Psihologia concursurilor de informatică

Cătălin Frâncu

Notă: Am publicat această carte în 1997 la Editura L&S Infomat. În 20 de ani s-au schimbat multe. Materia predată s-a schimbat mult. Concursurile s-au schimbat mult. Limbajele C și C++ au înlocuit aproape complet limbajul Pascal la concursuri. Mai ales, eu cel de acum nu mai sunt de acord cu unele principii, moduri de exprimare și stiluri de programare din această carte. Totuși, ocazional lumea îmi mai cere o copie a ei și mă simt jenat să le trimit un fișier Word (apropo de principii). De aceea, am publicat cartea online. Am păstrat tot conținutul original, reparând doar greșelile evidente de tipar și convertind formatul la LaTeXși imaginile la TikZ/PGF (cuvântul „migălos" abia începe să descrie aceste conversii). — Cătălin, București, 2 martie 2018.

Această carte îi este dedicată fratelui meu Cristi, căruia îi datorez o mare parte din cunoștințele mele în domeniul programării. Un gând bun pentru familia mea și pentru prietenii mei, care mi-au luat toate îndatoririle de pe umeri cât timp am scris cartea de față. Fără înțelegerea și răbdarea lor, nu mi-aș fi putut duce munca la bun sfârșit.

Cuvânt înainte

În ultimii ani s-au tipărit la noi în țară foarte multe culegeri de teorie și probleme de programare. Fiecare din ele acoperă diverse domenii ale informaticii. Unele își propun să inițieze cititorul în tainele diverselor limbaje de programare, altele pun accentul mai cu seamă pe tehnicile de programare și structurile de date folosite în rezolvarea problemelor. În general, cele din prima categorie conțin exemple cu caracter didactic și exerciții cu un grad nu foarte sporit de dificultate, iar celelalte demonstrează matematic fiecare algoritm prezentat, însă neglijează partea de implementare, considerând scrierea codului drept un ultim pas lipsit de orice dificultate.

Desigur, fiecare din aceste cărți își are rostul ei în formarea unui elev bine pregătit în domeniul informaticii. De altfel, citirea volumului de față presupune cunoașterea temeinică a conținutului ambelor tipuri de materiale enumerate mai sus. Totuși, pornind de la observația că scrierea unui program impune atât conceperea algoritmului și demonstrarea corectitudinii, cât și implementarea lui, ambele etape fiind complexe și nu lipsite de obstacole, am considerat necesară scrierea unui nou volum care să trateze simultan aceste două aspecte ale programării.

În afară de aceasta, după cum și titlul lucrării o spune, cartea se adresează pasionaților de informatică și celor care au de gând să participe la concursurile și olimpiadele de informatică. Concursul include apariția unui factor suplimentar care răstoarnă multe din obișnuințele programării „la domiciliu": timpul. Autorul a avut la dispoziție patru ani ca să descopere pe propria piele importanța acestui factor. Și, mai mult decât durata de timp în sine a concursului - care la urma urmei este aceeași pentru toți concurenții - contează capacitatea fiecăruia de a gestiona bine acest timp.

Dacă în fața calculatorului de acasă, cu o sticlă de Coca-Cola alături și casetofonul mergând, este într-adevăr un lucru lăudabil să justificăm matematic fiecare pas al algoritmului, să nu ne lăsăm înșelați de intuiție și să scriem programul fără să ne grăbim, alocându-ne o jumătate din timp numai pentru depanarea lui, în schimb în timp de concurs lucrurile stau tocmai pe dos. De demonstrații riguroase nu se mai ocupă nimeni,

intuiția este la mare preț și de nenumărate ori este criteriul care aduce victoria, iar timpul de-abia dacă este suficient pentru implementarea programului, despre depanare nemaiîncăpând discuții. În multe cazuri, cele două etape ale programării - conceperea și implementarea algoritmului - încep să se bată cap în cap. Uneori avem la dispoziție un algoritm foarte puternic, dar nu știm cum s-ar putea implementa, alteori acest algoritm nu face față volumului maxim de date de intrare, iar alteori ne dăm seama că am putea foarte ușor să scriem un program, dar nu suntem în stare să demonstrăm că el ar merge perfect. Foarte des se renunță la implementarea algoritmilor de complexitate optimă, care sunt alambicați și constituie adevărate focare de „bug"-uri, preferându-se un algoritm mai lent dar care să se poată implementa rapid și fără dureri de cap. Mulți olimpici pierd clasa a IX-a, poate chiar și pe a X-a descoperind aceste lucruri. Cartea de față își propune să le mai ușureze drumul.

