# 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 adus formatul la zi și am republicat cartea sub o licență liberă. 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, 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

# Capitolul 1

# Concursul de informatică

## - de la extaz la agonie -

> Cine dorește să-și rezolve treburile la vremea potrivită, să-și împartă cu atenție timpul.
>
> (Plaut)

Experiența demonstrează că, oricât de mare ar fi bagajul de cunoștințe acumulat de un elev, mai e nevoie de *ceva* pentru a-i asigura succesul la olimpiada de informatică. Aceasta deoarece în timp de concurs lucrurile stau cu totul altfel decât în fața calculatorului de acasă sau de la școală. Reușita depinde, desigur, în cea mai mare măsură de puterea fiecăruia de a pune în practică ceea ce a învățat acasă. Numai că în acest proces intervin o serie de factori care țin de temperament, de experiența individuală, de numărul de ore dormite în noaptea dinaintea concursului (care în taberele naționale este îngrijorător de mic) și așa mai departe.

Cu riscul de a cădea în demagogie, trebuie să spunem că un concurs de informatică presupune mult mai mult decât un simplu act de prezență la locul desfășurării ostilităților. Capitolul de față încearcă să enunțe câteva principii ale concursului, pe care autorul și le-a însușit în cei patru ani de liceu, atât din experiența proprie, cât și învățând de la alții. Cititorul este liber să respingă aceste sfaturi sau să le accepte, filtrându-le prin prisma personalității sale și alegând ceea ce i se potrivește.

## 1.1 Înainte de concurs

Primul și cel mai de seamă lucru pe care trebuie să îl știți este că e important și să participi, dar e și mai important să participi onorabil, iar dacă se poate să și câștigi. Nu trebuie să porniți la drum cu îngâmfare; modestia e bună, dar nu trebuie în nici un caz să ducă la neîncredere în sine. Fiecare trebuie să știe clar de ce e în stare și, mai presus de toate, să se gândească că la urma urmei nu dificultatea concursului contează, căci concursul, greu sau ușor, este același pentru toți. Mult mai importantă este valoarea individuală și nu în ultimul rând pregătirea psihologică.

Autorul a fost peste măsură de surprins să constate că mulți elevi merg la concurs fără ceas și fără hârtie de scris. Aceasta este fără îndoială o greșeală capitală. În timpul concursului trebuie ținută o evidență drastică a timpului scurs și a celui rămas. E drept că în general supraveghetorii anunță din când în când timpul care a trecut, dar e bine să nu vă bazați pe nimeni și nimic altceva decât pe voi înșivă. Unii colegi spuneau „Ei, ce nevoie am de ceas, oricum am ceasul calculatorului la îndemână". Așa e, dar e extrem de incomod să te oprești mereu la jumătatea unei idei, să deschizi o sesiune DOS din cadrul limbajului de programare și să afli cât e ceasul.

În ceea ce privește hârtia de scris, ea este în mod sigur necesară. De fapt, o parte importantă a rezolvării unei probleme este proiectarea matematică a algoritmului, lucru care nu se poate face decât cu creionul pe hârtie. Pe lângă aceasta, majoritatea problemelor operează cu vectori, matrice, arbori, grafuri etc., iar exemplele pe care este testat programul realizat trebuie neapărat verificate „de mână". E de preferat să aveți hârtie de matematică; este foarte folositoare pentru problemele de geometrie analitică, precum și pentru reprezentarea matricelor. Cantitatea depinde de imaginația fiecăruia. În unele cazuri speciale, autorului i s-a întâmplat să umple 7-8 coli A4.

## 1.2 În timpul concursului

Din fericire pentru unii și din nefericire pentru alții, majoritatea examenelor îți cer să dovedești nu că ești bine pregătit, ci că ești mai bine pregătit decât alții. Aceasta înseamnă că și la olimpiada de informatică se aplică legea peștelui mai mare sau, cum i se mai spune, a concurenței. Valoarea absolută a fiecăruia nu contează chiar în totalitate, ceea ce constituie sarea și piperul concursului. Într-adevăr, ce farmec ar avea să mergi la un concurs la care se știe încă dinainte cine este cel mai bun ? Este destul de amuzant să observi cum fiecare speră să prindă „o zi bună", iar adversarii săi „o zi proastă".

Este ușor să fii printre cei mai buni atunci când concursul este ușor. Mai greu e să fii cel mai bun atunci când concursul este dur, pentru că atunci intervine - inevitabil - dramul de noroc al fiecăruia. Niciodată însă nu se poate invoca greutatea concursului drept o scuză pentru un eventual eșec. Concursul este la fel de greu pentru toți. Se poate întâmpla, mai ales dacă probele durează mai multe zile (3-4) ca nici unul din concurenți să nu acumuleze mai mult de 70-80% din punctajul maxim. Totuși, aceasta nu înseamnă că ei nu sunt bine pregătiți; mai mult, unul dintre ei trebuie să fie primul. Așadar, niciodată nu trebuie adoptată o strategie de genul „problema asta e grea și n-am s-o pot rezolva perfect, așa că nu mă mai apuc deloc de ea". Nu trebuie să vă impacientați dacă vi se întâmplă să nu aveți o idee genială de rezolvare a unei probleme. Nu vă cere nimeni să faceți perfect o problemă, ci numai să prezentați o soluție care să acumuleze cât mai multe puncte. Evident, prima variantă este întotdeauna preferabilă, dar nu obligatorie.

De multe ori se întâmplă ca un elev să găsească o soluție cât de cât bună la o problemă și, măcar că știe că nu va lua punctajul maxim, ci doar o parte, să renunțe să caute o soluție mai eficientă, deoarece timpul pierdut astfel ar aduce un câștig prea mic și ar putea fi folosit la rezolvarea altor probleme. Desigur, dacă nu faci toate problemele perfect, nu mai poți fi sigur de premiul I, pentru că altcineva poate să te întreacă. Dar pe de altă parte, locul pe care te clasezi contează numai la etapa națională a olimpiadei sau la concursurile internaționale. În rest, important e numai să te califici, adică să intri în primele câteva locuri.

Feriți-vă ca de foc de criza de timp. E mare păcat să ratezi o problemă întreagă pentru că n-ai avut timp să scrii procedura de afișare a soluției. Rezervați-vă întotdeauna timpul pe care îl socotiți necesar pentru implementare și depanare.

Niciodată, chiar dacă concursul este ușor, nu e bine să ieșiți din sala de concurs înainte de expirarea timpului. Oricât ați fi de convinși că ați făcut totul perfect, mai verificați-vă; veți avea de furcă cu remușcările dacă descoperiți după aceea că ceva, totuși, nu a mers bine. Puteți face o mulțime de lucruri dacă mai aveți timp (deși acest lucru se întâmplă rar). Iată o serie de metode de a exploata timpul:

- Verificați-vă programul cu cât mai multe teste de mici dimensiuni. Să presupunem că programul vostru lucrează cu vectori de maxim 10.000 de elemente. E o idee bună să îl rulați pentru vectori de unul sau două elemente. Nu se știe cum pot să apară erori.
- Treceți la polul opus și creați-vă un test de dimensiune maximă, dar cu o structură particulară, pentru care este ușor de calculat rezultatul și de mână. De exemplu, vectori de 10.000 elemente cu toate elementele egale, sau vectori de forma (1, 2, ...,

9.999, 10.000). Dacă nu puteți să editați un asemenea fișier de mână, copiind și multiplicând blocuri, puteți scrie un program care să-l genereze.

- Dacă încă v-a mai rămas timp, creați-vă un program care să genereze teste aleatoare. Spre exemplu, un program care să citească un număr $N$ și să creeze un fișier `INPUT.TXT` în care să scrie $N$ numere aleatoare. Într-o primă fază, puteți folosi aceste teste pentru a verifica dacă nu cumva la valori mai mari programul nu dă eroare, nu se blochează (la alocarea unor zone mari de memorie) sau nu depășește limita de timp, caz în care mai aveți de lucru.

- Dacă tot nu vă dă nimeni afară din sală, puteți scrie un alt program auxiliar care, primind fișierul `INPUT.TXT` și fișierul `OUTPUT.TXT` produs de programul vostru, verifică dacă ieșirea este corectă. Aceasta deoarece, de obicei, este mult mai ușor de verificat o soluție decât de produs una (sau, cum spunea Murphy, „cunoașterea soluției unei probleme poate ajuta în multe cazuri la rezolvarea ei"). Folosind „generatorul" de teste și „verificatorul", puteți testa programul mult mai bine. De altfel, la multe probleme chiar testele rulate de comisia de corectare sunt create tot aleator.

