CONCURSUL de programare AGORA

Cea de-a doua rundă a Concursului de Programare Agora s-a încheiat. În acest număr vă vom prezenta soluțiile oficiale ale celor trei probleme, precum și rezultatele obținute de concurenți la această etapă. De asemenea, veți putea vedea care este clasamentul intermediar, după două etape.

Conform așteptărilor noastre, au existat o mulțime de concurenți care nu au reușit să se înscrie în timp util pentru a participa la prima rundă. Din acest motiv, numărul participanților la a doua rundă a fost mai mare decât cel de la prima.

La sfârșitul anului 2000 numărul celor înscriși la *Concursul de Programare Agora* s-a apropiat de 400. Dintre aceștia, mai mult de 100 sunt concurenți din afara României.

Din păcate, au existat câțiva concurenți care, deși au obținut punctaj maxim la prima rundă, nu au mai trimis rezolvări și pentru runda a doua, pierzând astfel șansa de a se afla între primii clasați.

De asemenea, mulți dintre cei care au avut punctaje mari la prima rundă, au avut rezultate aproape dezastruoase la cea de-a doua. Au fost și concurenți cu rezultate mai slabe la prima rundă care au obținut punctaje excelente la cea de a doua.

A doua rundă

Cele trei probleme de la prima rundă nu au mai fost la fel de simple ca cele de la prima. Acest lucru este dovedit de numărul relativ mic al celor care au obținut punctaj maxim. Au acumulat 300 de puncte doar trei concurenți, față de cei 28 de la prima rundă.

Două dintre cele trei probleme necesitau folosirea unui algoritm foarte cunoscut pentru obținerea rezultatului. Cea de-a treia problemă, deși nu era foarte cunoscută, a fost rezolvată de foarte mulți concurenți, deoarece a fost indicată destul de clar o "sursă de inspirație" pentru rezolvare.

La prima rundă trebuiau simulate operații cu numere întregi foarte mari pentru toate problemele. La această rundă, doar una dintre probleme necesita efectuarea de astfel de operații dar, de data aceasta, numerele erau reale.

Erori

Nu am scăpat nici la runda a doua de programele care foloseau unit-ul `CRT`. Reamintim faptul că evaluarea este realizată pe un calculator dotat cu un procesor *Intel Pentium III* la 500 MHz. Ca urmare, datorită unui bug din *Borland Pascal 7.0*, programele care folosesc unit-ul `CRT` realizează o împărțire la zero chiar la începutul execuției. Din acest motiv, toate programele care folosesc acest unit nu vor mai furniza rezultatul corect, deoarece vor fi întrerupte înainte de a efectua vreun calcul.

Din nou s-a folosit tipul `Comp` fără a include directiva de compilare `{$N+}`. Din acest motiv câteva programe, care ar fi putut fi corecte, nu au primit nici un punct datorită faptului că nu a fost posibilă compilarea lor.

O categorie destul de ciudată de erori o constituie cele apărute în redactarea fișierelor **`RUNDA2.BAT`**. Unii au încercat să compileze fișiere **`.CPP`**, folosind compilatorul pentru *Pascal*. Alții au încercat să compileze fișiere pe care, de fapt, nu le-au trimis, sau le-au trimis cu un alt nume. De asemenea, mulți nici nu au considerat că merită efortul de a mai crea fișierul. La această rundă am creat noi acest fișier, dar începând cu runda a treia nu o vom mai face. Cei care vor trimite arhive care nu conțin acest fișier nu vor primi nici un punct la runda respectivă.

Așteptăm părerile voastre...

Noi am dori ca *CPA* să fie un concurs pe gustul tuturor. Știm că acest lucru nu poate fi realizat, dar cu ajutorul vostru, concursul ar putea fi mai aproape de așteptări. Așadar, nu ezitați să ne scrieți!

RUNDA 2 - Top 50

Mohammad Ali Safari Ghahsareh, Iran 300
Cristian Băicoianu, Ploiești 300
Răzvan Mușaloiu-Elefteri, București 300
Bogdan Dumitru, Ploiești 290
Ciprian Cană, Vaslui 286
Maximilian Machedon, București 274
Cristian Toth, Lugoj 272
George Drumea, București 271
Mihai Stroe, București 263
Andrei Benea, Focșani 254

11. Radu Vătavu, Suceava 238
12. Liviu Lalescu, Craiova 233
13. Claudiu Gruia, București 232
14. Ioana-Maria Ileană, Alba Iulia 224
14. Adrian Cârcu, Bistrița 224
14. Radu Ștefan, Brașov 224
14. Cristian Alexandrescu, Botoșani 224
14. Vlad Vâlceanu, Arad 224
14. Victor Asavei, Dr. Tn. Severin 224
20. Csaba Andras, Oradea 220
20. Csaba Patcas, Oradea 220
22. Gabriel Pârvan, Râmnicu Sărat 218
23. Radu Cocieru, Sibiu 215
24. Daniel Crișan, Râmnicu Vâlcea 209
24. Cosmin Răianu, Constanța 209
26. Denis Bogdănaș, Iași 206
27. Mihai Pătrașcu, Craiova 205
28. Flaviu Pașca, Bistrița 200
28. Vincent Groenhuis, Olanda 200
28. Dan Ghinea, București 200
28. Valentin Bisa, Lugoj 200
28. Mihail Piț-Rada, Dr. Tn. Severin 200
33. Valentin Pop, Carei 190
34. Paul Marinescu, Buzău 189
34. Cristian Ioniță, București 189
36. Petru-Simon Moț, Brăila 187
37. Daniel Dumitran, București 185
38. Horia Ciurdar, Timișoara 184
39. Mihail Popa, București 172
39. Bogdan Stan, Câmpina 172
41. Adrian Balasko, Zalău 162
42. Andrei David, Râmnicu Vâlcea 161
43. Sorin Oțescu, Brașov 158
44. Dan Popovici, Arad 153
45. Lucian Burja, Blaj 144
46. Vlad Dascălu, Bacău 142
47. Ionuț Andreica, București 139
48. Marius Andrei, București 133
49. Bogdan Piloga, Timișoara 132
49. Lucian-Daniel Stanciu, Brăila 132

Acadele

Această problemă a fost rezolvată corect de către 71 dintre cei care au participat la a doua rundă a ***Concursului de Programare Agora***. Așa cum se preciza și în enunț rezolvarea putea fi găsită într-o culegere publicată la editura *Computer Libris Agora*. Titlul culegerii este ***Un Atelier de programare*** și a fost scrisă de ***dl. prof. Cristian Giumale***.

Rezolvarea, folosind metoda *backtracking* sau o altă metodă de generare și verificare a tuturor configurațiilor `(B,L,D)`, nu se va încadra în timp deoarece spațiul care trebuie explorat este foarte mare și parametrii problemei depind funcțional unii de alții.