S-a presupus cunoscut limbajul de programare Pascal, cu toate instrucțiunile și procedurile sale standard. În carte există multe surse în limbajul C standard. Am preferat acest lucru, deși la concursuri se recomandă programarea în Pascal, pentru că am sperat că un elev familiarizat cu limbajul Pascal va citi fără dificultate o sursă C și pentru că am dorit ca această carte să fie și un exercițiu de C, al cărui număr de utilizatori la nivelul liceului este destul de redus. Surse în Pascal există numai acolo unde se urmărește punerea în evidență a unei anumite subtilități a limbajului.

Tehnicile de programare s-au presupus cunoscute în esență, astfel încât am trecut direct la unele optimizări, la exemple de folosire a lor și la compararea lor, respectiv la prezentarea unor criterii în funcție de care să optăm pentru folosirea fiecăreia. De asemenea, am renunțat la definirea termenilor de graf, arbore, vector, matrice, listă, stivă și a tuturor celorlalte structuri de date de bază. Am considerat interesantă prezentarea pe larg numai a *heap*-urilor și a tabelelor de dispersie (*hash*), care sunt mai rar folosite și de aceea mai puțin cunoscute. În sfârșit, am presupus cunoscută noțiunea de complexitate a unui algoritm, deoarece în toate problemele se face calculul complexității.

Cartea prezintă interes și pentru problemele pe care le cuprinde. Ele nu sunt banale (de fapt, majoritatea au avut onoarea de a da bătăi de cap concurenților la olimpiade) și pot fi lucrate acasă de către elevi pentru menținerea în formă. Tocmai de aceea, am urmărit ca, ori de câte ori am propus o problemă spre rezolvare, enunțul să fie dat în aceeași formă pe care ar fi avut-o la un concurs: clar, detaliat, cu specificarea formatului datelor de intrare și ieșire și cu un exemplu sau două. Singura diferență este că, de regulă, la concursuri și olimpiade se precizează numai timpul limită admis pentru un test; în carte am considerat folositor să se specifice și o complexitate optimă a algoritmului care rezolvă problema, deoarece timpul de execuție variază în funcție de resursele calculatorului și de

limbajul de programare folosit, deci e un criteriu mai puțin semnificativ. Timpul destinat implementării unei probleme este în general egal cu cel care s-a acordat la concursul unde a fost propusă respectiva problemă.

În sfârșit, consider că programatorii care își propun să scrie aplicații de mari dimensiuni ar avea destul de multe lucruri de învățat din acest volum, deoarece am inclus și detalii privind structuri de date mai neobișnuite sau gestionarea economică a memoriei.

Sper să nu închideți această carte cu sentimentul că mai bine n-ați fi deschis-o.

Cătălin Frâncu

0.1 Problema 15

Problema următoare a fost propusă la proba de baraj de la Olimpiada Națională de Informatică, Slatina 1995 și este un exemplu tipic de aplicare a principiului lui Dirichlet.

ENUNȚ: La un SHOP din Slatina se găsesc spre vînzare $P-1$ (P este un număr prim) produse unicat de costuri $X(1), X(2), \ldots, X(P-1)$. Nici unul din produse nu poate fi cumpărat prin plata exactă cu bancnote de P\$. În SHOP intră un olimpic care are un număr nelimitat de bancnote de P\$ și o singură bancnotă de Q\$ ($1 \leq Q \leq P-1$). Ce produse trebuie să cumpere olimpicul pentru a putea plăti exact produsele cumpărate?

Intrarea: Datele de intrare se dau în fișierul de intrare `INPUT.TXT` ce conține două linii:

- pe prima linie valorile lui P și Q;
- pe a doua linie valorile costurilor produselor.

Ieșirea se va face în fișierul `OUTPUT.TXT` unde se vor lista în ordine crescătoare indicii produselor cumpărate de olimpic.