- În caz că ați dat o soluție euristică la o problemă NP-completă, puteți implementa și un backtracking ca să vedeți cât de bune sunt rezultatele găsite euristic. Apoi, puteți începe să modificați funcția euristică pentru a o face cât mai performantă.

Și, ca să nu mai lungim vorba, iată o strategie care pare să dea rezultate:

**A)** Imediat ce primiți problemele, citiți toate enunțurile și faceți-vă o idee aproximativă despre gradul de dificultate al fiecărei probleme. Neapărat verificați dacă se dau limite pentru datele de intrare (numărul maxim de elemente ale unui vector și valoarea maximă a acestora, numărul maxim de noduri dintr-un graf etc.) și pentru timpii de execuție pentru fiecare test. Dacă nu se dau, întrebați imediat. Dimensiunea input-ului poate schimba radical dificultatea problemei. Spre exemplu, pentru un vector cu $N = 100$ elemente, un algoritm $O(N^3)$ merge rezonabil, pe când pentru $N = 10.000$ același algoritm ar depăși cu mult cele câteva secunde care se acordă de obicei. Fair-play-ul cere să puneți întrebările cu voce tare, pentru ca și ceilalți să audă; de altfel, nu aveți nici un motiv să vă feriți de ceilalți concurenți. Cei care sunt interesați de aceste întrebări le-ar pune oricum și ei, iar cei care nu sunt interesați vor ignora oricum răspunsul.

Dacă există probleme care cer să se găsească un optim (maxim/minim) al unei valori, întrebați dacă se acordă punctaje parțiale pentru soluții neoptime. Și acest fapt poate schimba natura problemei. După aceasta,

```
1  Nr_probleme_nerezolvate := Nr_probleme_primite;
2
3  while (Nr_probleme_nerezolvate>0)
4    and not ('Timpul a expirat, va rugam sa salvati') do
5    begin
```

**B)** Faceți o împărțire a timpului pentru problemele rămase proporțional cu punctajul fiecărei probleme. În general problemele au punctaje egale, dar nu totdeauna. De exemplu, dacă o problemă e cotată cu 100 puncte, iar alta cu 50, veți aloca de două ori mai mult timp primei probleme, chiar dacă nu vi se pare prea grea. Încercați să nu depășiți niciodată limitele de timp pe care le-ați fixat. Dacă în schimb reușiți să economisiți timp față de cât v-ați propus, cu atât mai bine, veți face o realocare a timpului și veți avea mai mult pentru celelalte probleme.

**C)** Apucați-vă de problema **cea mai simplă**, chiar dacă e punctată mai slab. Mai bine să duceți la bun sfârșit o problemă ușoară și să luați un punctaj mai mic, decât să vă apucați de o problemă grea și să nu terminați niciuna. Dacă toate problemele par grele, alegeți-o pe cea din domeniul care vă este cel mai familiar, în care ați lucrat cel mai mult. Dacă vă este indiferent și acest lucru, alegeți o problemă unde simțiți că aveți o idee simplă de rezolvare. Dacă, în sfârșit, nu aveți nici o idee la nici o problemă, apucați-vă de cea mai bine punctată.

**D)** Citiți din nou enunțul, de data aceasta cu mare grijă. Întrebați supraveghetorul pentru orice nelămurire. Dacă anumite lucruri nu sunt specificate, iar profesorul nu vă dă nici un fel de informații suplimentare, tratați problema în cazul cel mai general. Iată mai multe exemple frecvente în care enunțul nu este limpede:

- Dacă nu se precizează cât de mari pot fi întregii dintr-un vector, nu lucrați pe `Integer`, nici pe `Word`, ci pe `LongInt`;
- În problemele de geometrie analitică, e bine să presupuneți că punctele nu au coordonate întregi, ci reale;
- De asemenea, pătratele și dreptunghiurile nu au neapărat laturi paralele cu axele, ci sunt așezate oricum în plan (aceasta poate într-adevăr să complice extrem de mult problema; nu vă doresc să vă izbiți de o asemenea neclaritate...);
- Dacă fișierul de intrare conține string-uri, să nu presupuneți că ele au maxim 255 de caractere. Mai bine scrieți propria voastră procedură de citire a unui string, care să citească din fișier caracter cu caracter până la `Eoln`, decât să aveți surprize.

Dacă $S$ este o variabilă de tip `String`, `ReadLn(S)` ignoră tot restul rândului care depășește lungimea lui $S$.

- Grafurile nu sunt neorientate, ci orientate. În principiu, enunțul nu are voie să fie neclar în această privință, dar au existat cazuri de neînțelegere.

**E)** Începeți să vă gândiți la algoritmi cât mai buni, estimând în același timp și cât v-ar lua ca să-i implementați. Faceți, pentru fiecare idee care vă vine, calculul complexității. Nu trebuie neapărat să găsiți cel mai eficient algoritm, ci numai unul suficient de bun. În general, trebuie ca, dintre toți algoritmii care se încadrează în timpul de rulare, să-l alegeți pe cel care este cel mai ușor de implementat. Iată un exemplu:

- Să presupunem că timpul de testare este de 5 secunde, lucrați pe un 486DX4, algoritmul vostru are complexitatea $O(N^3)$, iar $N$ este maxim 100. Un 486 face câteva milioane de operații elementare pe secundă, să zicem 4.000.000. Aceasta înseamnă ceva mai puține operații mai costisitoare (atribuiri, comparații etc.) pe secundă. Să ne oprim deci la cifra de 1.000.000. Programul vostru are timp de rulare cubic, iar $N^3$ este maxim 1.000.000. De aici deducem că programul ar trebui să se încadreze într-o secundă. Calculul nostru este grosier, dar luând și o marjă de eroare arhisuficientă, rezultă că programul trebuie să meargă cu ușurință în 5 secunde, deci algoritmul este acceptabil.

**F)** Dacă algoritmul găsit este greu de implementat, mai căutați un altul o vreme. Trebuie însă ca timpul petrecut pentru găsirea unui nou algoritm plus timpul necesar pentru scrierea programului să nu depășească timpul necesar pentru implementarea primului algoritm, altfel nu câștigați nimic. Deci nu exagerați cu căutările și nu încercați să reduceți dincolo de limita imposibilului complexitatea algoritmului. Mai ales, nu uitați că programul nu poate avea o complexitate mai mică decât dimensiunea input-ului sau a output-ului. De exemplu, dacă programul citește sau scrie matrice de dimensiune $N \times N$, nu are sens să vă bateți capul ca să găsiți un algoritm mai bun decât $O(N^2)$.

**G)** Dintre toate ideile de implementare găsite (care se încadrează fără probleme în timp), o veți alege pe cea mai scurtă ca lungime de cod. De exemplu:

- Dacă $N \leq 1.000$ și dispuneți de doi algoritmi, unul pe care îl estimați cam la 200 de linii de program, de complexitate $O(N \log N)$ și unul de 100 de linii de complexitate $O(N^2)$, cel de-al doilea este evident preferabil, pentru că nu pierdeți nimic din punctaj, sau cel mult pierdeți un test prin cine știe ce întâmplare, în schimb

câștigați timp prețios pe care îl puteți folosi pentru alte probleme. Bineînțeles, primul program este mai eficient, dar în condiții de concurs el este **prea** eficient. Este o mândrie să faceți o problemă perfect chiar dacă ratați o alta, dar este un câștig și mai mare să faceți amândouă problemele suficient de bine.

**H)** În general, pentru orice problemă există cel puțin o soluție, fie și una slabă. Sunt numeroase cazurile când nici nu vă vine altă idee de rezolvare decât cea slabă. De regulă, când nu aveți în minte decât o rezolvare neeficientă a problemei, care știți că nu o să treacă toate testele (un backtracking, sau un $O(N^5)$, $O(N^6)$ etc.), e bine să încercați următorul lucru:

- Să presupunem că v-a mai rămas o oră pentru rezolvarea acestei probleme. Calculați cam cât timp v-ar trebui ca să implementați rezolvarea slabă. Să zicem 40 de minute. În acest calcul trebuie să includeți și timpul de depanare a programului (care variază de la persoană la persoană) și pe cel de testare. Dacă sunteți foarte siguri pe voi, puteți să neglijați timpul de testare, dar orice program trebuie testat cel puțin pe exemplul de pe foaie.
- Mai rămân deci 20 de minute, timp în care vă puteți gândi la altceva, la altă soluție. Pentru a avea șanse mai mari să găsiți o altă soluție, este indicat să încercați să ignorați complet soluția slabă, să nu o luați ca punct de plecare. Încercați să vă „goliți" mintea și să găsiți ceva nou, altfel vă veți învârti mereu în cerc.
- Dacă vă vine vreo idee mai bună, ați scăpat de griji și mergeți la punctul **(F)**. Altfel, la expirarea timpului de 20 de minute, vă apucați să implementați soluția pe care o aveți, oricât de ineficientă ar fi (de fapt, orice soluție, oricât de ineficientă, trebuie să ia măcar o treime sau o jumătate din punctaj, dacă nu apar erori de implementare).
- Puteți, ca o măsură extremă, să depășiți cu **maximum** 5 minute cele 20 de minute planificate, dar de cele mai multe ori acesta e timp pierdut, deoarece intervine stresul și nu puteți să vă mai concentrați.