Cele N acadele produse zilnic sunt ambalate câte L în B cutii folosindu-se N ambalaje diferite, corespunzând celor N tipuri disponibile. Rezultă ecuația $B \cdot L = N$.

Restricția, potrivit căreia la sfârșitul fiecărui ciclu de producție, pentru fiecare pereche de ambalaje diferite există o singură cutie care conține exact două acadele cu cele două ambalaje, celelalte având ambalaje diferite față de cele două, poate fi reformulată astfel: *pentru fiecare tip de ambalaj din cele N (fie el x) și pentru fiecare orice alt tip y de ambalaj din cele N - 1 rămase, există o singură cutie cu ambalajele x și y, produsă în cele D zile de producție; într-o cutie, x se află împreună alături de L - 1 ambalaje diferite, iar x nu mai poate apărea în alte cutii produse în aceeași zi. Rezultă că după cele D zile, fiind zilnic alături de L - 1 ambalaje diferite, x trebuie să acopere cele N - 1 posibilități de asociere; se obține ecuația* $D \cdot (L - 1) = (N - 1)$.

La cele două ecuații se adaugă inegalitatea $B > 1$, care este prezentă în enunț.

Se obține următorul sistem nedeterminat:

$$B \cdot L = N \qquad (*)$$
$$D \cdot (L - 1) = (N - 1) \qquad (**)$$
$$B > 1 \qquad (***)$$

În acest sistem valorile L, B, N și D sunt numere întregi pozitive.

În continuare vom arăta că o soluție a acestui sistem este și o soluție a problemei noastre.

Teoremă

Fie L_V o valoare a variabilei L care satisface ecuațiile (*) și (**). Atunci L_V satisface toate restricțiile problemei.

Demonstrație

Fie B_V și D_V valorile întregi corespunzătoare valorii L_V. Deoarece aceste valori sunt obținute din ecuația (*), atunci primele trei restricții din enunț sunt, evident, îndeplinite. Vom arăta acum că și ultima restricție este satisfăcută.

Fie x un ambalaj, ales la întâmplare din cele N. Conform primei restricții, x apare într-o singură cutie produsă într-o anumită zi a ciclului de producție, iar pe întreg ciclul, x apare în D_V cutii. Să presupunem că x nu respectă ultima restricție. Atunci, există cel puțin două cutii care, în afară de x, conțin alte ambalaje comune și, deci, numărul de învelișuri distincte din cele două cutii este mai mic decât $2 \cdot (L_V - 1)$. Înseamnă că cele D_V cutii în care se găsește x conțin împreună $D_V \cdot (L_V - 1) < N - 1$ ambalaje, iar ecuația (**) nu este satisfăcută. Acest lucru este imposibil deoarece ipoteza ne asigură că L_V și D_V satisfac această ecuație. Prin urmare, ultima restricție este satisfăcută pentru x și, deoarece x a fost ales aleator, este satisfăcută pentru oricare dintre cele N ambalaje.

Această teoremă justifică un algoritm de rezolvare foarte simplu: dintre toate soluțiile sistemului ecuațiilor (*), (**), (***) se alege soluția pentru care valoarea $|D - B|$ este minimă.

```
algoritm acadele(N)
  min ← ∞
  pentru L=2,N-1 execută
    dacă rest[N/L]=0 şi rest[(N-1)/(L-1)]=0
    atunci // o soluţie posibilă
      B ← N/L
      D ← (N-1)/(L-1)
      dacă |D-B| < min atunci
        Lmin ← L
        Bmin ← B
        Dmin ← D
        min ← |D-B|
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
    dacă min ≠ ∞ atunci
      returnează Lmin,Bmin,Dmin
    altfel returnează 0,0,0
sfârşit algoritm
```

Acest algoritm este liniar și funcționează într-un timp acceptabil pentru valori medii ale lui N, dar pentru valori mari nu se va mai încadra în limita de timp admisă.

Din acest motiv algoritmul nu poate rămâne în această formă. Vom prezenta în continuare o adaptare care se va executa într-un timp subliniar și anume $O(\sqrt{N})$.

Problema are două proprietăți importante. Prima restrânge intervalul de căutare a soluției: este suficient ca valorile L care satisfac restricțiile problemei să fie căutate în intervalul $[2, \sqrt{N}]$.

A doua proprietate arată că diferența $D - B$ este pozitivă pentru L cuprins între 2 și $\sqrt{N}$, iar valoarea maximă a diferenței se obține pentru valoarea maximă a lui L, care satisface restricțiile problemei.

Vom demonstra în continuare cele două proprietăți.

Lemă

Nu există nici o valoare $\sqrt{N} < L < N$ pentru care ecuațiile (*) și (**) sunt satisfăcute.

Demonstrație

Fie ecuațiile (*) și (**) rescrise în forma:

```
N mod L = 0                    (*)
(N - 1) mod (L - 1) = 0        (**)
```

Să considerăm că există o valoare $\sqrt{N} < L < N$ care satisface ecuațiile (*) și (**). Atunci, trebuie să existe doi întregi, $q_1 > 0$ și $q_2 > 0$, astfel încât

$$N = q_2 \cdot L$$
$$N - 1 = q_1 \cdot (L - 1)$$

Din aceste ecuații obținem

$$q_1 = \frac{q_2 \cdot L - 1}{L - 1} = \frac{q_2 \cdot (L-1) + q_2 - 1}{L - 1} = q_2 + \frac{q_2 - 1}{L - 1}.$$

Avem $\sqrt{N} < L$; din ecuația $N = q_2 \cdot L$ obținem $q_2 < \sqrt{N}$, deci $q_2 < L$. Atunci, raportul dintre $q_2 - 1$ și $L - 1$ este subunitar și obținem $q_2 \le q_1 < q_2 + 1$.

Deoarece q_1 și q_2 sunt valori întregi rezultă că $q_1 = q_2$. Obținem $N / L = (N - 1) / (L - 1)$, adică

$$N \cdot L - N = N \cdot L - L.$$

Obținem egalitatea $N = L$, ceea ce contravine ipotezei.

Lemă

Valoarea minimă pentru $|D - B|$ se obține pentru cea mai mare valoare L din intervalul $(1, N)$ care satisface ecuațiile (*) și (**).

Demonstrație

Se observă că $D - B$ este o funcție care depinde de L:

$$f(L) = \frac{N-1}{L-1} - \frac{N}{L} = \frac{N - L}{L \cdot (L-1)}.$$

Vom arăta că această funcție este strict descrescătoare pentru valorile L cuprinse în intervalul $[2, \sqrt{N}]$.