INPUT.TXT	OUTPUT.TXT
5 4 1 3 6 7	1 2

Timp de implementare: la Slatina s-au acordat cam 90 de minute, dar 45 ar trebui să fie suficiente.

Timp de rulare pentru fiecare test: 1 sec.

Complexitate cerută: $O(P)$.

Enunțul original nu specifica nici o limită maximă pentru valoarea lui P. Vom adăuga noi această limită, respectiv $P < 10.000$.

REZOLVARE: Problema se reduce la a găsi un grup de obiecte pentru care suma costurilor să fie divizibilă cu P sau să fie congruentă cu Q modulo P. Am văzut deja în problema precedentă că dispunem de o soluție $O(N^2)$ pentru a găsi un număr de elemente care să se dividă cu P. În cazul nostru, trebuie să observăm însă că nu avem P obiecte, ci numai $P-1$; în schimb, dispunem de o bancnotă suplimentară de valoare Q. Aceste diferențe vor fi explicate mai târziu și se va vedea că ele nu schimbă cu nimic natura problemei. Diferența esențială provine din faptul că nu se mai cere ca suma numerelor să fie maximă, ca în problema precedentă. Orice combinație de numere care dau o sumă potrivită este suficientă.

Să începem prin a explica principiul lui Dirichlet, care de altfel face apel numai la intuiție și nu necesită cunoștințe speciale de matematică. Acest principiu spune că dacă distribuim N obiecte în K cutii, atunci cel puțin într-o cutie se vor afla minim $\lceil N/K \rceil$ obiecte (aici prin $\lceil N/K \rceil$ se înțelege „cel mai mic întreg mai mare sau egal cu N/K”). Demonstrația se face prin reducere la absurd: dacă în fiecare cutie s-ar afla mai puțin decât $\lceil N/K \rceil$ obiecte, atunci numărul total de obiecte ar fi mai mic decât $K \times \lceil N/K \rceil$, adică mai mic decât N.

Spre exemplu, oricum am distribui 7 obiecte în 4 cutii, putem fi siguri că cel puțin într-o cutie se vor afla minim $\lceil 7/4 \rceil = 2$ obiecte. Într-adevăr, dacă toate cutiile ar conține cel mult câte un obiect, atunci numărul total de obiecte nu ar putea fi mai mare ca 4, ceea ce este absurd.

Să vedem acum cum s-ar putea aplica acest principiu la problema de față. Să facem notația

$$S(k) = \sum_{i=1}^{k} X(i) \tag{1}$$

și convenția $S(0) = 0$. Prin urmare, putem scrie egalitatea

$$S(k_2) - S(k_1) = \sum_{i=k_1+1}^{k_2} X(i) \tag{2}$$

Dacă găsim în vectorul S două valori $S(k_1)$ și $S(k_2)$ care dau același rest la împărțirea prin P, înseamnă că diferența lor se divide cu P, deci șirul de obiecte $k_1+1, k_1+2, \ldots, k_2$

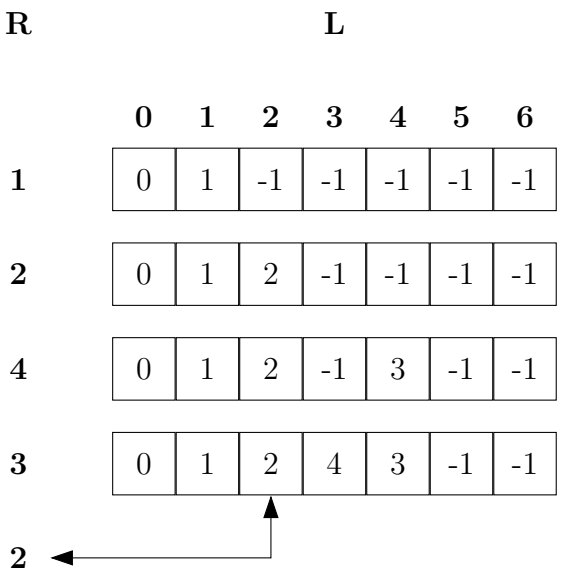
poate constitui o soluție.