**I)** Dacă ați ajuns până aici înseamnă că ați optat pentru o variantă de implementare. Din acest moment, pentru această variantă veți scrie programul, fără a vă mai gândi la altceva, chiar dacă pe parcurs vă vin alte idei. Iată unele lucruri pe care e bine să le știți despre scrierea unui program:

- Datele de intrare se presupun a fi corecte. Aceasta este o regulă nescrisă (uneori) a concursului de informatică. Chiar dacă, prin absurd, știți sigur că datele de intrare trebuie verificate, mai bine n-o faceți, din mai multe motive. În primul rând, scopul cu care v-a fost dată problema este altul decât să se constate cine verifică mai bine datele de intrare. De aceea, cel mult un test sau două vor fi cu date greșite. În al doilea rând, nu se justifică să risipiți atâta timp numai pentru câteva puncte pe care le-ați putea pierde dacă nu faceți verificarea. În al treilea rând, e posibil să greșiți oricum problema în sine, caz în care nu mai contează dacă ați citit perfect datele de intrare. În sfârșit, legea lui Murphy spune că „oricâte teste ar efectua cineva asupra datelor de intrare, se va găsi cineva care să introducă date greșite". Efortul este deci zadarnic...

- Ultimul lucru, când sunteți convinși că programul este terminat și când v-ați hotărât să nu îl mai modificați, adăugați opțiunile de compilare. Puteți face aceasta apăsând `Ctrl-O O`. La începutul programului vor apărea directivele de compilare. Setați $B, $I, $R și $S pe - (minus). Eventual, puteți include direct linia `{$B-,I-,R-,S-}` imediat după titlul programului. Aceasta va face compilatorul să nu mai evalueze complet expresiile booleene, să nu mai verifice operațiile de intrare/ieșire, domeniul de atribuire (*Range Checking*) și stiva (*Stack Checking*). Există două avantaje majore: în primul rând că programul merge mai repede (se câștigă câteva procente bune la viteză), iar în al doilea rând, psihologic vorbind, este preferabil ca un program să se blocheze decât să se oprească printr-un banal `Range check error`. Nu vă grăbiți să puneți directivele de compilare încă de la început, deoarece nu veți mai primi mesajele corespunzătoare de eroare și vă va fi mai greu să depanați programul.

- Pe cât este posibil, încercați să convingeți juriul să nu vă ruleze sursa (Pascal sau C/C++), ci direct executabilul. Merge simțitor mai repede. Asta numai în cazul în care vă temeți că programul ar putea să nu se încadreze în timp.

- Dacă se poate, evitați lucrul cu pointeri. Programele care îi folosesc sunt mai greu de depanat și se pot bloca mult mai ușor.

- Să presupunem că aveți de lucrat cu matrice de dimensiuni maxim $100 \times 100$. Unii elevi au obiceiul să dimensioneze la început matricele de $5 \times 5$ sau $10 \times 10$, doarece sunt mai comod de evaluat cu *Evaluate* (`Ctrl-F4`) sau *Watch* (`Ctrl-F7`). Acest lucru este adevărat, dar există riscul ca la sfârșit să uitați să redimensionați matricele de $100 \times 100$. Decât să faceți o asemenea greșeală (care în mod sigur vă va compromite toată problema), mai bine setați dimensiunile corecte de la început.

De altfel, ideal ar fi ca depanarea programelor să fie suprimată cu totul și programul să meargă din prima.

- Evitați lucrul cu numere reale, dacă puteți. Operațiile în virgulă mobilă sunt mult mai lente. De exemplu, testul dacă un număr este prim nu va începe în nici un caz cu linia

```
    while i <= Sqrt(N) do
```

ci cu linia

```
    while i * i <= N do
```

Din punct de vedere logic, condițiile sunt perfect echivalente. Totuși, prima se evaluează de câteva zeci de ori mai încet decât a doua.

- Dacă lucrați cu numere reale, nu folosiți testul

```
    R1=R2
```

deoarece pot apărea erori, ci implementați o funcție:

```
    function Equal(R1,R2:Real):Boolean;
    begin
      Equal:=(Abs(R1-R2)<0.00001);
    end;
```

Numărul de zerouri de după virgulă trebuie să fie suficient de mare astfel încât două numere diferite să nu fie tratate drept egale (se poate lucra de pildă cu 0.000001).

- Tot în cazul numerelor reale, evitați pe cât posibil să faceți împărțiri, deoarece sunt foarte greoaie. De exemplu:

  $X/5 \leftrightarrow 0.2 * X$
  $X/Y/Z \leftrightarrow X/(Y * Z)$

- Optimizările de genul „`X shl 1`” respectiv „`X<<1`” în loc de „`2*X`” sunt niște artificii de cele mai multe ori neesențiale, care în schimb fac formulele mai lungi, greu de urmărit și pot crea complicații. Cel mai bine este să lucrați cu notațiile obișnuite și doar la sfârșit, dacă timpul de rulare trebuie redus cu orice preț, să faceți înlocuirile.

- Alegeți-vă numele de variabile în așa fel încât programul să fie clar. Sunt permise mai mult de două litere! Numele fiecărei proceduri, funcții, variabile trebuie să-i explice clar utilitatea. E drept, lungimea programului crește, dar codul devine mult mai limpede și timpul de depanare scade foarte mult. Ca o regulă generală, claritatea programelor face mult mai ușoară înțelegerea lor chiar și după o perioadă mai îndelungată de timp (luni, ani). Nu trebuie nici să cădeți în cealaltă extremă. De exemplu, nu depășiți 10 caractere pentru un nume de variabilă.

- Salvați programul cât mai des. Dacă vă obișnuiți, chiar la fiecare două-trei linii. După ce o să vă intre în reflex n-o să vă mai incomodeze cu nimic acest obicei, mai ales că în ziua de azi salvarea unui program de 2-3 KB se face practic instantaneu. Au fost frecvente cazurile în care o pană de curent prindea pe picior greșit mulți concurenți, iar după aceea nu mai este absolut nimic de făcut, pentru că nimeni nu vă va crede pe cuvânt că ați făcut programul și că el mergea.

- Obișnuiți-vă să programați modular. Faceți proceduri separate pentru citirea și inițializarea datelor, pentru sortare, pentru afișarea rezultatelor etc. În general nu se recomandă să scrieți proceduri în alte proceduri (adică e bine ca toate procedurile să aparțină direct de programul principal). Procedurile, acolo unde e posibil, nu trebuie să depășească un ecran, pentru a putea avea o viziune de ansamblu asupra fiecăreia în parte . Acest lucru ajută mult la depanare.

- Rulați programul cât mai des. În primul rând după ce scrieți procedura de citire a datelor. Dacă e nevoie de sortarea datelor de intrare, nu strică să vă convingeți că programul sortează bine, rulând două-trei teste oarecare. E păcat să pierdeți puncte dintr-o greșeală copilărească.

- O situație delicată apare când fișierul de intrare conține mai multe seturi de date (teste). În acest caz, atenția trebuie sporită, deoarece dacă la primul sau al doilea test programul vostru dă eroare și se oprește din execuție, veți pierde automat și toate celelalte teste care urmează. Dacă în fișierul de intrare exista un singur set de date, atunci pierderea din vedere a unui caz particular al problemei nu putea duce, în cel mai rău caz, decât la picarea unui test. Așa însă, picarea unui test poate atrage după sine picarea tuturor celor care îi urmează. Pe lângă corectitudinea strict necesară, programul trebuie să se încadreze și în timp pentru orice fel de test. Dacă la primul sau al doilea test din suită programul depășește timpul (sau, și mai rău, se blochează), e foarte probabil să fie oprit din execuție de către comisie, deci din nou veți pierde toate testele care au rămas neexecutate. Uneori comisia este binevoitoare și acceptă să modifice fișierul de date, tăind din el setul pe care

programul vostru nu merge, dar acest lucru este greoi și nu tocmai cinstit față de ceilalți concurenți, deci nu vă bazați pe asta.