Vom considera două valori L_1 și L_2 ($L_1 < L_2$) din acest interval și vom demonstra că $f(L_1) > f(L_2)$.

Trebuie să demonstrăm că

$$\frac{N - L_1}{L_1 \cdot (L_1 - 1)} > \frac{N - L_2}{L_2 \cdot (L_2 - 1)}.$$

Avem $(N - L_1) \cdot L_2 \cdot (L_2 - 1) > (N - L_2) \cdot L_1 \cdot (L_1 - 1)$ care se reduce, după efectuarea unor calcule la

$$(L_2 - L_1) \cdot (N \cdot L_2 + N \cdot L_1 - N - L_1 \cdot L_2) > 0$$

Primul termen al produsului este întotdeauna pozitiv deoarece $L_2 > L_1$.

Al doilea termen poate fi scris sub forma

$$N \cdot (L_2 - 1) + L_1 \cdot (N - L_2).$$

Datorită intervalului ales pentru valorile L_1 și L_2, și acest termen este pozitiv.

În concluzie, inegalitatea inițială este adevărată pentru intervalele care ne interesează, deci funcția $f(L)$ este descrescătoare pe intervalul considerat.

Rezultă imediat că valoarea minimă posibilă a expresiei $|D - B|$ este obținută pentru cea mai mare valoare L din intervalul $(1, N)$ care satisface ecuațiile (*) și (**).

Din aceste două leme și teorema prezentată la început rezultă că soluția problemei este unică și corespunde valorii maxime a parametrului L care satisface ecuațiile (*) și (**).

Se obține astfel următorul algoritm:

```
algoritm acadele2(N)
  pentru L=[sqrt(N)], 2 pas -1 execută
    dacă rest[N/L]=0 şi rest[(N-1)/(L-1)]=0
      atunci
       returnează L, N/L, (N-1)/(L-1)
    sfârşit dacă
  sfârşit pentru
  returnează 0, 0, 0
sfârşit algoritm
```

Acest algoritm poate fi implementat foarte ușor chiar dacă demonstrarea corectitudinii sale a necesitat câteva observații matematice nu foarte ușor de demonstrat.

Vă prezentăm în continuare versiunea oficială a soluției acestei probleme. Aceasta a fost adaptată după rezolvarea propusă de ***dl. prof. Cristian Giumale*** în culegerea sa, ***Un atelier de programare***.

Listing LOL.PAS

```
#include <stdio.h>
#include <math.h>

void main(void){
  long N,L;
  FILE *f=fopen("LOL.IN","rt");
  fscanf(f,"%ld",&N);
  fclose(f);
  for (L=sqrt(N);L>1;L--)
    if (!(N%L) && !((N-1)%(L-1))
      break;
  f=fopen("LOL.OUT","wt");
  if (L==1) fprintf(f,"0 0 0");
  else fprintf("%ld %ld %ld",L,N/L,
                                (N-1)/(L-1));
  fclose(f);
}
```

Au rezolvat corect...

Cristian Alexandrescu, Botoșani
Csaba Andras, Oradea
Marius Andrei, București
Mugurel Ionuț Andreica, București
Liviu-Cosmin Andreicuț, București
Victor Asavei, Drobeta Turnu Severin
Constantin Asofiei, Târgu Neamț
Cristian Băicoianu, Ploiești
Ciprian-Nicolae Baicu, Dr. Turnu Severin
Adrian Balasko, Zalău
Andrei Benea, Focșani
Valentin Bișa, Lugoj
Denis Bogdănaș, Iași
Vencel Bors, Oradea
Lucian Burja, Blaj
Ciprian Cană, Vaslui
Adrian Cârcu, Bistrița
Horia Ciurdar, Timișoara
Radu Cocieru, Sibiu
Horea Coroiu, Cluj-Napoca
Daniel Crișan, Râmnicu Vâlcea
Andrei Csibi, Cluj-Napoca
Vlad Dascălu, Bacău
Laurent Demonet, Franța
Andrei-Liviu Dumbrava, Iași
Octavian-Daniel Dumitran, București
Bogdan Dumitru, Ploiești
Aurel-Mihai Fulger, Constanța
Ciprian Gheorghe, București
Dan Ghinea, București
Andrei Giurgiu, București
Mihai Gojol, Cluj-Napoca
Vincent Groenhuis, Olanda
Claudiu Gruia, București
Mihai Hărănguș, Cluj-Napoca
Ioana-Maria Ileană, Alba Iulia
Cristian Ioniță, București
Liviu Lalescu, Craiova
Codruț-Lucian Lazăr, Câmpeni
Maximilian Machedon, București
Harsha Madhyastha, India
Paul Marinescu, Buzău
Mihail-Cosmin Pițˇ-Rada, Dr. Turnu Severin
Alexandru Mosoi, Bacău
Petru-Simon Moț, Brăila
Raluca Mușaloiu-Elefteri, București
Răzvan Mușaloiu-Elefteri, București
Venkitasubramaniam Muthuramakrishnan, India
Bogdan Nicolae, Sibiu
Tiberius Pârcălabu, București
Flaviu Pașca, Bistrița
Csaba Patcas, Oradea
Mihai Pătrașcu, Craiova
Rusu Paul, Cluj-Napoca
Liviu Păunescu, Constanța
Valentin Pop, Carei
Mihail Popa, București
Dan Popovici, Arad
Cosmin Răianu, Constanța
Mohammad Ali Safari Ghahsareh, Iran
Bogdan-Nicolae Șanta, Cluj-Napoca
Lucian-Raul Silistru, Vaslui
Costin Speciac, Buzău
Lucian-Daniel Stanciu, Brăila
Radu Ștefan, Brașov
Radu Ștefan, Curtea de Argeș
Mihai Stroe, București
Cristian Toth, Lugoj
Vlad Vâlceanu, Arad
Radu Vătavu, Suceava
Victor Vernescu, Constanța

Runda 2, problema 2

Puncte

Această problemă a fost rezolvată corect de către 28 dintre cei care au participat la a doua rundă a *Concursului de Programare Agora*. Problema era destul de simplă, un algoritm optim pentru rezolvarea ei putând fi găsit în numeroase cărți care tratează algoritmi care rezolvă probleme de geometrie. Metoda care ar fi trebuit folosită este ***divide et impera***.

Distanța dintre două puncte P_1 de coordonate x_1, y_1 și P_2 de coordonate x_2, y_2 se calculează cu ajutorul formulei

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Două puncte pot să coincidă, caz în care distanța dintre acestea este 0.

Un algoritm banal care rezolvă problema va verifica distanțele dintre toate perechile de puncte și alegerea minimului acestor distanțe. Acest algoritm are ordinul de complexitate $O(n^2)$ și nu poate fi aplicat în cazul nostru, deoarece putem avea foarte multe puncte în plan (maximum 10000).