Să presupunem pentru început că dispunem de P obiecte. Se pune întrebarea: ce valori poate lua restul împărțirii lui $S(k)$ prin P? Desigur, orice valoare între 0 și $P-1$. Există deci în total P resturi distincte. Pe de altă parte, există $P+1$ elemente în vectorul S (se consideră și elementul $S(0)$). Începem să recunoaștem aici principiul lui Dirichlet, în care „obiectele" sunt resturile $S(0) \bmod P, S(1) \bmod P, \ldots, S(P) \bmod P$, iar cutiile sunt clasele de resturi modulo P. Avem de distribuit P+1 obiecte în P cutii, așadar cel puțin într-o cutie se vor afla $\lceil (P+1)/P \rceil = 2$ obiecte. Prin urmare, vor exista cu siguranță doi indici $k_1 < k_2$ astfel încât $S(k_2) - S(k_1) \equiv 0 \pmod{P}$. Nu avem decât să tipărim secvența $k_1 + 1, k_1 + 2, \ldots, k_2$.

Să vedem acum ce se întâmplă dacă avem numai $P-1$ obiecte, așa cum este cazul problemei. Atunci avem numai P resturi posibile, deci se poate ca toate elementele din S să dea resturi distincte la împărțirea prin P. Dar în acest caz, există un indice k astfel încât $S(k) \equiv Q \pmod{N}$, deci trebuie doar să tipărim secvența de indici $1, 2, \ldots, k$.

Pentru a reuni aceste două cazuri într-unul singur, putem considera expresia $S(k)$ drept un alt mod de a scrie expresia $S(k) - S(0)$. Problema se reduce la a căuta doi indici $k_1, k_2 \in \{0, 1, 2, \ldots, P\}$ astfel încât $(S(k_2) - S(k_1)) \bmod N \in [0, Q]$. Să nu uităm că trebuie să efectuăm această operație într-un timp liniar, deci nu avem voie să comparăm pur și simplu două câte două elementele vectorului S. Vom prezenta modul în care se pot găsi două elemente congruente modulo P, cazul celălalt tratându-se analog. Metoda constă în crearea unui alt vector, L, în care $L(i) = j$ înseamnă că suma $S(j)$ dă restul i la împărțirea prin P. Inițial, toate elementele vectorului L vor avea o valoare specială, eventual negativă. Apoi se parcurge vectorul S și pentru fiecare $S(j)$ se efectuează operația $L(S(j) \bmod P) \leftarrow j$. În momentul în care se încearcă reatribuirea unui element din L care are deja o valoare dată, înseamnă că am găsit cei doi indici pe care îi căutam.

Iată un exemplu. Dacă $P = 7$ și $X = (8, 8, 2, 6, 13, 3)$, rezultă vectorul $S = (8, 16, 18, 24, 37, 40)$. Resturile la împărțirea prin 7 sunt respectiv 1, 2, 4, 3, 2 și 5.



După cum se vede, restul 2 poate fi obținut atât cu primele două obiecte, cât și cu primele 5, deci suma prețurilor obiectelor 3, 4 și 5 este divizibilă cu 7.

Un ultim detaliu de implementare constă în aceea că nu este necesară memorarea vectorului S, ci numai a elementului curent; orice alte informații care ne trebuie la un moment dat le putem afla din vectorii X și L. Pentru memorarea elementului curent din S, se pornește cu valoarea 0 și la fiecare pas se adaugă valoarea elementului corespunzător din X.

```
#include <stdio.h>
#define NMax 10000
#define None -1

int X[NMax], L[NMax], P, Q;

void ReadData(void)
{ FILE *F = fopen("input.txt", "rt");
  int i;

  fscanf(F, "%d %d\n", &P, &Q);
  for (i=1; i<P; fscanf(F, "%d", &X[i++]));
  fclose(F);
}

void FindSum(void)
{ long S=0;
  FILE *F = fopen("output.txt", "wt");
  int i,j;
```

```
20
21    for (i=1, L[0]=0; i<P; L[i++] = None);
22    i=0;
23    while ( L[ (S+=X[++i]) % P ]==None && // Restul 0
24            L[ (S%P+P-Q) % P ]==None )    // Restul Q
25      L[S%P]=i;
26    for (j = 1 + ((L[S%P]!=None)? L[S%P]: L[ (S%P+P-Q) % P ]);
27         j <= i; fprintf(F, "%d␣", j++));
28    fclose(F);
29  }
30
31  void main(void)
32  {
33    ReadData();
34    FindSum();
35  }
```

