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 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,

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

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:

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

- 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.

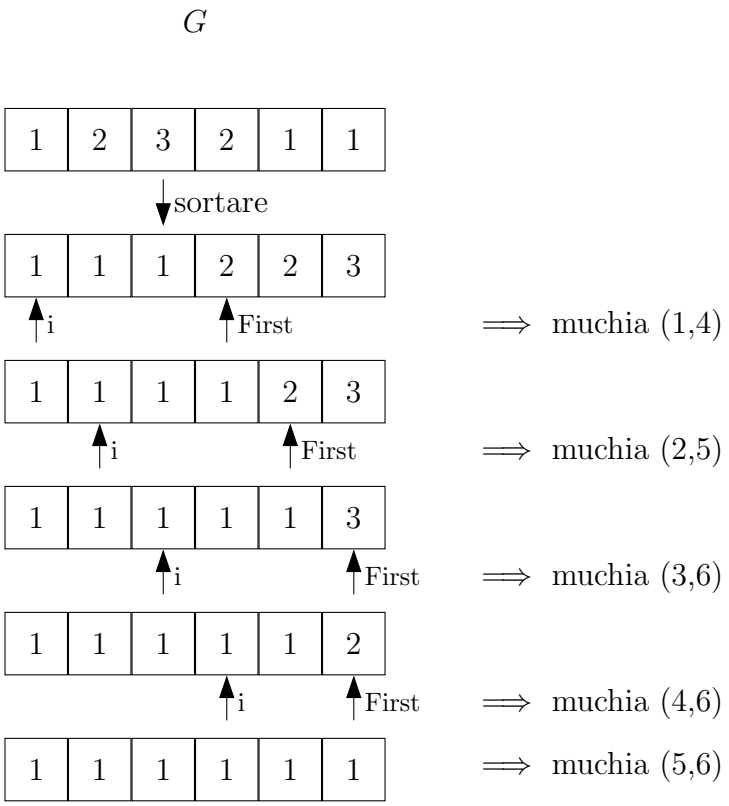
- Întrucât secvența $G[2], G[3], \ldots, G[N]$ reprezintă gradele unui arbore, putem aplica același raționament ca mai înainte pentru a deduce că $G[2] = 1$. Cu ce nod vom uni nodul 2? Dacă $G[K] > 1$, îl vom uni cu nodul K. Dacă prin decrementare, $G[K]$ a ajuns la valoarea 1, vom trece la nodul $K+1$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

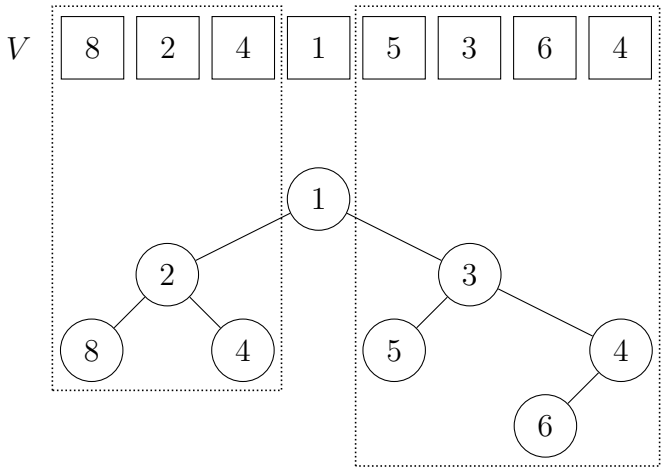
```
#include <stdio.h>
int G[10001], N;
long Sum;
FILE *OutF;

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)
```

```
{ int i;

  /* Construieste heap-ul */
  for (i=N>>1; i;) Sift(i--,N);
  /* Sorteaza vectorul */
  for (i=N; i>=2;)
    { G[1]=(G[1]^G[i])^(G[i]=G[1]);
      Sift(1,--i);
    }
}

void Match(void)
{ int i, First=1;

  while (G[First]==1 && First<N) First++;
  /* Trebuie adaugata si conditia First<N
     pentru a acoperi cazul particular N=2 */
  for (i=1; i<=N-2; i++)
    { fprintf(OutF,"%d %d\n", i, First);
      First+=(--G[First]==1);
    }
  fprintf(OutF,"%d %d\n", N-1, N);
}

void main(void)
{
  ReadData();
  OutF=fopen("output.txt","wt");
  if (Sum==(N-1)<<1)
    { HeapSort();
      Match();
    }
    else fputs("Problema nu are solutie!\n",OutF);
  fclose(OutF);
}
```

0.2 Problema 17

Iată un nou exemplu de problemă care admite două rezolvări: una evidentă, dar neeficientă și una mai puțin evidentă, dar cu mult mai eficientă.