Pentru a obține un algoritm cu o complexitate mai bună ($O(n \cdot \log n)$) vom folosi metoda ***divide et impera***.

Un apel recursiv al algoritmului va avea ca intrare o submulțime P a mulțimii punctelor și două șiruri X și Y. Punctele din șirul X sunt ordonate în așa fel încât coordonatele x să fie în ordine crescătoare. De asemenea, punctele din șirul Y sunt ordonate în așa fel încât coordonatele y să fie crescătoare.

Pentru a obține ordinul de complexitate dorit, nu ne permitem să efectuăm sortări la fiecare apel recursiv. Pentru a evita acest lucru vom folosi o metodă numită ***presortare***, cu ajutorul căreia vom menține șirurile ordonate fără a le sorta efectiv la fiecare apel recursiv.

Un apel recursiv cu intrările P, X și Y va verifica mai întâi dacă în mulțimea P există mai mult de trei puncte. Dacă avem cel mult trei puncte, atunci vom verifica distanțele dintre toate perechile de puncte (avem o pereche dacă sunt două puncte și trei perechi dacă sunt trei puncte; nu este posibil să avem un punct) și o vom alege pe cea minimă. Pentru mulțimi cu mai mult de trei puncte vom aplica cele trei etape ale algoritmilor divide et impera: *divide*, *stăpânește* și *combină*.

Etapa divide

Vom determina o dreaptă verticală d care împarte mulțimea punctelor P în două submulțimi P_S și P_D, fiecare conținând jumătate din punctele mulțimii P. Dacă numărul de puncte din P este impar, atunci una dintre mulțimi va conține cu un punct mai mult decât cealaltă. Toate punctele din P_S vor fi pe dreapta d sau în stânga acesteia și toate punctele din P_D sunt pe dreapta d sau în dreapta acesteia.

Șirul X este împărțit în șirurile X_S și X_D care conțin punctele din P_S, respectiv P_D, ordonate crescător după coordonata x. Analog, șirul Y este împărțit în șirurile Y_S și Y_D care conțin punctele din P_S, respectiv P_D, ordonate crescător după coordonata y.

Etapa stăpânește

Având împărțită mulțimea P în submulțimile P_S și P_D, vom efectua două apeluri recursive, unul pentru a găsi cea mai apropiată pereche de puncte din P_S și celălalt pentru a găsi cea mai apropiată pereche din P_D.

Pentru primul apel vom folosi ca intrări mulțimea P_S și șirurile X_S și Y_S, iar pentru al doilea mulțimea P_D și șirurile X_D și Y_D. Fie d_S și d_D valorile returnate în urma celor două apeluri și D minimul acestora.

Etapa combină

Cea mai apropiată pereche de puncte din mulțimea P este, fie cea aflată la distanța D, fie o altă pereche de puncte, unul aflat în mulțimea P_S și celălalt aflat în mulțimea P_D. Va trebui să verificăm dacă există o astfel de pereche aflată la o distanță mai mică decât D. Observăm că, dacă există o astfel de pereche, atunci aceste puncte se află la o distanță de cel mult D față de dreapta d. Ca urmare, ele trebuie să se afle într-o regiune de lățime $2 \cdot D$ centrată în jurul dreptei d. Pentru a găsi o pereche vom proceda astfel:

- Construim un șir Y' care conține toate punctele din șirul Y, care se află în interiorul regiunii de lățime $2 \cdot D$. Acest șir va fi sortat crescător după coordonata y a punctelor.
- Pentru fiecare punct p din șirul Y' vom încerca să găsim punctele din Y' care se află la o distanță cel mult D față de p. Așa cum vom arăta ulterior, este necesar să fie considerate doar șapte puncte din Y', care urmează după p. Vom calcula distanțele de la p la cele șapte puncte și vom reține distanța minimă D' a perechii celei mai apropiate.
- Dacă $D' < D$ atunci regiunea verticală conține o pereche mai apropiată decât cea găsită prin apelurile recursive. În acest caz este returnată această pereche și distanța sa D'. Altfel, este returnată perechea cea mai apropiată găsită prin apelurile recursive, împreună cu distanța sa D.

În această descriere a algoritmului am omis câteva detalii de implementare care sunt necesare pentru a obține ordinul de complexitate $O(n \cdot \log n)$. Vom reveni asupra acestora după ce vom demonstra corectitudinea algoritmului.

Corectitudinea acestui algoritm este evidentă, cu excepția a două aspecte. Primul constă în oprirea apelurilor recursive atunci când avem mai puțin de patru puncte deoarece trebuie să ne asigurăm că nu vom încerca niciodată să împărțim o mulțime care conține un singur punct. Al doilea aspect constă în demonstrarea faptului că avem nevoie de verificarea a doar șapte puncte pentru fiecare punct p din Y'. Această proprietate este demonstrată în continuare.

Să presupunem că la un anumit nivel al recursivității, punctele cele mai apropiate se află unul în mulțimea P_S și unul în mulțimea P_D. Fie acestea p_S și p_D; distanța D' dintre acestea este strict mai mică decât D. Punctul p_S trebuie să fie pe dreapta d sau în stânga ei, iar punctul p_D trebuie să fie pe dreapta d sau în dreapta ei; cele două puncte trebuie să se afle la o distanță de cel mult D unități față de dreapta d. Mai mult, distanța pe verticală dintre p_S și p_D trebuie să fie cel mult D. În concluzie, p_S și p_D se află într-un dreptunghi de lungime $2 \cdot D$ și înălțime D, centrat pe dreapta d.

Vom arăta în continuare că există cel mult opt puncte care se pot afla în interiorul unui astfel de dreptunghi. Dreapta d împarte dreptunghiul în două pătrate cu latura de lungime D.

Să considerăm acum pătratul din stânga dreptei d. Deoarece toate punctele din P_S se află la o distanță de cel puțin D față de p_S (altfel după apelul recursiv ar fi fost returnată o altă distanță mai mică), cel mult patru puncte pot fi situate în interiorul acestui pătrat.

În mod analog putem demonstra că cel mult patru puncte se pot situa în pătratul din dreapta dreptei d.

În concluzie, cel mult opt puncte din P se pot situa în interiorul dreptunghiului. Din acest motiv, este suficient să verificăm cel mult șapte puncte care urmează după fiecare punct din Y'.

Presupunem, fără a reduce generalitatea, că punctul p_S se află înaintea punctului p_D în șirul Y'. În acest caz, chiar dacă p_S apare cât mai devreme în șirul Y' și p_D apare cât mai târziu în acest șir, p_D se află pe una dintre cele șapte poziții care urmează după p_S.

