# Finala BURSELE AGORA 2003/2004

*Cosmin-Silvestru Negrușeri*

**Vă prezentăm în continuare soluțiile celor patru probleme propuse spre rezolvare la etapa finală a celei de-a V-a ediții a concursului de programare organizat de revista noastră.**

## P040619: Cifre

Această problemă este o variație a părții de decompresie a unui algoritm clasic de compresie apărut în 1994.

David Wheeler

Algoritmul amintit se numește *BWT* (*Borrows-Wheeler Transform*) și metoda de compresie este cea explicată în textul problemei.

De fapt, acest algoritm, nu este unul de compresie ci unul de transformare a informației. Informația transformată poate duce la obținerea unei rate de compresie mai mare decât cea netransformată datorită faptului că în urma transformării apar șiruri de caractere consecutive identice mai lungi și mai multe. Publicația originală poate fi studiată accesând adresa:

http://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html

Problema propusă a fost folosită și la finala concursului *Bursele Agora* din 2002 precum și la *Olimpiada Internațională de Informatică* din 2001 (ca problemă de rezervă).

Problema curentă prezintă o mică modificare față de cea inițială, șirurile fiind sortate în ordine descrescătoare. Algoritmul original are șirurile ordonate ascendent.

Putem trece ușor peste această modificare, transformând la începutul algoritmului, fiecare cifră în: `10 - cifra originală`, aplicând algoritmul original, iar la sfârșit aplicând din nou aceeași transformare, pentru a obține șirul corect.

Să vedem cum s-au transformat șirurile din exemplu:

• *prima transformare*

```
2457   8653
4572   6538
5724   5386
7245   3865
```

• *sortarea elementelor*

```
7245   3865
5724   5386
4572   6538
2457   8653
```

| | F | | | | | | L |
|---|---|---|---|---|---|---|---|
| 4 | B | B | S | D | R | D | O |
| 5 | B | S | D | R | D | O | B |
| 2 | D | O | B | B | S | D | R |
| 0 | D | R | D | O | B | B | S |
| 3 | O | B | B | S | D | R | D |
| 1 | R | D | O | B | B | S | D |
| 6 | S | D | R | D | O | B | B |

Primul pas al algoritmului nostru de decompresie sortează cifrele ultimei coloane, pentru a obține cifrele primei coloane (aici: `3`, `5`, `6`, `8`).

Pentru a realiza aceasta putem folosi un algoritm de sortare prin numărare.

Algoritmul de rezolvare a acestei probleme constă în următorii pași, excluzând transformările de la început și sfârșit:

- se determină frecvența de apariție a cifrelor, ceea ce permite obținerea primei coloane a matricei (prima coloană conține întotdeauna cifrele ordonate).
- se construiește un șir de corespondențe $A$ între cifrele de pe prima coloană și cele de pe ultima, ținând cont de faptul că celei de-a $i$-a apariții a cifrei $c$ de pe prima coloană îi corespunde cea de-a $i$-a apariție a cifrei $c$ de pe ultima.
- pentru determinarea numărului cerut se scriu cifrele corespunzătoare pozițiilor $A[1]$, $A[A[1]]$, $A[A[A[1]]]$ etc.; pentru aceasta se folosește un indice $i$ care, după fiecare pas va primi valoarea $A[i]$.

Acest algoritm a fost demonstrat matematic de către cei doi cercetători și algoritmul a primit numele amândurora.

### Analiza complexității

Datele de intrare constau în citirea șirului cifrelor de pe ultima coloană; deoarece avem $N$ litere, ordinul de complexitate al operației este $O(N)$.

Determinarea frecvențelor de apariție se realizează în timp *liniar* prin simpla parcurgere a șirului.

Construirea șirului de corespondențe se realizează tot în timp *liniar* deoarece la fiecare pas este realizată câte o corespondență.

Pentru afișarea datelor de ieșire (a cuvântului căutat) se realizează o simplă parcurgere a șirului de corespondențe, deci această operație are ordinul de complexitate $O(N)$.

Transformarea cifrelor la începutul și sfârșitul algoritmului se realizează *liniar*.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(N) + O(N) + O(N) + O(N) + O(N) = \mathbf{O(N)}$.