ENUNȚ: Fie V un vector. Arborele cartezian atașat vectorului V este un arbore binar care se obține astfel: Dacă vectorul V este vid (are 0 elemente), atunci arborele cartezian atașat lui este de asemenea vid. Altfel, se selectează elementul de valoare minimă din vector și se pune în rădăcina arborelui, iar arborii cartezieni atașați fragmentelor de vector din stânga (respectiv din dreapta) elementului minim se pun în subarborele stâng, respectiv drept al rădăcinii. Iată, de exemplu, care este arborele cartezian al următorului vector cu 8 elemente:



În figură au fost încadrate prin dreptunghiuri punctate porțiunile din stânga, respectiv din dreapta elementului minim, împreună cu subarborii atașați. Trebuie observat că arborele cartezian atașat unui vector poate să nu fie unic, în cazul în care există mai multe elemente de valoare minimă. Vom impune ca o condiție suplimentară ca elementul care va fi trecut în rădăcină să fie cel mai din stânga dintre minime (cel cu indicele cel mai mic). Astfel, arborele cartezian este unic.

Cerința problemei este ca, dându-se un vector, să i se construiască arborele cartezian.

Intrarea: Fișierul de intrare `INPUT.TXT` conține pe prima linie valoarea lui N ($N \leq 10.000$), iar pe a doua N numere naturale mai mici ca 30.000, separate prin spații.

Ieșirea se va face în fișierul text `OUTPUT.TXT` sub următoarea formă:

$T_1 \quad T_2 \quad T_3 \quad \ldots \quad T_N$

unde T_i este indicele în vector al elementului care este părintele lui $V[i]$ în arborele cartezian. Dacă $V[i]$ este rădăcina arborelui, atunci $T_i = 0$.

Exemplu: Pentru exemplul dat mai sus, fișierul `INPUT.TXT` este:

```
8
8 2 4 1 5 3 6 4
```

După cum reiese din figură, tatăl elementului 8 este elementul 2, adică al doilea în vector; tatăl elementului 2 este elementul 1, adică al 4-lea în vector; tatăl elementului 5 este elementul 3, adică al 6-lea în vector ș.a.m.d. Fișierul de ieșire este deci:

```
2 4 2 0 6 4 8 6
```

Complexitate cerută: $O(N)$.

Timp de implementare: 45 minute - 1h.

Timp de rulare: 2 secunde.

REZOLVARE: Nu întâmplător s-a impus o complexitate liniară pentru rezolvarea acestei probleme. Altfel, ea ar fi trivială în $O(N^2)$, prin următoarea metodă: scriem o procedură care parcurge vectorul și caută minimul, apoi se reapelează pentru bucățile de vector aflate în stânga, respectiv în dreapta minimului. Pentru a demonstra că această variantă de rezolvare are complexitate pătratică, să ne imaginăm cum s-ar comporta ea pe cazul:

$$V = (N \quad N-1 \quad N-2 \quad \dots \quad 2 \quad 1) \tag{3}$$

La primul apel, procedura ar face N comparații pentru a parcurge vectorul (deoarece elementul minim este ultimul în vector) și s-ar reapela pentru porțiunea din vector care cuprinde primele $N-1$ elemente. La al doilea apel, ar face $N-1$ comparații și s-ar reapela pentru primele $N-2$ elemente etc. În concluzie, numărul total de comparații făcute este

$$N + (N-1) + (N-2) + \cdots + 1 = \frac{N(N+1)}{2} \tag{4}$$

de unde rezultă complexitatea. O problemă interesantă, pe care îi vom lăsa plăcerea cititorului să o rezolve, este de a demonstra că această versiune **nu poate atinge o complexitate mai bună decât** $O(N \log N)$ și de a arăta care sunt cazurile cele mai favorabile pe care se obține această complexitate.

A doua metodă este și ea destul de ușor de înțeles și de implementat. Ceea este mai greu de acceptat este că ea are complexitate liniară, așa cum vom încerca să explicăm la sfârșit. Iată mai întâi principiul de rezolvare: vom porni cu un arbore cartezian vid și, la fiecare pas, vom adăuga câte un element al vectorului V la acest arbore, astfel încât structura obținută să rămână un arbore cartezian. La al k-lea pas, vom adăuga

elementul $V[k]$ în arbore și vom restructura arborele în așa fel încât să obținem arborele cartezian atașat primelor k elemente din V. Trebuie să ne concentrăm atenția asupra a două lucruri:

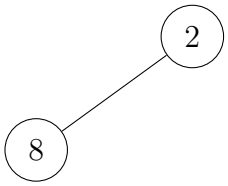
1. Cunoscând arborele cartezian atașat primelor k-1 vârfuri și elementul $V[k]$, cum se obține arborele cartezian atașat primelor k vârfuri?
2. Cum reușim să actualizăm de N ori arborele astfel încât timpul total consumat să fie liniar?