Cu aceasta am demonstrat corectitudinea algoritmului de determinare a celei mai apropiate perechi de puncte.

Pentru a obține complexitatea dorită trebuie să ne asigurăm că punctele din șirurile X_S, X_D, Y_S și Y_D care sunt folosite în apelurile recursive, sunt sortate după coordonatele x, respectiv y. De asemenea, punctele din șirul Y' trebuie să fie sortate după coordonate y. Observăm că dacă șirul punctelor din mulțimea P este deja sortat, împărțirea mulțimii P în submulțimile P_S și P_D se realizează foarte ușor în timp liniar.

Observația cheie a rezolvării acestei probleme este aceea că dorim să obținem un subșir sortat al unui șir sortat. Fie un apel particular, având ca date de intrare mulțimea P și șirul Y sortate după coordonata y. După ce am împărțit mulțimea P în submulțimile P_S și P_D, trebuie să formăm în timp liniar șirurile Y_S și Y_D, care vor fi sortate după coordonata y. Pentru aceasta vom examina, în ordine, punctele din șirul Y. Dacă un anumit punct se află în mulțimea P_S atunci îl adăugăm la sfârșitul șirului Y_S, iar dacă se află în mulțimea P_D atunci îl adăugăm la sfârșitul șirului Y_D. Șirurile X_S, X_D și Y' sunt create în mod analog.

Mai rămâne de văzut cum sortăm punctele la început. Vom face o simplă ***presortare*** a lor, adică le vom sorta, o dată, înainte de primul apel recursiv. Șirurile sortate sunt transmise ca parametri în primul apel și apoi sunt reduse pe măsură ce se avansează în recursivitate. Presortarea poate fi realizată într-un timp $O(n \cdot \log n)$.

Vom analiza acum complexitatea algoritmului ***divide et impera***. Să notăm cu $T(n)$ ordinul de complexitate al acestuia. La fiecare pas vom avea două apeluri recursive ale aceluiași algoritm pentru jumătate din puncte la care se adaugă câteva operații care pot fi efectuate în timp liniar. Putem scrie relația de recurență pentru ordinul de complexitate al algoritmului ca fiind $T(n) = 2 \cdot T(n / 2) + O(n)$. Această formulă este valabilă pentru $n > 3$ deoarece ne oprim din recursivitate dacă nu avem cel puțin patru puncte. Pentru $n < 3$ ordinul este $O(1)$, deoarece sunt efectuate un număr constant de operații.

Din formula recursivă a ordinului de complexitate rezultă că acesta este $O(n \cdot \log n)$. Mai trebuie adăugat și algoritmul de presortare, dar ordinul de complexitate nu se schimbă deoarece există metode de sortare care funcționează într-un timp $O(n \cdot \log n)$.

Aceasta a fost o prezentare generală a algoritmului. Ea a fost preluată din cartea ***Introducere în algoritmi*** scrisă de ***T.H. Cormen***, ***C.E. Leiserson***, ***R.R. Rivest***. Evident, în implementare sunt folosite anumite "trucuri" care reduc timpul de execuție. Din acest motiv, versiunea oficială a rezolvării acestei probleme nu respectă toți pașii care au fost descriși aici.

Listing POINTS.CPP

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <values.h>

float huge x[10000], huge y[10000];
int huge px[10000], huge py[20000];
float min,x1,y1,x2,y2;
int n;

void ReadData(void){
  FILE *f=fopen("POINTS.IN","rt");
  fscanf(f,"%d",&n);
  for (int i=0;i<n;i++)
    fscanf(f,"%f%f",&x[i],&y[i]);
}
```

```
void SortX(int left,int right){
  int i=left,
      j=right,
      k=x[px[(left+right)/2]];
  float aux;
  do{
    while (x[px[i]]<k)
      i++;
    while (x[px[j]]>k)
      j--;
    if (i<=j){
      aux=px[i];
      px[i]=px[j];
      px[j]=aux;
      i++; j--;
    }
  } while (i<=j);
  if (j>left)
    SortX(left,j);
  if (i<right)
    SortX(i,right);
}

void SortY(int left,int right){
  int i=left,
      j=right,
      k=y[py[(left+right)/2]];
  float aux;
  do{
    while (y[py[i]]<k)
      i++;
    while (y[py[j]]>k)
      j--;
    if (i<=j){
      aux=py[i];
      py[i]=py[j];
      py[j]=aux;
      i++; j--;
    }
  } while (i>=j);
  if (j>left)
    SortY(left,j);
  if (i<right)
    SortY(i,right);
}

void SortData(void){
  for (int i=0;i<n;i++)
    px[i]=py[i]=i;
  SortX(0,n-1);
  SortY(0,n-1);
  for (i=0;i<n-1;i++)
    if (x[px[i]]==x[px[i+1]] &&
                    y[px[i]]==y[px[i+1]]){
      FILE *f=fopen("POINTS.OUT","wt");
      fprintf(f,"%.3f %.3f\n%.3f %.3f\n",
       x[px[i]],y[px[i]],x[px[i]],y[px[i]]);
      fclose(f); exit(0);
    }
}

inline float dist(int i,int j){
  return sqrt((x[py[i]]-x[py[j]])*
              (x[py[i]]-x[py[j]])+
              (y[py[i]]-y[py[j]])*
              (y[py[i]]-y[py[j]]));
}

void DivideEtImpera
              (int i1,int j1,int i2,int j2){
  int i,j; float half;
  if (j2-i2<3){
    for (i=i2;i<=j2-1;i++)
      for (j=i+1;j<=j2;j++)
        if (dist(i,j)<min){
          min=dist(i,j);
          x1=x[py[i]];
          y1=y[py[i]];
          x2=x[py[j]];
          y2=y[py[j]];
        }
    return;
  }
  half=x[px[(i1+j1)/2]];
  j=0;
  for (i=i2;i<=j2;i++)
    if (x[py[i]]<half)
      py[j2+(++j)]=py[i];
  if (j>1)
    DivideEtImpera(i1,i1+j-1,j2+1,j2+j);
  j=0;
  for (i=i2;i<=j2;i++)
    if (x[py[i]]>half)
      py[j2+(++j)]=py[i];
  if (j>1)
    DivideEtImpera(j1-j+1,j1,j2+1,j2+j);
  j=0;
  for (i=i2;i<=j2;i++)
    if (fabs(x[py[i]]-half)<min)
      py[j2+(++j)]=py[i];
  for (i=j2+1;i<j2+j;i++)
    for (int m=i+1;m<i+8 && m<=j2+j;m++)
      if (dist(i,m)<min){
        min=dist(i,m);
        x1=x[py[i]];
        y1=y[py[i]];
        x2=x[py[m]];
        y2=y[py[m]];
      }
}
```