## P040620: Produse

Pentru o permutare data $p$, produsul elementelor va fi: $(n!)^p \cdot (n!)^u$. Deoarece vor fi înmulțite $n!$ permutari, rezultatul va fi $(n!)^{p+u+1}$.

Pentru un număr întreg $X$, numărul $K$ de cifre ale acestuia va satisface următoarea inegalitate: $10^K \le X < 10^{K+1}$.

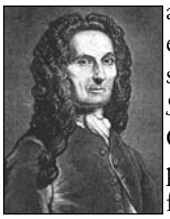
Logaritmând această inegalitate (în baza 10), obținem: $K \le \log_{10} X < K + 1$.

Aplicând această inegalitate în problema noastră obținem: $K \le (p + u + 1) \cdot \log_{10} n! < K + 1$.

Pentru a determina soluția acestui sistem de inegalități, putem fie să folosim egalitatea $\log(a \cdot b) = \log a + \log b$ obținând $\log_{10} n! = \log_{10} 1 + \log_{10} 2 + ... + \log_{10} n$, fie să folosim formula lui *Stirling* de aproximare a factorialului: $n! \approx (2 \cdot \pi \cdot n)^{1/2} \cdot (n/e)^n \cdot e^{1 + 1/(12 \cdot n) + O(1/n2)}$.

Pentru prima variantă, ordinul de complexitate al algoritmului este $\mathbf{O(n)}$, iar pentru a doua ordinul de complexitate este $\mathbf{O(1)}$.

Mai trebuie menționat faptul că această formulă îi este atribuită incorect scoțianului *James Stirling* (1692 - 1770). Cel care a enunțat pentru prima dată a fost matematicianul francez *Abrahan de Moivre* (1667 - 1754).



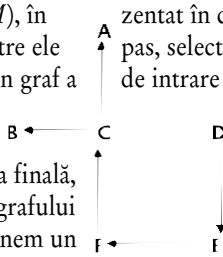
Abraham de Moivre

## P040621: Drum

Această problemă poate fi rezolvată combinând algoritmul de sortare topologică a nodurilor unui graf aciclic cu un algoritm de programare dinamică.

A sorta topologic nodurile unui graf înseamnă a ordona nodurile într-un șir $A$, astfel încât pentru oricare doi indici $i$ și $j$, cu: $i < j$, nu există nici un drum de la nodul $A_i$ la nodul $A_j$. Practic, această ordonare garantează o permutare în care nici un nod nu apare mai devreme decât strămoșii lui.

Putem realiza această ordonare într-un timp de ordinul $O(M)$, în mai multe moduri. Unul dintre ele este următorul: eliminarea din graf a nodurilor cu grad de intrare zero și introducerea lor în șirul ce va reprezenta ordinea finală, repetarea acestui pas asupra grafului astfel obținut până când obținem un graf cu zero noduri. O altă idee este aplicarea algoritmului de căutare în lățime puțin modificat.




Varianta în pseudocod a acestui algoritm este:

```
algoritm CăutareÎnLățime(nod)
  parcurs_nod ← adevărat;
  pentru fiecare nod fiu adiacent
                 nodului nod execută
    dacă nu parcurs_fiu
      CăutareÎnLățime(fiu)
    sfârșit dacă
  sfârșit pentru
  adaugă la începutul listei A nodul
                                nod
sfârșit algoritm
```

După aplicarea acestui algoritm, ne va fi ușor să calculăm pentru fiecare *nod* două numere: $maxD_{nod}$ reprezentând drumul cel mai lung care începe din nod și $numD_{nod}$ reprezentând numărul de drumuri de lungime maximă care încep în *nod*.

Obținerea acestor valori se poate realiza conform următoarei secvențe descrise folosind limbajul pseudocod:

```
dacă maxD_nod < maxD_fiu + 1 atunci
  maxD_nod ← maxD_fiu + 1;
  numD_nod ← numD_fiu
sfârșit dacă
dacă maxD_nod < maxD_fiu + 1 atunci
  numD_nod ← numD_nod + numD_fiu
sfârșit dacă
```

Valorile vor fi calculate corect dacă *fiu* va fi procesat înainte de *nod*, lucru asigurat prin procesarea nodurilor în ordine inversă față de ordinea dată de algoritmul de sortare topologică.