0.2 Problema 16

Următoarele probleme aparțin categoriei de probleme pe care, dacă ne grăbim, le putem clasifica drept „ușoare". Într-adevăr, ele au soluții vizibile și foarte la îndemână, dar și soluții mai subtile și mult mai performante. Pentru a obliga cititorul să se gândească și la aceste soluții, am ales limite pentru datele de intrare suficient de mari încât să facă nepractice rezolvările „la minut".

ENUNȚ: (Generarea unui arbore oarecare când i se cunosc gradele) Se dă un vector cu N numere întregi. Se cere să se construiască un arbore cu N noduri numerotate de la 1 la N astfel încât gradele celor N noduri să fie exact numerele din vector. Dacă acest lucru nu este posibil, se va da un mesaj de eroare corespunzător.

Intrarea: Datele de intrare se află în fișierul `INPUT.TXT`. Pe prima linie se află numărul de noduri N ($N \leq 10.000$), iar pe a doua linie se află cele N numere separate prin spații. Toate numerele sunt strict pozitive și mai mici ca 10.000.

Ieșirea se va face în fișierul `OUTPUT.TXT`. Dacă problema are soluție, se va tipări arborele prin muchiile lui. Fiecare muchie se va lista pe câte o linie, prin nodurile adiacente separate printr-un spațiu. Dacă problema nu are soluție, se va afișa un mesaj corespunzător.

INPUT.TXT	**OUTPUT.TXT**
6 1 2 3 2 1 1	1 4 2 5 3 6 4 6 5 6
3 2 2 1	Problema nu are solutie!

Timp de implementare: 30 minute - 45 minute.

Timp de rulare: 2-3 secunde.

Complexitate cerută: $O(N \log N)$; dacă vectorul citit la intrare se presupune sortat, se cere o complexitate $O(N)$.

REZOLVARE: Să începem prin a ne pune întrebarea: când are problema soluție și când nu?

Se știe că un arbore oarecare cu N noduri are $N-1$ muchii. Fiecare din aceste muchii va contribui cu o unitate la gradele nodurilor adiacente. Deducem de aici că suma gradelor tuturor nodurilor este egală cu dublul numărului de muchii, adică, notând cu $G[1], G[2], \ldots, G[N]$ gradele nodurilor,

$$\sum_{i=1}^{N} G[i] = 2 \cdot (N-1) \tag{3}$$

Am aflat deci o condiție necesară pentru ca problema să aibă soluție. O a doua condiție este ca toate nodurile să aibă grade cuprinse între 1 și $N-1$. Totuși, ținând cont de afirmația enunțului că toate numerele din vector sunt strict pozitive, rezultă că a doua condiție nu mai trebuie verificată. Iată de ce: să presupunem că am verificat prima condiție și am constatat că suma celor N numere este $2(N-1)$, iar unul dintre numere este cel puțin N. Atunci ar rezulta că suma celorlalte $N-1$ numere este cel mult $N-2$, de unde rezultă că există cel puțin un nod de grad 0, ceea ce contrazice informația din enunț. Prin urmare, numai prima condiție este importantă, cea de-a doua fiind redundantă.

Vom demonstra că această condiție este și suficientă indicând efectiv modul de construcție a arborelui în cazul în care ea este satisfăcută. Începem prin a sorta vectorul de numere. Acest lucru era oricum de așteptat, deoarece complexitatea $N \log N$ ne-o permite. Trebuie numai să avem grijă să alegem un algoritm de sortare de com-

plexitate $N \log N$. Programul care urmează folosește heapsort-ul. Odată ce am sortat vectorul, trebuie să reconstituim muchiile în timp liniar, și iată cum:

- Se poate demonstra că primele două elemente din vectorul sortat au valoarea 1. Într-adevăr, dacă toate elementele ar fi mai mari sau egale cu 2, atunci suma lor ar fi mai mare sau egală cu $2N$, ori noi știm că suma trebuie să fie $2N-2$, adică există cel puțin două elemente egale cu 1 în vector. Acest lucru rezultă imediat dacă ne gândim că orice arbore are cel puțin două frunze.