```
void Solve(void){
  min=MAXFLOAT;
  DivideEtImpera(0,n-1,0,n-1);
}

void WriteSolution(void){
  FILE *f=fopen("POINTS.OUT","wt");
  fprintf(f,"%.3f %.3f\n%.3f %.3f\n",
                                x1,y1,x2,y2);
  fclose(f);
}

void main(void){
  ReadData();
  SortData();
  Solve();
  WriteSolution();
}
```


Au rezolvat corect...

Cristian Alexandrescu, Botoșani
Csaba Andras, Oradea
Victor Asavei, Drobeta Turnu Severin
Cristian Băicoianu, Ploiești
Valentin Bișa, Lugoj
Ciprian Cană, Vaslui
Adrian Cârcu, Bistrița
Radu Cocieru, Sibiu
Daniel Crișan, Râmnicu Vâlcea
Andrei David, Râmnicu Vâlcea
Vincent Groenhuis, Olanda
Claudiu Gruia, București
Ioana-Maria Ileană, Alba Iulia
Artur Kuczapski, Oradea
Maximilian Machedon, București
Mihail-Cosmin Pițigoi-Rada, Dr. Turnu Severin
Răzvan Mușaloiu-Elefteri, București
Sorin Oțescu, Brașov
Gabriel Pârvan, Râmnicu Sărat
Flaviu Pașca, Bistrița
Csaba Patcas, Oradea
Cosmin Răianu, Constanța
Mohammad Ali Safari Ghahsareh, Iran
Bogdan Stan, Câmpina
Mihai Stroe, București
Radu Ștefan, Brașov
Cristian Toth, Lugoj
Vlad Vâlceanu, Arad

2001: O odisee spațială...

Un monolit misterios a apărut într-un parc din Seattle pentru a ne reaminti de structura similară din filmul sau cartea "2001: O odisee spațială".

La fel ca și în cazul monolitului din filmul sau romanul SF, se pare că nimeni nu știe cine a construit această structură bizară. Dimensiunile blocului din Seattle respectă raportul 1/4/9 care apare și în "2001: O odisee spațială".

Există puține indicii referitoare la originea construcției din Seattle. Pe ea nu a fost amplasată nici o plăcuță comemorativă, iar pământul a fost aranjat după ridicarea structurii. Totuși, prezența unor capace de plastic în jurul construcției demonstrează că cei care au amplasat-o sunt niște petrecăreți pământeni și nu reprezentanți ai unei civilizații extraterestre.

Chiar dacă se spune că "2001: O odisee spațială" este filmul preferat al lui Bill Gates, nimeni nu presupune că cei de la Microsoft au construit monolitul. Se spune că ideea a fost una mult prea inteligentă pentru a fi a celor din Redmond.

Principalii suspecți sunt membrii unui grup numit Support the Monolith care, pe 31 decembrie 2000, au efectuat o paradă cu un monolit de lemn pe străzile din Seattle.

Poliția a confiscat monolitul de lemn cam în aceeași perioadă în care a fost ridicat cel de oțel.

Runda 2, problema 3

Determinant

Această problemă a fost rezolvată corect doar de 7 dintre cei care au participat la a doua rundă a *Concursului de Programare Agora*. Numai 85 de concurenți au reușit să obțină puncte la această problemă, majoritatea însă au acumulat mai puțin de 10 puncte.

În principiu, problema nu era foarte dificilă, algoritmi pentru rezolvarea sa putând fi găsiți în numeroase cărți de specialitate. Se pare că cei mai mulți au fost descurajați atunci când au văzut că trebuie să implementeze operații cu numere reale foarte mari. Puțini s-au hotărât să încerce acest lucru, iar unii, mai inventivi, au descărcat de pe *Internet* clase C++ care implementau aceste operații.

După rezolvarea problemei referitoare la numerele mari nu mai rămânea decât să se găsească un algoritm cât mai eficient pentru calcularea determinantului unei matrice. Se poate demonstra matematic că un algoritm care rezolvă această problemă are o complexitate minimă de $O(n^{2+\varepsilon})$ unde $\varepsilon \to 0$, dar nu ajunge niciodată la 0. În acest moment, cel mai performant algoritm pentru calcularea determinantului unei matrice are un ordin de complexitate de aproximativ $O(n^{2.3})$. Totuși, dimensiunea datelor de intrare permitea folosirea unui algoritm având complexitatea $O(n^3)$.

O metodă de a calcula determinantul unei matrice a fost prezentată în cadrul articolului *LU*-factorizare din numărul 7/2000 al ***GInfo***. În acel articol era prezentat modul în care poate fi descompusă o matrice A într-un produs de matrice $L \cdot U$, dintre care prima era inferior triunghiulară (avea doar valori 0 deasupra diagonalei principale), iar una era superior triunghiulară (avea doar valori 0 sub diagonala principală). Mai mult, elementele de pe diagonala principală a matricei U aveau toate valoarea 1.

Folosind formula $\det A \cdot B = \det A \cdot \det B$ rezultă că $\det A = \det L \cdot \det U$. Deoarece matricele L și U sunt triunghiulare, valorile determinanților lor pot fi obținute înmulțind elementele de pe diagonalele principale. Determinantul matricei U va fi 1, deci determinantul matricei A va fi egal cu determinantul matricei L.

În principiu, tot ce avem de făcut este să calculăm o *LU*-factorizare a matricei date și să înmulțim valorile de pe diagonala principală a matricei L. Apare un singur inconvenient; teorema care stătea la baza algoritmului de *LU*-factorizare prevedea că matricele obținute prin tăierea ultimelor k linii și a ultimelor k coloane trebuie să aibă determinant nenul.

Au rezolvat corect:

Cristian Băicoianu, Ploiești
George Drumea, București
Bogdan Dumitru, Ploiești
Dan Ghinea, București
Aurelian Ghiță, Buzău
Răzvan Mușăloiu-Elefteri, București
Mohammad Ali Safari Ghahsareh, Iran

Chiar dacă este puțin probabil ca acest caz să apară, el nu poate fi ignorat. Din aceste motive, trebuie găsit un algoritm care să rezolve și această problemă, aparent minoră.

Soluția o constituie algoritmul de *LUP*-factorizare care, dându-se o matrice A, determină trei matrice L, U și P astfel încât $P \cdot A = L \cdot U$.

Matricele L și U au aceeași semnificație ca și în cazul *LU*-factorizării, iar matricea P este o matrice de permutare, adică o matrice care conține doar elemente 0 și 1, și suma elementelor de pe fiecare linie și de pe fiecare coloană este 1.

Cu alte cuvinte, fiecare linie și fiecare coloană a matricei P conține exact un 1, restul elementelor fiind 0.

Cazul în care algoritmul de *LU*-factorizare nu funcționa corect era atunci când pe diagonala matricei L se obținea valoarea 0. Apariția matricei P ne asigură acum că dacă pe diagonala matricei L apare valoarea 0, atunci valoarea determinantului este 0.

Este evident faptul că determinantul matricei P este 1, deci vom avea și de această dată $\det A = \det L$.

Elementele matematice aflate în spatele *LUP*-factorizării sunt aceleași ca și în cazul *LU*-factorizării. Algoritmul poate fi adaptat destul de ușor.