Un algoritm recursiv care ne va ajuta să obținem al $k$-lea cel mai lung drum în ordine lexicografică este prezentat în cele ce urmează. La primul pas, selectăm dintre nodurile cu grad de intrare zero și care sunt primele noduri în cel puțin un drum de lungime maximă; aceste noduri vor fi procesate în ordine lexicografică.

Dacă considerăm aceste noduri sortate în ordine lexicografică și

$$sumD_{i_curent} = numD_{1_curent} + numD_{2_curent} + ... + numD_{i_curent},$$

atunci al $k$-lea lanț începe în nodul *x_curent* cu proprietatea $sum_{x-1_curent} < k \le sum_{x_curent}$.

Repetând acest pas pentru succesorii lui *x_curent* vom obține în final întregul drum.

### Analiza complexității

Ordinul de complexitate al operației de citire a datelor este $O(N)$, iar ordinul de complexitate al algoritmului de sortare topologică este $O(M + N)$.

Algoritmul care determină prin metoda programării dinamice *numD* și *maxD* are ordinul de complexitate $O(M + N)$, iar algoritmul recursiv are același ordin de complexitate $O(M + N)$. Prin urmare, ordinul de complexitate al întregului algoritm este $O(M + N)$.

Cu toate acestea, există algoritmi cu ordinul de complexitate $O(N^3)$ care ar fi putut duce la soluții care s-ar fi încadrat în timpul de execuție admis.

O capcană a acestei probleme a fost specificarea faptului că ordinea lexicografică considerată nu este cea

utilizată în cadrul codului *ASCII*, ci literele mici erau considerate din punct de vedere lexicografic ca fiind mai mici decât literele mari.

Citirea neatentă a propoziției ce specifică acest lucru a dus pentru unul dintre concurenți la pierderea a 100 de puncte la această problemă, rezolvarea lui fiind corectă dacă nu ar fi fost prezentă această precizare.

## P040622: Numere

Un algoritm corect pentru rezolvarea acestei probleme constă într-o căutare exhaustivă în spațiul soluțiilor.

Totuși, o căutare naivă, care încearcă fiecare permutare ar fi dus la obținerea a doar jumătate din punctajul maxim.

Algoritmul de rezolvare propus este este mult mai rapid și este bazat pe anumite observații matematice.

Dacă în soluția noastră avem: $b_i \le b_{i+1} \le ... \le b_j$, atunci suma $|b_i - b_{i+1}| + |b_{i+1} - b_{i+2}| + ... |b_{j-1} - b_j|$ se transformă (explicitând modulele) în $b_{i+1} - b_i + b_{i+2} - b_{i+1} + ... + b_j - b_{j-1} = b_j - b_i$, iar în cazul în care avem $b_i \ge b_{i+1} \ge ... \ge b_j$, atunci suma $|b_i - b_{i+1}| + |b_{i+1} - b_{i+2}| + ... |b_{j-1} - b_j|$ se transformă (explicitând modulele) în $b_i - b_{i+1} + b_{i+1} - b_{i+2} + ... + b_{j-1} - b_j = b_i - b_j$.

Pe baza acestei observații putem trage concluzia că daca notam cu *x_i* minimele locale (indicii *x* cu proprietatea $b_{x-1} \ge b_x \le b_{x+1}$) iar cu *y_i* maximele locale (indicii *y* cu proprietatea $b_{y-1} \le b_y \ge b_{y+1}$), atunci observăm că suma $|b_1 - b_2| + .. + |b_n - b_1|$ este dublul sumei elementelor *y_i* din care se scade dublul sumei elementelor *x_i*).

Dacă sortăm elementele șirului *a*, putem folosi un algoritm recursiv `back(x, nmin, nmax, currentsum)`, unde *x* este indicele curent din *a*, *nmin* este numărul de minime locale pe care le avem la pasul curent, *nmax* numărul de maxime locale și *currentsum* suma curentă.

La fiecare pas numărul minimelor locale trebuie să fie mai mare sau egal decât numărul de maxime locale, iar în final numărul minimelor locale trebuie să fie egal cu numărul maximelor locale.

Soluția va fi reprezentată sub forma unui șir *sol*, în care $sol_i = 0$ dacă $a_i$ nu este nici minim local nici maxim local, $sol_i = 1$ dacă $a_i$ este minim local, iar $sol_i = 2$ dacă $a_i$ este maxim local. Putem observa de asemenea că $a_1$ este minim global, deci și minim local, iar $a_n$ este maxim global, deci și maxim local.