- Tot în situația în care există mai multe seturi de date în fișierul de intrare, dacă ieșirea se face într-un fișier, este bine ca după afișarea rezultatului pentru fiecare test să actualizați fișierul de ieșire (fie prin comanda `Flush`, fie prin două proceduri, `Close` și `Append`). În felul acesta, chiar dacă la unul din teste programul se blochează sau dă eroare, rezultatele deja scrise rămân scrise. Altfel, e posibil ca rezultatele de la testele anterioare să rămână într-un buffer în memorie, fără a fi „vărsate" pe disc.

- Dacă fișierul de ieșire are dimensiuni foarte mari, de exemplu dacă vi se cer toate soluțiile, iar numărul acestora este de ordinul zecilor de mii, puteți avea surpriza ca timpul să nu vă ajungă pentru a le tipări pe toate în fișierul de ieșire. Și în acest caz, este recomandat ca după tipărirea fiecărei soluții să executați comanda `Flush`, sau să închideți fișierul de ieșire și să-l redeschideți în modul `Append`.

**J)** Dacă ați trecut cu bine și de faza de scriere a programului, mai aveți doar părțile de depanare și testare, care de multe ori se îmbină. Metoda cea mai bună de depanare este următoarea:

- Începeți cu un test nici prea simplu, nici prea complicat (și ușor de urmărit cu creionul pe hârtie) și executați-l de la cap la coadă. Dacă merge perfect, treceți la teste mai complexe (**minimum** 4 teste și maxim 7-8). Dacă le trece și pe acestea, puteți fi mândri. Legile lui Murphy în programare se aplică în continuare: „Depanarea nu poate demonstra că un program merge; ea poate cel mult demonstra că un program nu merge". Totuși, dacă programul vostru a mers perfect pe 7-8 teste date la întâmplare, există șanse foarte mari să meargă pe majoritatea testelor comisiei, sau chiar pe toate.

- Exemplul dat în enunț nu are în general nici o semnificație deosebită (de fapt, are mai curând darul de a semăna confuzie printre concurenți), iar dacă programul merge pe acest test particular, nu înseamnă că o să meargă și pe alte teste. În culegere nu a fost explicat pe larg algoritmul decât pe exemplul din enunțul fiecărei probleme, dar aceasta s-a făcut numai pentru a nu supraîncărca materialul.

- Dacă la unul din teste programul nu merge corespunzător, rulați din nou testul, dar de data aceasta procedură cu procedură. După fiecare procedură evaluați variabilele

și vedeți dacă au valorile așteptate. În felul acesta puteți localiza cu precizie procedura, apoi linia unde se află eroarea. Corectați în această manieră toate erorile, până când testul este trecut.

- În acest moment, luați de la capăt toate testele pe care programul le-a trecut deja. În urma depanării, s-ar putea ca alte greșeli să iasă la suprafață și programul să nu mai meargă pe vechile teste.
- Repetați procedeul de mai sus până când toate testele merg. Dacă vă obișnuiți să programați modular și îngrijit, depanarea și testarea n-ar trebui să dureze mai mult de 5-10 minute. Din acest moment, nu mai modificați nici măcar o literă în program, sau dacă țineți să o faceți, păstrați-vă în prealabil o copie. Nu vă bazați pe faptul că puteți să țineți minte modificările făcute și să refaceți oricând forma inițială a programului în caz că noua versiune nu va fi bună.
- Dacă totuși nu-i puteți „da de cap" programului, iar timpul alocat problemei respective expiră, aduceți programul la o formă în care să meargă măcar pe o parte din teste (pe jumătate, de exemplu) și treceți la problema următoare.

**K)**

```
    Dec(Nr_probleme_nerezolvate);
  end; { while }
```

Probabil nu veți fi de acord cu toate sfaturile date mai sus. E bine însă să le aplicați. Scopul pentru care ele au fost incluse în această carte este de a ajuta concurenții să se acomodeze mai ușor cu atmosfera concursului. De multe ori, primul an de participare la olimpiadă se soldează cu un rezultat cel mult mediu, deoarece, oricât ar spune cineva „ei, nu-i așa mare lucru să mergi la un concurs", experiența acumulată contează mult. De aceea, abia de la a doua participare și uneori chiar de mai târziu încep să apară rezultatele. Intenția autorului a fost să vă ușureze misiunea și să vă dezvăluie câteva din dificultățile de toate felurile care apar la orice concurs, pentru a nu vă da ocazia să le descoperiți pe propria piele. Poate că aceste ponturi vă vor fi de folos.

# Capitolul 2

# Lucrul cu numere mari

De multe ori, în probleme, apar situații când este nevoie să memorăm numere întregi foarte mari (de ordinul sutelor de cifre), iar uneori trebuie să efectuăm și operații aritmetice cu aceste numere. Iată un asemenea exemplu:

**ENUNȚ**: Se dă un număr natural cu $N \leq 1.000$ cifre. Se cere să se extragă rădăcina cubică a numărului. Se garantează că numărul citit este cub perfect.

**Intrarea**: Fișierul `INPUT.TXT` conține un singur rând, terminat cu `EOF`, pe care se află numărul, cifrele fiind nedespărțite.

**Ieșirea**: În fișierul `OUTPUT.TXT` se va tipări rădăcina cubică a numărului, pe o singură linie terminată cu `EOF`.

**Exemplu:**

| **INPUT.TXT** | **OUTPUT.TXT** |
|---|---|
| `2097152` | `128` |

**Timp de implementare**: 1h 30'.

**Timp de rulare**: 10 secunde.

**Complexitate cerută**: $O(N^2)$.

**REZOLVARE**: Problema este cât se poate de simplă din punct de vedere matematic; dificultatea apare la implementare, atât datorită structurilor de date necesare, cât mai ales datorită complexității cerute. Despre lucrul cu numere întregi (chiar și `Longint`) nici nu poate fi vorba, iar la lucrul cu numere reale apar erori de calcul.

Structura de date propusă pentru abordarea acestui gen de probleme este următoarea: numerele vor fi reprezentate printr-un vector de cifre zecimale. Prima poziție (poziția 0)

din vector este rezervată pentru a memora numărul de cifre. Definiția C a tipului de date este:

```
1 typedef int Huge[1001];
```

Iată cum s-ar memora numărul „1997” într-un asemenea vector:

| | H[0] | H[1] | H[2] | H[3] | H[4] |
|---|---|---|---|---|---|
| H | 4 | 7 | 9 | 9 | 1 |

Se observă că vectorul este oarecum „întors pe dos”. Totuși, această formă este cea mai convenabilă, pentru că ea permite implementarea cu o mai mare ușurință a operațiilor aritmetice.

Mai trebuie remarcat că pe fiecare poziție $K$ în vector se află cifra care îl reprezintă pe $10^{K-1}$ în numărul reprezentat: în $H[1]$ se află cifra unităților ($10^0$), în $H[2]$ se află cifra zecilor ($10^1$), în $H[3]$ se află cifra sutelor ($10^2$) ș.a.m.d. Formatul zecimal nu este cea mai fericită (a se citi „eficientă”) alegere. El ocupă doar patru biți din cei opt ai unui octet, deci face risipă de memorie. Dacă am alege baza de numerație 256, am folosi la maximum memoria, și în plus operațiile ar fi cu mult mai rapide (deoarece 256 este o putere a lui 2, înmulțirile și împărțirile la 256 sunt simple deplasări la stânga și la dreapta ale reprezentărilor binare). Să facem următorul calcul ca să vedem câte cifre are în baza 256 un număr care are 1.000 de cifre în baza 10:

$$10^{1.000} = 10 \cdot (10^3)^{333} \approx 10 \cdot (2^{10})^{333} \approx 10 \cdot 2^2 \cdot (2^8)^{416} = 40 \cdot 256^{416} \tag{2.1}$$

Așadar, numărul de cifre s-a redus la sub jumătate. Un algoritm liniar ar funcționa de două ori mai repede pe reprezentări în baza 256, iar unul pătratic ar funcționa de patru ori mai repede. Inconvenientul major este dificultatea depanării unui program care operează într-o bază aritmetică atât de străină nouă. Vom rămâne deci la baza 10, cu mențiunea că acei mai temerari dintre voi pot încerca folosirea bazei 256.

Numerele reprezentate pe vectori le vom numi pur și simplu „vectori”, iar numerele reprezentate printr-un tip ordinal de date le vom numi, printr-o analogie ușor forțată cu matematica, „scalari”. Să vedem acum cum se efectuează operațiile elementare pe aceste numere.