Pentru a răspunde la prima întrebare, pe lângă vectorii V și T, mai este necesară o stivă S, în care vom stoca elemente ale vectorului V. Inițial, stiva este vidă. Atunci când un nou element X sosește, el va fi introdus în stivă imediat după ultimul număr din stivă care are o valoare mai mică sau egală cu X. Toate elementele care se aflau înainte în stivă pe poziții mai mari sau egale cu poziția pe care a fost inserat X vor fi eliminate din stivă, iar elementul care se afla exact pe poziția lui X va deveni fiul stâng al lui X. X însuși va deveni fiul drept al predecesorului său în stivă. La fiecare moment, primul element din stivă este rădăcina arborelui cartezian.

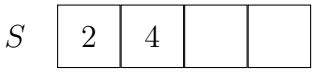
Pentru a înțelege mai bine principiul de funcționare a stivei, să analizăm mai de aproape exemplul din enunț.

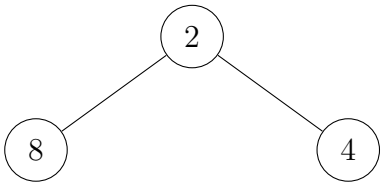
La început stiva este vidă. Primul element din V are valoarea 8, drept care îl vom pune în stivă, iar arborele cartezian va avea un singur nod:

Următorul element sosit este 2. Acesta este mai mic decât 8, deci trebuie introdus înaintea lui în stivă. El va fi deci primul element din stivă și rădăcina arborelui cartezian la acest moment. Concomitent, 8 va fi eliminat din stivă și va deveni fiul stâng al lui 2:

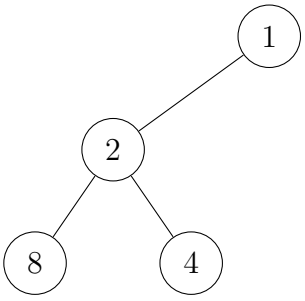


Se observă că arborele obținut este tocmai arborele cartezian atașat secvenței $(V[1], V[2])$. Următorul element este 4, care este mai mare decât 2, deci trebuie adăugat în vârful stivei. Nici un element nu este eliminat din stivă, iar 4 devine fiul drept al lui 2:

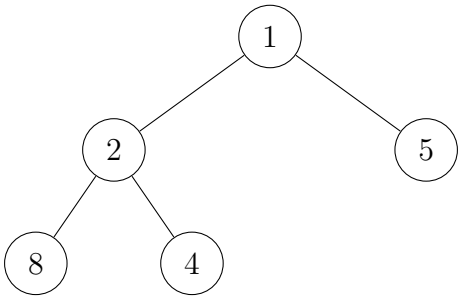




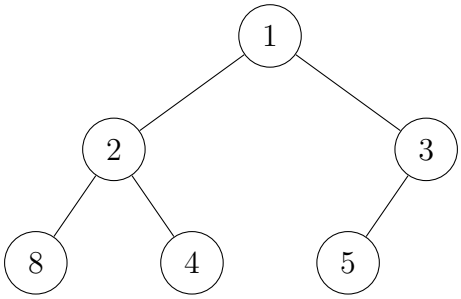
Următorul element sosit este 1, care este mai mic decât toate numerele din stivă. Stiva se va goli, iar numărul 2 (cel peste care se va scrie 1) va deveni fiul stâng al lui 1:



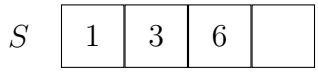
Deja arborele începe să semene cu forma sa finală. Urmează elementul 5, care va fi adăugat în stivă și „atârnat” în dreapta lui 1:

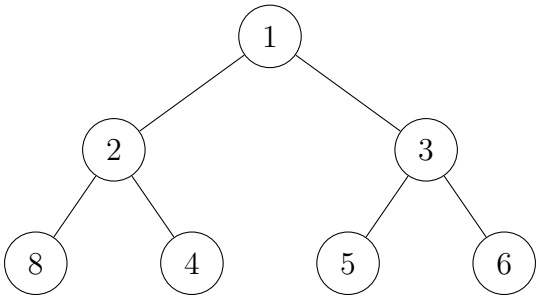


Elementul 3 este mai mare ca 1, căruia îi va deveni fiu drept, dar mai mic ca 5, pe care îl va elimina din stivă:

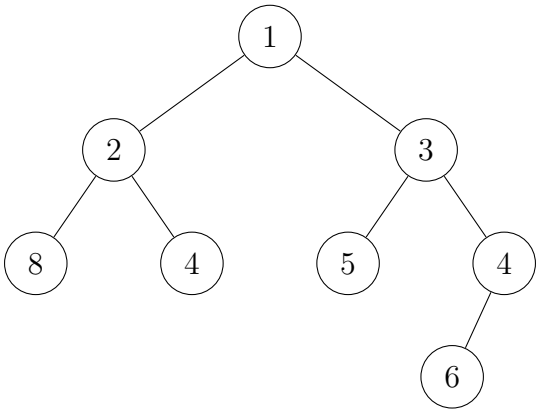


Următorul număr, 6, va fi adăugat la extremitatea dreaptă a arborelui și în vârful stivei:





În sfârșit, elementul 4 va fi fiul drept al lui 3 și îl va elimina din stivă pe 6, care îi va deveni fiu stâng:



Se observă că arborele a ajuns tocmai la forma sa corectă. Trebuie acum să ne ocupăm de un detaliu de implementare. Pentru a afla poziția pe care trebuie inserat un element în stivă avem două metode:

1. Putem să căutăm în stivă de la dreapta la stânga (ar fi mai corect spus „de la vârf spre bază") până dăm de un element mai mic decât cel de inserat; programul folosește această metodă și îi vom discuta în final eficiența.

2. Putem face o căutare binară în stivă, întrucât elementele din stivă au valori crescătoare de la bază spre vârf (lăsăm demonstrația acestei afirmații în seama cititorului). O căutare binară într-un vector de k elemente poate necesita, în cazul cel mai nefavorabil, $\log k$ comparații. În cazul cel mai nefavorabil, când vectorul V este sortat crescător, elementele vor fi introduse pe rând în stivă și nu vor mai fi scoase, deci la fiecare pas se vor face $\log k$ comparații, unde k ia valori de la 1 la N. Complexitatea care rezultă este mai slabă decât cea cerută:

$$\begin{aligned} O(\log 1 + \log 2 + \cdots + \log N) = O(\sum_{k=1}^{N} \log k) = \\ = O(\log \prod_{k=1}^{N} k) = O(\log N!) = O(N \cdot \log N) \end{aligned} \tag{5}$$

Acesta este unul din puținele cazuri în care căutarea binară este mai ineficientă decât cea secvențială.

Pentru ușurința programării, sursa C de mai jos reține în stiva S nu valorile elementelor, ci indicii lor în vectorul V (deoarece aceștia sunt ceruți pentru construcția vectorului T).

```
#include <stdio.h>
#define NMax 10001

int V[NMax], /* Vectorul */
    T[NMax], /* Vectorul de tati */
    S[NMax], /* Stiva */
    N;       /* Numarul de elemente */

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

  fscanf(F,"%d\n",&N);
  for (i=1; i<=N;)
    fscanf(F, "%d", &V[i++]);
}

void BuildTree(void)
{ int i,k,LenS=0;

  S[0]=0; /* Pentru ca initial T[1] sa fie 0 */
  for (i=1; i<=N; i++)
    { /* Cauta pozitia pe care va fi inserat V[i] */
      k=LenS+1;
      while (V[S[k-1]]>V[i]) k--;
      /* Face corecturile in S si T */
      T[i]=S[k-1];
      if (k<=LenS) T[S[k]]=i;
      /* i este ultimul element din stiva, deci... */
      S[LenS=k]=i;
    }
}

void WriteSolution(void)
{ FILE *F=fopen("output.txt","wt");
  int i;
  for (i=1; i<=N;)
    fprintf(F,"%d␣",T[i++]);
  fprintf(F,"\n");
}

```

```
42 void main(void)
43 {
44   ReadData();
45   BuildTree();
46   WriteSolution();
47 }
```

Acum să analizăm și complexitatea acestui algoritm. În primul rând, ea nu poate fi mai bună decât $O(N)$, pentru că aceasta este complexitatea funcțiilor de intrare și ieșire. Procedura `BuildTree` se compune dintr-un ciclu `for` în care se execută patru operații în timp constant și o instrucțiune repetitivă `while`. Numărul total de operații în timp constant care se execută în procedură este prin urmare $O(N)$. Problema este: care este numărul total maxim de evaluări ale condiției logice din bucla `while`? Aparent, bucla while se execută de $O(N)$ ori, deci numărul total de evaluări ar fi $O(N^2)$. Să aruncăm totuși o privire mai atentă.

Fiecare evaluare a condiției din bucla `while` are ca efect decrementarea lui k și, implicit, eliminarea unui element deja existent în stivă. Pe de altă parte, fiecare element este introdus în stivă o singură dată și deci nu poate fi eliminat din stivă decât cel mult o dată. Așadar numărul maxim de elemente ce pot fi eliminate din stivă pe parcursul executării procedurii `BuildTree` este $N-1$, deci numărul total de evaluări ale condiției este $O(N)$. De aici rezultă că programul are complexitate liniară.

Cuprins