```
algoritm determinant(A)
  det ← 1
  n ← numărul liniilor matricei A
  pentru i ← 1,n execută                (*)
    p[i] ← i                            (*)
  sfârşit pentru
```

```
  pentru k ← 1,n execută
    pivot ← 0
    pentru i ← k,n execută
      dacă |a_ik|>pivot
        atunci
          pivot ← |a_ik|
          k' ← i
      sfârşit dacă
    sfârşit pentru
    dacă pivot=0 atunci
      returnează 0
    sfârşit dacă
    interschimbă p[k] cu p[k']          (*)
    pentru i ← 1,n execută
      interschimbă a_ki cu a_k'i
    sfârşit pentru
    pentru i ← k+1,n execută
      a_ik ← a_ik / a_kk
      pentru j ← k+1,n execută
        a_ij ← a_ij - a_ik · a_kj
      sfârşit pentru
    sfârşit pentru
    det ← det · a_kk
  sfârşit pentru
  returnează det
sfârşit algoritm
```

Se observă că, la fel ca și în cazul *LU*-factorizării, matricea *A* este factorizată "pe loc". Matricea *P* este păstrată în vectorul *p*; p_i este coloana pe care se află elementul de pe linia *i*, care are valoarea 1.

Pentru a calcula determinantul nu mai este nevoie să determinăm și matricea *P*, motiv pentru care liniile marcate cu `(*)` pot fi eliminate.

Din motive de spațiu nu putem prezenta sursa completă pentru varianta oficială de rezolvare a acestei probleme. Am "eliminat" liniile de cod în care erau implementate operațiile cu numere mari. Vom presupune că acestea sunt definite într-o clasă numită `BigReal` care conține o metodă pentru inițializarea cu un număr real, una pentru înmulțirea cu un număr real și una care returnează partea zecimală a numărului sub forma unui șir de caractere. Definirea acestei clase ar fi trebuit să apară în loc de *#include <bigreal.h>*. Vă prezentăm în continuare această variantă.

Listing DET.CPP

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <bigreal.h>

long double huge a[200][200];
int n;
BigReal det;

void ReadData(void){
  FILE *f=fopen("DET.IN","rt");
  fscanf(f,"%d",&n);
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
      fscanf(f,"%Lf",&a[i][j]);
  fclose(f);
}

void Zero(void){
  FILE *f=fopen("DET.OUT","wt");
  fprintf(f,"0");
  fclose(f);
  exit(0);
}

void LUP(void){
  int kp;
  long double aux;
  det.init(1);
  for (int k=0;k<n;k++){
    long double p=0;
    for (int i=k;i<n;i++)
      if (fabsl(a[i][k])>p){
        p=fabsl(a[i][k]);
        kp=i;
      }
    if (!p)
      Zero();
    for (i=k;i<n;i++){
      aux=a[k][i];
      a[k][i]=a[kp][i];
      a[kp][i]=aux;
    }
    for (i=k+1;i<n;i++){
      a[i][k]/=a[k][k];
      for (int j=k+1;j<n;j++)
        a[i][j]-=a[i][k]*a[k][j];
    }
    det.multiply(a[k][k]);
  }
}

void WriteSolution(void){
  FILE *f=fopen("DET.OUT","wt");
  fprintf(f,"%s",
            det.TheDecimalPartAsAString());
  fclose(f);
}

void main(void){
  ReadData();
  LUP();
  WriteSolution();
}
```

Runda 4

Vă prezentăm în continuare cele trei probleme de la cea de-a patra rundă a Concursului de Programare Agora. Concurenții care nu s-au înscris până la data de 10 Ianuarie 2000 pot participa la concurs începând cu această rundă, dacă trimit mesajul de înscriere până pe data de 20 Februarie 2001.

CPAR4P1: Sfere

Nume fișier sursă: **SPHERE.PAS**, **SPHERE.C** sau **SPHERE.CPP**
Nume fișier de intrare: **SPHERE.IN**
Nume fișier de ieșire: **SPHERE.OUT**
Descriere: Dacă avem o sferă atunci spațiul este împărțit în două zone: cea din interiorul sferei și cea din exterior. Dacă avem două sfere atunci spațiul poate fi împărțit în trei sau patru zone. Dacă sferele nu se intersectează vom avea trei zone: interiorul primei sfere, interiorul celei de-a doua sfere și zona exterioară sferelor. Dacă sferele se intersectează atunci vor fi patru zone: zona comună celor două sfere, zona din interiorul primei sfere (fără zona comună), zona din interiorul celei de-a doua sfere (fără zona comună) și zona din exteriorul sferelor. Dându-se n ($2 \le n \le 1000$) sfere, determinați care este numărul maxim de zone în care poate fi împărțit spațiul aranjând sferele într-un mod convenabil.
Date de intrare: Fișierul de intrare va conține pe prima linie numărul n al sferelor.
Date de ieșire: Fișierul de ieșire va conține numărul maxim de zone în care poate fi împărțit spațiul.
Exemplu:

SPHERE.IN	**SPHERE.OUT**
2	4

Timp de execuție: 1 secundă/test.

CPAR4P2: Asemănare

Nume fișier sursă: **POLY.PAS**, **POLY.C** sau **POLY.CPP**
Nume fișier de intrare: **POLY.IN**
Nume fișier de ieșire: **POLY.OUT**
Descriere: Se dau două poligoane în plan, fiecare având n ($3 \le n \le 100$) vârfuri. Poligoanele sunt date prin coordonatele vârfurilor lor (numere reale cu o precizie de trei zecimale, din intervalul [0, 1000]), în ordine trigonometrică, sau invers trigonometrică. Se cere să se verifice dacă cele două poligoane sunt ***asemenea***.
Date de intrare: Prima linie a fișierului de intrare conține numărul n al vârfurilor poligoanelor. Următoarele n linii conțin coordonatele vârfurilor primului poligon. Următoarele n linii conțin coordonatele vârfurilor celui de-al doilea poligon.
Date de ieșire: Dacă poligoanele sunt asemenea, fișierul de ieșire va conține n numere, pe câte o linie, a i-a linie indicând vârful din al doilea poligon care corespunde celui de-al i-lea vârf din primul poligon (unghiurile corespunzătoare acestor două vârfuri trebuie să fie egale). Dacă poligoanele nu sunt asemenea, fișierul de ieșire va conține doar cifra 0.
Exemplu:

POLY.IN	**POLY.OUT**
5	2
0.000 0.000	1
0.000 4.000	5
2.000 6.000	4
4.000 4.000	3
4.000 0.000	
0.000 1.000	
0.000 3.000	
2.000 3.000	
2.000 1.000	
1.000 0.000	

Timp de execuție: 1 secundă/test.