## 2.1 Inițializarea

Un vector poate fi inițializat în trei feluri: cu 0, cu un scalar sau cu un alt vector.

La inițializarea cu 0, singurul lucru pe care îl avem de făcut este să setăm numărul de cifre pe 0. De aceea, este practic inutil să implementăm această funcție ca atare; putem folosi în loc singura instrucțiune pe care ea o conține.

```
1 void Atrib0(Huge H)
2 { H[0]=0; }
```

La inițializarea cu un scalar nenul, trebuie să așezăm fiecare cifră pe poziția corespunzătoare, aflând în paralel și numărul de cifre. Se începe cu cifra unităților, și la fiecare pas se pune în vector cifra cea mai puțin semnificativă, după care numărul de reprezentat se împarte la 10 (neglijându-se restul), iar numărul de cifre se incrementează.

```
1 void AtribValue(Huge H, unsigned long X)
2 { H[0]=0;
3   while (X)
4     { H[++H[0]]=X%10;
5       X/=10;
6     }
7 }
```

Iată, de exemplu, cum se pune pe vector numărul 195:

| | | H | |
|---|---|---|---|
| | | 0 0 0 0 | |
| $X = 195$ | $X \bmod 10 = 5$ | 1 5 0 0 | $X/10 = 19$ |
| $X = 19$ | $X \bmod 10 = 9$ | 2 5 9 0 | $X/10 = 1$ |
| $X = 1$ | $X \bmod 10 = 1$ | 3 5 9 1 | $X/10 = 0$ |

În sfârșit, inițializarea unui vector cu altul se face printr-o simplă copiere (se pot folosi cu succes rutine de lucru cu memoria, cum ar fi `FillChar` în Pascal sau `memmove` în C). Pentru eleganță, poate fi folosită și atribuirea cifră cu cifră:

```
void AtribHuge(Huge H, Huge X)
{ int i;

  for (i=0;i<=X[0];i++) H[i]=X[i];
}
```

## 2.2 Compararea

Pentru a compara două numere „uriașe”, începem prin a afla numărul de cifre semnificative (deoarece, în urma anumitor operații pot rezulta zerouri nesemnificative care „atârnă” totuși la numărul de cifre). Aceasta se face decrementând numărul de cifre al fiecărui număr atâta timp cât cifra cea mai semnificativă este 0. După ce ne-am asigurat asupra acestui punct, comparăm numărul de cifre al celor două cifre. Numărul cu mai multe cifre este cel mai mare. Dacă ambele numere au același număr de cifre, pornim de la cifra cea mai semnificativă și comparăm cifrele celor două numere până la prima diferență întâlnită. În acest moment, numărul a cărui cifră este mai mare este el însuși mai mare. Dacă toate cifrele numerelor sunt egale două câte două, atunci în mod evident numerele sunt egale.

După cum se vede, algoritmul seamănă foarte bine cu ceea ce s-a învățat la matematică prin clasa a II-a (doar că atunci nu ni s-a spus că este vorba de un „algoritm”). Rutina de mai jos compară două numere uriașe $H_1$ și $H_2$ și întoarce -1, 0 sau 1, după $H_1$ este mai mic, egal sau mai mare decât $H_2$.

```
int Sgn(Huge H1,Huge H2)
{ int i;

  while (!H1[H1[0]] && H1[0]) H1[0]--;
  while (!H2[H2[0]] && H2[0]) H2[0]--;
  if (H1[0]!=H2[0])
    return H1[0]<H2[0] ? -1 : 1;
  i=H1[0];
  while (H1[i]==H2[i] && i) i--;
  return H1[i]<H2[i] ? -1 : H1[i]==H2[i] ? 0 : 1;
}
```

## 2.3 Adunarea a doi vectori

Fiind dați doi vectori, $A$ cu $M$ cifre și $B$ cu $N$ cifre, adunarea lor se face în mod obișnuit, ca la aritmetică. Pentru a evita testele de depășire, se recomandă să se completeze mai întâi vectorul cu mai puține cifre cu zerouri până la dimensiunea vectorului mai mare. La sfârșit, vectorul sumă va avea dimensiunea vectorului mai mare dintre $A$ și $B$, sau cu 1 mai mult dacă apare transport de la cifra cea mai semnificativă. Procedura de mai jos adaugă numărul $B$ la numărul $A$.

```
void Add(Huge A, Huge B)
/* A <- A+B */
{ int i,T=0;

  if (B[0]>A[0])
    { for (i=A[0]+1;i<=B[0];) A[i++]=0;
      A[0]=B[0];
    }
    else for (i=B[0]+1;i<=A[0];) B[i++]=0;
  for (i=1;i<=A[0];i++)
    { A[i]+=B[i]+T;
      T=A[i]/10;
      A[i]%=10;
    }
  if (T) A[++A[0]]=T;
}
```

## 2.4 Scăderea a doi vectori

Se dau doi vectori $A$ și $B$ și se cere să se calculeze diferența $A - B$. Se presupune $B \leq A$ (acest lucru se poate testa cu funcția `Sgn`). Pentru aceasta, se pornește de la cifra unităților și se memorează la fiecare pas „împrumutul" care trebuie efectuat de la cifra de ordin imediat superior (împrumutul poate fi doar 0 sau 1). Deoarece $B \leq A$, avem garanția că pentru a scădea cifra cea mai semnificativă a lui $B$ din cifra cea mai semnificativă a lui $A$ nu e nevoie de împrumut.

```
void Subtract(Huge A, Huge B)
/* A <- A-B */
{ int i, T=0;
```

```
4
5     for (i=B[0]+1;i<=A[0];) B[i++]=0;
6     for (i=1;i<=A[0];i++)
7       A[i]+= (T=(A[i]-=B[i]+T)<0) ? 10 : 0;
8       /* Adica A[i]=A[i]-(B[i]+T);
9          if (A[i]<0) T=1; else T=0;
10         if (T) A[i]+=10; */
11    while (!A[A[0]]) A[0]--;
12  }
```

## 2.5 Înmulțirea și împărțirea cu puteri ale lui 10

Aceste funcții sunt uneori utile. Ele pot folosi și funcțiile de înmulțire a unui vector cu un scalar, care vor fi prezentate mai jos, dar se pot face și prin deplasarea întregului număr spre stânga sau spre dreapta. De exemplu, înmulțirea unui număr cu 100 presupune deplasarea lui cu două poziții înspre cifra cea mai semnificativă și adăugarea a două zerouri la coadă. Principalul avantaj al scrierii unor funcții separate pentru înmulțirea cu 10, 100, ..., este că se pot folosi rutinele de acces direct al memoriei (`FillChar`, respectiv `memmove`). Iată funcțiile care realizează deplasarea vectorilor, atât prin mutarea blocurilor de memorie, cât și prin atribuiri succesive.

```
1   void Shl(Huge H, int Count)
2   /* H <- H*10^Count */
3   {
4     /* Shifteaza vectorul cu Count pozitii */
5     memmove(&H[Count+1],&H[1],sizeof(int)*H[0]);
6     /* Umple primele Count pozitii cu 0 */
7     memset(&H[1],0,sizeof(int)*Count);
8     /* Incrementeaza numarul de cifre */
9     H[0]+=Count;
10  }
11
12  void Shl2(Huge H, int Count)
13  /* H <- H*10^Count */
14  { int i;
15
16    /* Shifteaza vectorul cu Count pozitii */
17    for (i=H[0];i;i--) H[i+Count]=H[i];
18    /* Umple primele Count pozitii cu 0 */
19    for (i=1;i<=Count;) H[i++]=0;
```

```
20    /* Incrementeaza numarul de cifre */
21    H[0]+=Count;
22  }
23
24  void Shr(Huge H, int Count)
25  /* H <- H/10^Count */
26  {
27    /* Shifteaza vectorul cu Count pozitii */
28    memmove(&H[1],&H[Count+1],sizeof(int)*(H[0]-Count));
29    /* Decrementeaza numarul de cifre */
30    H[0]-=Count;
31  }
32
33  void Shr2(Huge H, int Count)
34  /* H <- H/10^Count */
35  { int i;
36
37    /* Shifteaza vectorul cu Count pozitii */
38    for (i=Count+1;i<=H[0];i++) H[i-Count]=H[i];
39    /* Decrementeaza numarul de cifre */
40    H[0]-=Count;
41  }
```

## 2.6 Înmulțirea unui vector cu un scalar

Și această operație este o simplă implementare a modului manual de efectuare a calculului. La înmulțirea „de mână” a unui număr mare cu unul de o singură cifră, noi parcurgem deînmulțitul de la sfârșit la început, și pentru fiecare cifră efectuăm următoarele operații:

- Înmulțim cifra respectivă cu înmulțitorul;
- Adăugăm „transportul” de la înmulțirea precedentă;
- Separăm ultima cifră a rezultatului și o trecem la produs;
- Celelalte cifre are rezultatului constituie transportul pentru următoarea înmulțire;
- La sfârșitul înmulțirii, dacă există transport, acesta are o singură cifră, care se scrie înaintea rezultatului.