- Vom căuta în vector primul număr mai mare sau egal cu 2. Se pune întrebarea: există întotdeauna acest număr? Nu cumva există un arbore în care toate nodurile au grad 1? Să aplicăm condiția precedentă și să vedem ce se întâmplă. Dacă toate nodurile au grad 1, atunci suma gradelor este N, ceea ce conduce la ecuația:

$$N = 2 \cdot (N-1) \implies N = 2 \tag{4}$$

- Iată deci că există un singur arbore în care toate nodurile sunt frunze, anume cel cu 2 noduri unite printr-o muchie. Vom reveni mai târziu la acest caz particular. Deocamdată presupunem că există în vector un număr mai mare ca 1, pe poziția K în vector. Atunci vom uni nodul 1 din arbore (care știm că are gradul 1) cu nodul K. În felul acesta, nodul 1 și-a completat numărul necesar de vecini și poate fi neglijat pe viitor, iar $G[K]$ va fi decrementat cu o unitate, întrucât nodul K și-a completat unul din vecini. Astfel, problema s-a redus la un arbore cu $N-1$ noduri numerotate de la 2 la N.

- Vectorul G este în continuare sortat, deoarece $G[K-1] = 1 < G[K] \leq G[K+1]$ înainte de decrementarea lui $G[K]$, deci după decrementare vom avea $G[K-1] = 1 \leq G[K] < G[K+1]$, adică dubla relație de ordonare se păstrează.

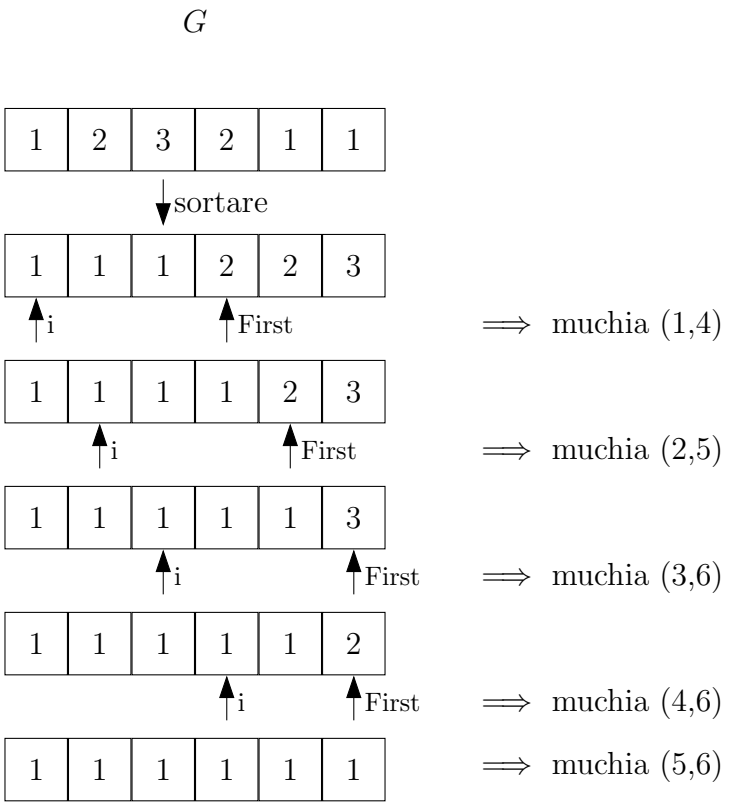
- Întrucât secvența $G[2], G[3], \ldots, G[N]$ reprezintă gradele unui arbore, putem aplica același raționament ca mai înainte pentru a deduce că $G[2] = 1$. Cu ce nod vom uni nodul 2? Dacă $G[K] > 1$, îl vom uni cu nodul K. Dacă prin decrementare, $G[K]$ a ajuns la valoarea 1, vom trece la nodul $K+1$ (despre care știm că are gradul mai mare ca 1) și vom trasa muchia $2 \leftrightarrow (K+1)$.