CPAR4P3: Dreptunghi

Nume fișier sursă: **RECT.PAS**, **RECT.C** sau **RECT.CPP**
Nume fișier de intrare: **RECT.IN**
Nume fișier de ieșire: **RECT.OUT**
Descriere: Se consideră un dreptunghi ale cărui dimensiuni sunt numere naturale cuprinse între 1 și 100. Dreptunghiul trebuie descompus într-un număr minim de pătrate ale căror laturi sunt paralele cu laturile dreptunghiului. Într-un dreptunghi poate fi efectuată o tăietură completă (de la o margine la alta) paralelă cu o latură.
Date de intrare: Prima linie a fișierului de intrare conține valorile m și n, separate printr-un spațiu, reprezentând dimensiunile dreptunghiului.
Date de ieșire: Fișierul de ieșire va conține numărul minim de pătrate în care poate fi împărțit dreptunghiul, respectând regula de efectuare a tăieturilor.
Exemplu:

RECT.IN	**RECT.OUT**
2 4	2

Timp de execuție: 1 secundă/test.

Ginfo Top 100

1. Mohammad Ali Safari Ghahsareh, Iran 600

2. Bogdan Dumitru, Ploiești 590

3. Ciprian Cană, Vaslui 586

4. George Drumea, București 571
5. Mihai Stroe, București 563
6. Radu, Vătavu, Suceava 538
7. Liviu Lalescu, Craiova 533
8. Claudiu Gruia, București 532
9. Cristian Băicoianu, Ploiești 530
10. Cristian Alexandrescu, Botoșani 524
10. Adrian Cârcu, Bistrița 524
10. Ioana-Maria Ileană, Alba Iulia 524
10. Radu Ștefan, Brașov 524

14. Csaba Andras, Oradea 520
15. Vlad Vâlceanu, Arad 519
16. Gabriel Pârvan, Râmnicu Sărat 518
17. Răzvan Mușaloiu-Elefteri, București 516
18. Daniel Crișan, Râmnicu Vâlcea 509
19. Denis Bogdănaș, Iași 506
20. Mihai Pătrașcu, Craiova 505
21. Flaviu Pașca, Bistrița 500
22. Vincent Groenhuis, Olanda 487
22. Dan Ghinea, București 487
24. Daniel Dumitran, București 485
25. Cosmin Răianu, Constanța 483
26. Mihail Popa, București 472
27. Bogdan Stan, Câmpina 472
28. Horia Ciurdar, Timișoara 467
29. Sorin Oțescu, Brașov 458
30. Mugurel Ionuț Andreica, București 432
31. Vlad Dascălu Bacău, 429
32. Dan Popovici Arad, 428
33. Victor Asavei, Dr. Turnu Severin 424
34. V. Muthuramakrishnan, India 421
35. Liviu Păunescu, Constanța 416
36. Vencel Bors, Oradea 409
37. Ciprian Baicu, Dr. Turnu Severin 406
38. Aurelian Ghiță, Buzău 388
39. Alexandra Constantin, Ploiești 374
40. Csaba Patcas, Oradea 366
41. Paul Marinescu, Buzău 365
42. Cristian Toth, Lugoj 352
43. Andrei Csibi, Cluj-Napoca 347
44. Andrei David, Râmnicu Vâlcea 333
45. Raluca Sauciuc, București 330
46. Lucian Burja, Blaj 320
47. Raluca Mușaloiu-Elefteri, București 316
48. Victor Vernescu, Constanța 315
49. Artur Kuczapski, Oradea 311
50. Bogdan Nicolae, Sibiu 309
51. Mihai-Vlad Pantiș, Cluj-Napoca 306
52. Codruț-Lucian Lazăr, Câmpeni 303
53. Arcadie Crăcan, Iași 300
53. Cătălin Kesa, Bușteni 300
53. Cosmin-Silvestru Negruseri, Bistrița 300
56. Valentin Bisa, Lugoj 299
57. Mihai Gojol, Cluj-Napoca 296
58. Maximilian Machedon, București 274
59. Mihai Hărănguș, Cluj-Napoca 271
60. Preslav Nakov, Bulgaria 266
61. Narayanan Arvind, India 264
62. Andrei Benea, Focșani 254
63. Bogdan-Milovan Piloga, Timișoara 254
63. Andrei Giurgiu, București 254
65. Bogdan Șanta, Cluj-Napoca 251
66. Adrian Cearnău, București 250
67. Rusu Paul, Cluj-Napoca 246
68. Cristian Ioniță, București 243
69. Petko Minkov, Bulgaria 241
70. Valentin Pop, Carei 235
71. Andrei Adler, București 228
72. Andrei Markovits, Satu Mare 223
73. Edison Nica, Iași 216
74. Radu Cocieru, Sibiu 215
75. Dao The Phuong, Vietnam 203
76. Damir Korencic Croația 200
76. Mihail Pit-Rada, Dr. Turnu Severin 200
78. Horea Coroiu, Cluj-Napoca 196
78. Radu Dondera, București 196
80. Mojca Miklavec, Slovenia 187
80. Andrei Tătăranu, Buzău 187
80. Pentru-Simon Moț, Brăila 187
83. Aurel-Mihai Fulger, Constanța 181
84. Andrei-Liviu Dumbrava, Iași 164
85. Adrian Balasko, Zalău 162
86. Constantin Asofiei, Târgu Neamț 158
87. Harsha Madhyastha, India 153
88. Radu Ștefan, Curtea de Argeș 140
89. Liviu-Cosmin Andreicuț, București 139
90. Alina Fești, Cluj-Napoca 135
91. Marius Andrei, București 133
92. Lucian-Daniel Stanciu, Brăila 132
93. Cristian Bojinovici, București 129
94. Silviu-Gabriel Udrea, Târgu Jiu 129
95. Nadina Busuioc, Constanța 124
96. Ciprian Gheorghe, București 120
97. Daniel Comșa, Crăciunelu de Jos 118
98. Ștefan Bucur, București 117
99. Costin Speciac, Buzău 116
100. Ionel-Cristian Iană, București 115