, zero și negativ. Aceste costuri au întotdeauna același semn.

Facem o altă observație: dacă adăugăm (sau scădem) aceeași valoare $x$ la toate costurile muchiilor, costul ciclului mediu minim crește (respectiv scade) exact cu aceeași valoare $x$.

Se conturează deja o idee. Putem afla dacă un anumit număr $x$ este costul cerut astfel: scădem $x$ din costurile muchiilor și verificăm dacă ciclul minim are costul 0. Mai mult, de aici putem să aflăm dacă $x$ este prea mic sau prea mare. Dacă $x$ este prea mare, atunci va exista un ciclu negativ. Dacă nu, $x$ este bun sau este prea mic.

De aici devine evidentă soluția: o căutare binară a rezultatului, în care verificăm existența unui ciclu negativ. În etapa de verificare se va folosi algoritmul *Bellman-Ford* care are ordinul de complexitate $O(N \cdot M)$. Complexitatea finală a algoritmului este $O(N \cdot M \cdot \log X)$, unde $X$ este valoarea maximă a costului unui arc, înmulțită cu 100 (pentru cele 2 zecimale exacte).

O altă soluție care ar fi putut obține în jur de 70 de puncte, dar care merită menționată, deoarece poate fi folosită și în rezolvarea altor probleme, este bazată pe *programarea dinamică*. Această variantă are avantajul că nu lucrează decât cu numere întregi (cu excepția unor verificări).

Se alege un nod de start $s$ și se construiește (păstrând doar ultimele două linii) matricea $A$ cu semnificația că $A(i, j)$ este costul minim al unui drum care conține exact $i$ arce, care începe cu nodul $s$ și se termină cu nodul $j$. Evident, $A(i, j)$ se calculează folosind valorile $A(i - 1, k)$ pentru care avem arc de la $k$ la $j$.

Valorile care ne interesează sunt $A(i, s)/i$; o astfel de valoare reprezintă lungimea medie a ciclului care începe în $s$, se termină în $s$ și conține exact $i$ arce.

Optimizarea majoră se bazează pe faptul că, odată ce au fost considerate ciclurile care încep dintr-un anumit nod, nu mai este nevoie ca acesta să fie considerat pe viitor. Astfel, pentru un anumit nod $s$, nu sunt calculate decât valorile din matrice pentru care $j \ge s$.

Complexitatea acestei rezolvări este $O(N^2 \cdot M)$; se alege, pe rând, un nod de start, după care valorile sunt construite în $O(N \cdot M)$. Acest fapt este evident dacă observăm că valorile pe o linie a matricei se construiesc în $O(M)$, deoarece valorile unui element depind de valorile asociate vecinilor din graf, aflate pe linia precedentă.

## P080312: Prefix

Pentru fiecare șir se calculează funcția *prefix* corespunzătoare algoritmului *Knuth-Morris-Pratt* (*KMP*).

Se consideră că pozițiile caracterelor din șir sunt numerotate de la 1 la $N$ (lungimea șirului). Lungimea celui mai lung prefix periodic este, dacă un astfel de prefix există, cel mai mare număr natural $K$ ($1 < K \le N$) , având proprietățile:

- $P[K] > 0$;
- $K$ *modulo* $(K - P[K]) = 0$.

Prin $P[K]$ s-a notat valoarea funcției prefix corespunzătoare poziției $K$. Dacă nu vi se pare evident că un astfel de șir este periodic, încercați să demonstrați.

Dacă prefixul de lungime $K$ al șirului este periodic, atunci $K - P[K]$ reprezintă lungimea celei mai scurte perioade a acestui prefix.

De exemplu, pentru șirul "abcababcababca", funcția prefix determinată este următoarea:

```
      a b c a b a b c a b  a  b  c  a
   K: 1 2 3 4 5 6 7 8 9 10 11 12 13 14
P[K]: 0 0 0 1 2 1 2 3 4 5  6  7  8  9
```

Cel mai lung prefix periodic are lungimea 10, deoarece $P[10] = 5 > 0$ și 10 *mod* $(10 - P[10]) = 10$ *mod* $5 = 0$. De asemenea, se observă că lungimea perioadei "abcab" este $5 = 10 - P[10]$.

### Analiza complexității

Operația de citire a datelor de intrare are ordinul de complexitate $O(N)$.

Calculul funcției prefix din cadrul algoritmului *KMP* are ordinul de complexitate $O(N)$. Pentru fiecare valoare a funcției prefix, verificarea ($P[K] > 0$ și $K$ *modulo* $(K - P[K]) = 0$) are ordinul de complexitate constant.

În concluzie, ordinul de complexitate al algoritmului este $O(N)$.