Exact același procedeu se poate aplica și dacă înmulțitorul are mai mult de o cifră. Singura deosebire este că transportul poate avea mai multe cifre (poate fi mai mare ca

9). Din această cauză, la sfârșitul înmulțirii, poate rămâne un transport de mai multe cifre, care se vor scrie înaintea rezultatului. Iată de exemplu cum se calculează produsul $312 \times 87$:

$$
\begin{array}{lll}
87 \times 2 & = 174 & = 17 \times 10 + 4 \\
87 \times 1 + 17 & = 104 & = 10 \times 10 + 4 \\
87 \times 3 + 10 & = 271 & = 27 \times 10 + 1 \\
87 \times 0 + 27 & = 27 & = 2 \times 10 + 7 \\
87 \times 0 + 2 & = 2 & = 0 \times 10 + 2
\end{array}
\qquad
\begin{array}{rrrl}
 & \mathbf{3} & \mathbf{1} & \mathbf{2} \quad \times \\
 & & \mathbf{8} & \mathbf{7} \\
\hline
 & & & \mathbf{4} \\
 & & \mathbf{4} & \\
 & \mathbf{1} & & \\
\mathbf{7} & & & \\
\mathbf{2} & & &
\end{array}
$$

Procedura este descrisă mai jos:

```
1  void Mult(Huge H, unsigned long X)
2  /* H <- H*X */
3  { int i;
4    unsigned long T=0;
5
6    for (i=1;i<=H[0];i++)
7      { H[i]=H[i]*X+T;
8        T=H[i]/10;
9        H[i]=H[i]%10;
10     }
11   while (T) /* Cat timp exista transport */
12     { H[++H[0]]=T%10;
13       T/=10;
14     }
15 }
```

## 2.7 Înmulțirea a doi vectori

Dacă ambele numere au dimensiuni mari și se reprezintă pe tipul de date `Huge`, produsul lor se calculează înmulțind fiecare cifră a deînmulțitului cu fiecare cifră a înmulțitorului și trecând rezultatul la ordinul de mărime (exponentul lui 10) cuvenit. De fapt, același lucru îl facem și noi pe hârtie. Considerând același exemplu, în care ambele numere sunt „uriașe”, produsul lor se calculează de mână astfel:

| | | | | |
|---|---|---|---|---|
| | | 3 | 1 | 2 | × |
| | | | 8 | 7 | |
| | 2 | 1 | 8 | 4 | |
| 2 | 4 | 9 | 6 | | |
| 2 | 7 | 1 | 4 | 4 | |

S-a luat deci fiecare cifră a înmulțitorului și s-a efectuat produsul parțial corespunzător, corectând la fiecare pas rezultatul prin calculul transportului. Rezultatul pentru fiecare produs parțial s-a scris din ce în ce mai în stânga, pentru a se alinia corect ordinele de mărime. Acest procedeu este oarecum incomod de implementat. Se pot face însă unele observații care ușurează mult scrierea codului:

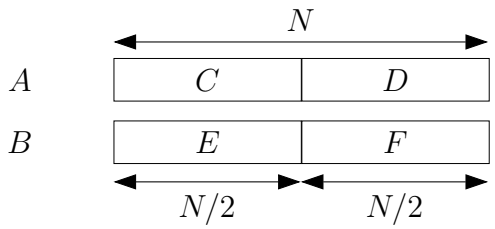
- Prin înmulțirea cifrei cu ordinul de mărime $10^i$ din primul număr cu cifra cu ordinul de mărime $10^j$ din al doilea număr se obține o cifră corespunzătoare ordinului de mărime $10^{i+j}$ în rezultat (sau se obține un număr cu mai mult de o singură cifră, caz în care transportul merge la cifra corespunzătoare ordinului de mărime $10^{i+j+1}$).
- Dacă numerele au $M$ și respectiv $N$ cifre, atunci produsul lor va avea fie $M+N$ fie $M+N-1$ cifre. Într-adevăr, dacă numărul $A$ are $M$ cifre, atunci $10^{M-1} \leq A < 10^M$ și $10^{N-1} \leq B < 10^N$, de unde rezultă $10^{M+N-2} \leq A \times B < 10^{M+N}$.
- La calculul produselor parțiale se poate omite calculul transportului, acesta urmând a se face la sfârșit. Cu alte cuvinte, într-o primă fază putem pur și simplu să înmulțim cifră cu cifră și să adunăm toate produsele de aceeași putere, obținând un număr cu „cifre" mai mari ca 9, pe care îl aducem la forma normală printr-o singură parcurgere. Să reluăm același exemplu:

| | | | | | |
|---|---|---|---|---|---|
| Intrarea: Vectorii $A$ și $B$ | | | **3** | **1** | **2** × |
| | | | | **8** | **7** |
| Etapa I: Efectuarea produselor intermediare | | | 21 | 7 | 14 |
| | | 24 | 8 | 16 | |
| Etapa a II-a: Adunarea produselor intermediare | | 24 | 29 | 23 | 14 |
| | | $\overleftarrow{2}$ | $\overleftarrow{3}$ | $\overleftarrow{2}$ | $\overleftarrow{1}$ |
| Etapa a III-a: Corectarea rezultatului | **2** | **7** | **1** | **4** | **4** |

Această operație efectuează $M \times N$ produse de cifre și $M+N$ (sau $M+N-1$, după caz) „transporturi" pentru aflarea rezultatului, deci are complexitatea $O(M \times N)$. Iată și implementarea:

```
void MultHuge(Huge A, Huge B, Huge C)
/* C <- A x B */
{ int i,j,T=0;

  C[0]=A[0]+B[0]-1;
  for (i=1;i<=A[0]+B[0];) C[i++]=0;
  for (i=1;i<=A[0];i++)
    for (j=1;j<=B[0];j++)
      C[i+j-1]+=A[i]*B[j];
  for (i=1;i<=C[0];i++)
    { T=(C[i]+=T)/10;
      C[i]%=10;
    }
  if (T) C[++C[0]]=T;
}
```

Mai există o altă modalitate de a înmulți două numere de câte $N$ cifre fiecare, care are complexitatea $O(N^{\log_2 3}) \approx O(N^{1,58}) \approx O(N\sqrt{N})$. Ea derivă de un algoritm propus de Strassen în 1969 pentru înmulțirea matricelor. Diferența se face simțită, ce-i drept pentru valori mari ale lui $N$, dar constanta multiplicativă crește destul de mult și, în plus, soluția e mai greu de implementat; de aceea nu recomandăm implementarea ei în timpul concursului. Ideea de bază este să se micșoreze numărul de înmulțiri și să se mărească numărul de adunări, deoarece adunarea a doi vectori se face în $O(N)$, pe când înmulțirea se face în $O(N^2)$. Să considerăm întregii $A$ și $B$, fiecare de câte $N$ cifre. Trebuie să-i înmulțim într-un timp $T(N)$ mai bun decât $O(N^2)$. Să împărțim numărul $A$ în două „bucăți” $C$ și $D$, fiecare de câte $N$ / 2 cifre, iar întregul $B$ în două bucăți $E$ și $F$, tot de câte $N$ / 2 cifre (presupunem că $N$ este par):




Atunci se poate scrie relația

$$\begin{aligned} A \times B &= (C \times 10^{N/2} + D) \times (E \times 10^{N/2} + F) \\ &= CE \times 10^N + (CF + DE) \times 10^{N/2} + DF \end{aligned} \tag{2.2}$$

Pentru a putea calcula produsul $A \times B$ avem, prin urmare, nevoie de patru produse parțiale, de trei adunări și de două înmulțiri cu puteri ale lui 10. Adunările și înmulțirile cu puteri ale lui 10 se fac în timp liniar. Dacă efectuăm cele patru produse parțiale prin patru înmulțiri, rezultă formula recurentă de calcul

$$T(N) = 4T(N/2) + O(N) \tag{2.3}$$

care duce prin eliminarea recurenței la $T(N) \in O(N^2)$. Cu alte cuvinte, încă n-am câștigat nimic. Trebuie să reușim cumva să reducem numărul de înmulțiri de la 4 la 3, chiar dacă prin aceasta vom mări numărul de adunări necesare. Să definim produsul

$$\begin{aligned} G &= (C + D) \times (E + F) \\ &= CE + CF + DE + DF \\ &= CE + DF + (CF + DE) \end{aligned} \tag{2.4}$$