- Procedeul acesta se repetă până când au fost trasate $N-2$ muchii. Aceasta înseamnă că a mai rămas o singură muchie de trasat. Iată deci că, mai devreme sau mai târziu, este oricum inevitabil să ajungem la cazul particular de arbore de care am amintit mai devreme. Deoarece la primul pas am unit nodul 1 cu nodul K, la al

doilea pas am unit nodul 2 cu un alt nod (K sau $K+1$) ș.a.m.d., rezultă că în $N-2$ iterații, toate nodurile de la 1 la $N-2$ și-au completat numărul de vecini. De aici rezultă că ultima muchie pe care o vom trasa este $(N-1) \leftrightarrow N$; putem să tipărim această muchie „cu ochii închiși", fără nici un fel de teste suplimentare. Ultima muchie trasată este diferită de celelalte și necesită o operație separată de trasare din cauză că, în timp ce primele $N-2$ iterații uneau o frunză cu un nod intern, această ultimă iterație are de unit două frunze, deci nu are sens să mai căutăm un nod de grad mai mare ca 1.

Aceasta este metoda de lucru. Calculul complexității este simplu: Avem nevoie doar de doi indici: Unul care marchează frunza curentă (în program el se numește pur și simplu `i`) și care avansează la fiecare pas, și unul care marchează primul număr mai mare ca 1 din vector (în program se numește `First`) și care se incrementează cu cel mult 1 la fiecare pas (deci de mai puțin de N ori în total). De aici rezultă complexitatea liniară a algoritmului.

Să vedem cum se comportă această metodă pe cazul particular al exemplului 1:



Mai trebuie remarcat că soluția nu este unică. Propunem ca temă cititorului să scrie un program care să verifice în timp $O(N \log N)$ dacă soluția furnizată de un alt program este corectă.

```
#include <stdio.h>
int G[10001], N;
long Sum;
FILE *OutF;

void ReadData(void)
{ FILE *F=fopen("input.txt","rt");
  int i;

  fscanf(F,"%d\n",&N);
  for (i=1, Sum=0; i<=N; i++)
    { fscanf(F,"%d",&G[i]);
      Sum+=G[i];
    }
  fclose(F);
}

void Sift(int K, int N)
/* Cerne al K-lea element dintr-un heap de N elemente */
{ int Son;

  /* Alege un fiu mai mare ca tatal */
  if (K<<1<=N)
    { Son=K<<1;
      if (K<<1<N && G[(K<<1)+1]>G[(K<<1)])
        Son++;
      if (G[Son]<=G[K]) Son=0;
    }
    else Son=0;
  while (Son)
    { /* Schimba G[K] cu G[Son] */
      G[K]=(G[K]^G[Son])^(G[Son]=G[K]);
      K=Son;
      /* Alege un alt fiu */
      if (K<<1<=N)
        { Son=K<<1;
          if (K<<1<N && G[(K<<1)+1]>G[(K<<1)])
            Son++;
          if (G[Son]<=G[K]) Son=0;
        }
        else Son=0;
    }
}

void HeapSort(void)
```

```
46 { int i;
47
48   /* Construieste heap-ul */
49   for (i=N>>1; i;) Sift(i--,N);
50   /* Sorteaza vectorul */
51   for (i=N; i>=2;)
52     { G[1]=(G[1]^G[i])^(G[i]=G[1]);
53       Sift(1,--i);
54     }
55 }
56
57 void Match(void)
58 { int i, First=1;
59
60   while (G[First]==1 && First<N) First++;
61   /* Trebuie adaugata si conditia First<N
62      pentru a acoperi cazul particular N=2 */
63   for (i=1; i<=N-2; i++)
64     { fprintf(OutF,"%d␣%d\n", i, First);
65       First+=(--G[First]==1);
66     }
67   fprintf(OutF,"%d␣%d\n", N-1, N);
68 }
69
70 void main(void)
71 {
72   ReadData();
73   OutF=fopen("output.txt","wt");
74   if (Sum==(N-1)<<1)
75     { HeapSort();
76       Match();
77     }
78     else fputs("Problema␣nu␣are␣solutie!\n",OutF);
79   fclose(OutF);
80 }
```

Cuprins