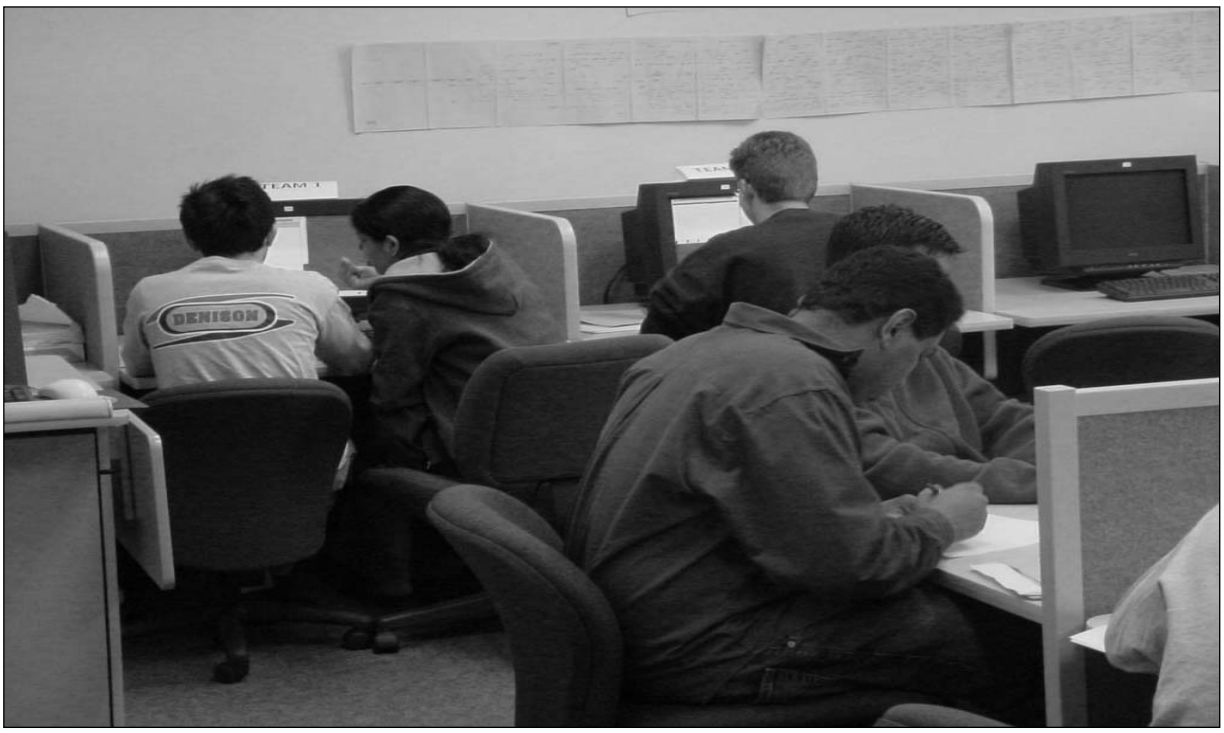
Implementarea acestei idei ar fi dus la obținerea unui punctaj maxim.

O altă idee abordată cu succes de un concurent a fost o căutare locală randomizată.

Mai concret, concurentul a pornit cu o permutare aleatoare a șirului *a* și a încercat interschimbări de elemente $a_i$ cu $a_j$ pentru a obține o sumă cât mai apropiată de suma cerută. Dacă nu se mai putea realiza o optimizare a soluției curente, concurentul încerca să pornească de la o nouă permutare.

Alți concurenți au încercat să implementeze idei asemănătoare, dar nu au obținuit rezultate comparabile.

### Analiza complexității

Soluția prezentată are cu siguranță un ordin de complexitate cel mult egal cu $O(3^n)$. Restricția suplimentară ca numărul de minime locale să fie mai mic decât numărul de maxime locale, ne duce cu gândul la numărul de parantezări și la numerele lui *Catalan*.

Pentru soluția inventivă randomizată este și mai greu de estimat ordinul de complexitate. De astfel de soluții avem nevoie în cazul unor probleme foarte greu abordabile cu algoritmii clasici.