Atunci putem scrie:

$$A \times B = CE \times 10^N + (G - CE - DF) \times 10^{N/2} + DF \tag{2.5}$$

Pentru această variantă, folosim doar trei înmulțiri, și chiar dacă avem nevoie de șase adunări și scăderi și două înmulțiri cu puteri ale lui 10, complexitatea se va reduce la $O(N^{\log_2 3})$. În cazul în care $N$ este o putere a lui 2, împărțirea în două a numerelor se poate face fără probleme la fiecare pas, până se ajunge la numere de o singură cifră, care se înmulțesc direct. În cazul în care $N$ nu este o putere a lui 2, este comod să se completeze numerele cu zerouri până la o putere a lui 2. În funcțiile descrise mai jos, `MultRec` nu face decât înmulțirea recursivă, pe când `MultHuge2` se ocupă și de corectarea numărului de cifre (incrementarea până la o putere a lui 2). Pentru calculul produselor $C \times E$ și $D \times F$, procedura `MultRec` se autoapelează; pentru calcularea produsului $(C + D) \times (E + F)$, însă, este nevoie să fie apelată procedura `MultHuge2`, deoarece prin cele două adunări poate să apară o creștere a numărului de cifre al factorilor, care în acest caz trebuie readuși la un număr de cifre egal cu o putere a lui 2.

```
1 void MultHuge2(Huge A, Huge B, Huge P);
2
3 void MultRec(Huge A, Huge B, Huge P)
4 { Huge C,D,E,F,CE,DF;
```

```
5
6     if (A[0]==1)
7       { P[1]=A[1]*B[1];
8         P[0]=(P[2]=P[1]/10)>0 ? 2 : 1;
9         P[1]%=10;
10      }
11      else { P[0]=0;
12             AtribHuge(C,A);Shr(C,A[0]/2);
13             AtribHuge(D,A);D[0]=A[0]/2;
14             AtribHuge(E,B);Shr(E,B[0]/2);
15             AtribHuge(F,B);F[0]=B[0]/2;
16             MultRec(C,E,CE);MultRec(D,F,DF);
17             Add(C,D);Add(E,F);
18             MultHuge2(C,E,P);
19             Subtract(P,CE);Subtract(P,DF);
20             Shl(P,A[0]/2);
21             Shl(CE,A[0]);Add(P,CE);
22             Add(P,DF);
23           }
24  }
25
26  void MultHuge2(Huge A, Huge B, Huge P)
27  /* P <- A x B, varianta N^(lg 3) */
28  { int i,j,NDig=A[0]>B[0] ? A[0] : B[0],Needed=1,SaveA,SaveB;
29
30    /* Corecteaza numarul de cifre */
31    while (Needed<NDig) Needed<<=1;
32    SaveA=A[0];SaveB=B[0];A[0]=B[0]=Needed;
33    for (i=SaveA+1;i<=Needed;) A[i++]=0;
34    for (i=SaveB+1;i<=Needed;) B[i++]=0;
35    MultRec(A,B,P);
36
37    /* Restaureaza numarul de cifre */
38    A[0]=SaveA;B[0]=SaveB;
39    while (!P[P[0]] && P[0]>1) P[0]--;
40  }
```

## 2.8 Împărțirea unui vector la un scalar

Ne propunem să scriem o funcție care să împartă numărul $A$ de tip `Huge` la scalarul $B$, să rețină valoarea câtului tot în numărul $A$ și să întoarcă restul (care este o variabilă scalară). Să pornim de la un exemplu particular și să generalizăm apoi procedeul: să

calculăm câtul și restul împărțirii lui 1997 la 7. Cu alte cuvinte, să găsim acele numere $C$ de tip `Huge` și $R \in \{0, 1, 2, 3, 4, 5, 6\}$ cu proprietatea că $1997 = 7 \times C + R$.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 9 | 9 | 7 | \| | 7 | | | |
| 0 | | | | \| | 0 | 2 | 8 | 5 ← câtul |
| 1 | 9 | | | | | | | |
| 1 | 4 | | | | | | | |
| | 5 | 9 | | | | | | |
| | 5 | 6 | | | | | | |
| | | 3 | 7 | | | | | |
| | | 3 | 5 | | | | | |
| | | | 2 ← restul | | | | | |

La fiecare pas se coboară câte o cifră de la deîmpărțit alături de numărul deja existent (care inițial este 0), apoi rezultatul se împarte la împărțitor (7 în cazul nostru). Câtul este întotdeauna o cifră și se va depune la sfârșitul câtului împărțirii, iar restul va fi folosit pentru următoarea împărțire. Restul care rămâne după ultima împărțire este tocmai $R$ pe care îl căutăm. Procedeul funcționează și atunci când deîmpărțitul are mai multe cifre. La sfârșit trebuie să decrementăm corespunzător numărul de cifre al câtului, prin neglijarea zerourilor create la începutul numărului. Numărul maxim de cifre al câtului este egal cu cel al deîmpărțitului.

```
unsigned long Divide(Huge A, unsigned long X)
/* A <- A/X si intoarce A%X */
{ int i;
  unsigned long R=0;

  for (i=A[0];i;i--)
    { A[i]=(R=10*R+A[i])/X;
      R%=X;
    }
  while (!A[A[0]] && A[0]>1) A[0]--;
  return R;
}
```

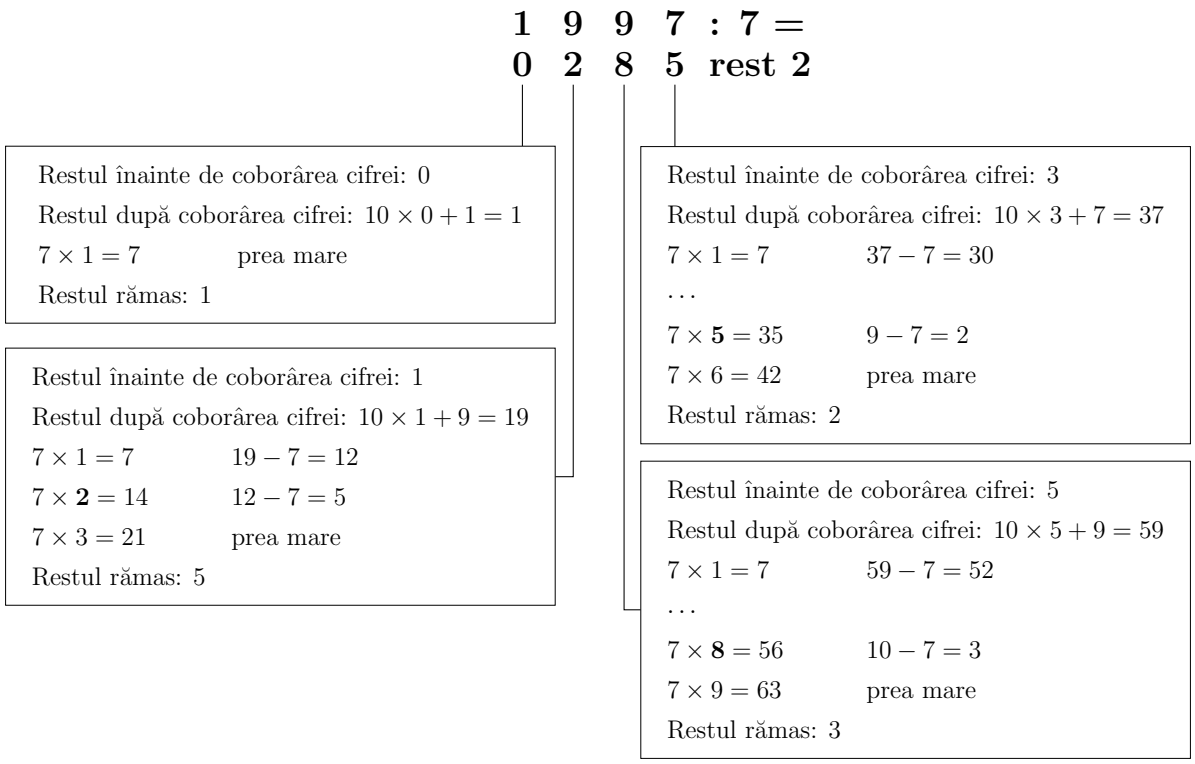
Dacă dorim numai să aflăm restul împărțirii, nu mai avem nevoie decât să recalculăm restul la fiecare pas, fără a mai modifica vectorul $A$:

```
unsigned long Mod(Huge A, unsigned long X)
/* Intoarce A%X */
```

```
{ int i;
  unsigned long R=0;

  for (i=A[0];i;i--)
    R=(10*R+A[i])%X;
  return R;
}
```

## 2.9 Împărțirea a doi vectori

Dacă se dau doi vectori $A$ și $B$ și se cere să se afle câtul $C$ și restul $R$, etapele de parcurs sunt aceleași ca la punctul precedent, numai că operatorii „/” și „%” trebuie implementați de utilizator, ei nefiind definiți pentru vectori. Cu alte cuvinte, după ce „coborâm” la fiecare pas următoarea cifră de la deîmpărțit, trebuie să aflăm cea mai mare cifră $X$ astfel încât împărțitorul să se cuprindă de $X$ ori în restul de la momentul respectiv. Acest lucru se face cel mai comod prin adunări repetate: pornim cu cifra $X = 0$ și o incrementăm, micșorând concomitent restul, până când restul care rămâne este prea mic. Să efectuăm aceeași împărțire, 1997 : 7, considerând că ambele numere sunt reprezentate pe tipul `Huge`.




Cazul cel mai defavorabil (când $X = 9$) presupune 9 scăderi și 10 comparații, cazul cel

mai favorabil (când $X = 0$) presupune numai o comparație, deci cazul mediu presupune 4 scăderi și 5 comparații. Căutarea lui $X$ se poate face și binar, prin înjumătățirea intervalului, ceea ce reduce timpul mediu de căutare la aproximativ 3 comparații și trei înmulțiri, dar codul se complică nejustificat de mult (de cele mai multe ori).

```
void DivideHuge(Huge A, Huge B, Huge C, Huge R)
/* A/B = C rest R */
{ int i;

  R[0]=0;C[0]=A[0];
  for (i=A[0];i;i--)
    { Shl(R,1);R[1]=A[i];
      C[i]=0;
      while (Sgn(B,R)!=1)
        { C[i]++;
          Subtract(R,B);
        }
    }
  while (!C[C[0]] && C[0]>1) C[0]--;
}
```

## 2.10 Extragerea rădăcinii cubice

Vom sări peste prezentarea algoritmului de extragere a rădăcinii pătrate, pe care îl vom lăsa ca temă cititorului, și ne vom îndrepta atenția asupra celui de extragere a rădăcinii cubice, care este puțin mai complicat, dar care poate fi ușor extins pentru rădăcini de orice ordin. Problema este exact cea din enunț, așa că vom porni de la exemplul dat. Să notăm $A = 2.097.152$ și $X = \sqrt[3]{A}$. Cum se află $X$?

O primă variantă ar fi căutarea binară a rădăcinii, prin înjumătățirea intervalului. Inițial se pornește cu intervalul $(1,A)$, deoarece rădăcina cubică se află undeva între 1 și A (evident, încadrarea este mai mult decât acoperitoare; ea ar putea fi mai limitativă, dar nu ar reduce timpul de lucru decât cu câteva iterații). La fiecare pas, intervalul va fi înjumătățit. Cum, probabil că știți deja; se ia jumătatea intervalului, se ridică la puterea a treia și se compară cu $A$. Dacă este mai mare, înseamnă că rădăcina trebuie căutată în jumătatea inferioară a intervalului. Dacă este mai mică, vom continua căutarea în jumătatea superioară a intervalului. Dacă cele două numere sunt egale, înseamnă că am găsit tocmai ce ne interesa. Prima variantă a pseudocodului este:

1: **citește** $A$ cu $N$ cifre
2: $Lo \leftarrow 1, Hi \leftarrow A, X \leftarrow 0$
3: **cât timp** $X = 0$ **execută**
4: $Mid \leftarrow (Lo + Hi)/2$
5: **dacă** $Mid3 < A$ **atunci**
6: $Lo \leftarrow Mid + 1$
7: **altfel dacă** $Mid3 > A$ **atunci**
8: $Hi \leftarrow Mid - 1$
9: **altfel**
10: $X \leftarrow Mid$
11: **sfârșit dacă**
12: **sfârșit cât timp**

În cazul cel mai rău, algoritmul de mai sus efectuează $\log_2 A$ înjumătățiri de interval, fiecare din ele presupunând o adunare, o împărțire la 2 și o ridicare la cub. Dintre aceste operații, cea mai costisitoare este ridicarea la cub, $O(N^2)$. Complexitatea totală este prin urmare $O(N^2 \log_2 A)$. Deoarece $A$ are ordinul de mărime $10^N$, rezultă complexitatea $O(N^3 \log 10) = O(N^3)$, adică mai proastă decât cea cerută (de altfel, un algoritm cu această complexitate nici nu s-ar încadra în timp pentru $N = 1.000$). Dacă timpul ne permite, trebuie să căutăm altă metodă.

În exemplul ales, să observăm că $10^6 \leq A < 10^9$, de unde deducem că $10^2 \leq X < 10^3$. Cu alte cuvinte, $X$ are 3 cifre. În cazul general, dacă $A$ are $N$ cifre, atunci $X$ are $\lceil N/3 \rceil$ cifre (prin $\lceil N/3 \rceil$ se înțelege „cel mai mic întreg mai mare sau egal cu $N/3$"). Care ar putea fi prima cifră a lui $X$ ? Dacă $X$ începe cu cifra 2, atunci $200 \leq X < 300 \implies 8.000.000 \leq 2.097.152 < 27.000.000$, ceea ce este fals. Cu atât mai puțin poate prima cifră a lui $X$ să fie mai mare ca 2. Rezultă că prima cifră a lui $X$ este 1. De altfel, pentru a afla acest lucru, putem să și neglijăm ultimele 6 cifre ale lui $A$. Ne interesează doar prima cifră, cea a milioanelor, iar prima cifră a lui $X$ o alegem în așa fel încât cubul ei să fie mai mic sau egal cu 2.

Ce putem spune despre a doua cifră? Dacă ar fi 3, atunci $130 \leq X < 140 \implies 2.197.000 \leq 2.097.152 < 2.744.000$, fals (deci cifra este cel mult 2). Dacă ar fi 1, atunci $110 \leq X < 120 \implies 1.331.000 \leq 2.097.152 < 1.728.000$, fals. Rezultă că a doua cifră este obligatoriu 2. Analog, putem neglija ultimele trei cifre ale lui $A$, iar a doua cifră a lui $X$ este cel mai mare $C$ pentru care $\overline{1C}^3 \leq 2.097$. Pentru a afla ultima cifră, aplicăm același raționament: Dacă ar fi 9, atunci $X = 129$ și ar rezulta $2.146.688 = X^3 = 2.097.152$, absurd. Dacă considerăm că cifra este 8, atunci calculul se verifică. Am aflat așadar că $X = 128$.

Procedeul general este următorul: dându-se un număr $A$ cu $N$ cifre, îl completăm cu zerouri nesemnificative până când $N$ se divide cu 3 (poate fi necesar să adăugăm maxim două zerouri). Numărul de cifre semnificative ale rădăcinii cubice este $M = N/3$. Aflăm pe rând fiecare cifră, începând cu cea mai semnificativă. Să presupunem că am aflat cifrele $X_M, X_{M-1}, \ldots, X_{K+1}$. Cifra $X_K$ este cea mai mare cifră pentru care numărul $\overline{X_M X_{M-1} \ldots X_{K+1} X_K 00 \ldots 00}^3 \leq A$. Cifra $X_K$ este unică, deoarece există, în general, mai multe cifre care verifică proprietatea cerută, dar una singură este „cea mai mare". O a doua versiune a pseudocodului este deci:

1: **citește** $A$ cu $N$ cifre
2: $X \leftarrow 0, T \leftarrow 0$
3: **cât timp** $N \bmod 3 \neq 0$ **execută**
4: adaugă un 0 nesemnificativ
5: **sfârșit cât timp**
6: **pentru** $i = 1$ **la** $N/3$ **execută**
7: adaugă la $T$ următoarele 3 cifre din $A$
8: adaugă la $X$ cea mai mare cifră astfel încât $X^3 \leq T$
9: **sfârșit pentru**

Să evaluăm complexitatea acestei versiuni. Linia 1 se execută în timp liniar, $O(N)$. Liniile 2-5 se execută în timp constant. Linia 7 presupune adăugarea unei cifre ($O(N)$), iar linia 8 un număr constant de ridicări la cub, așadar înmulțiri ($O(N^2)$). Deoarece liniile 7 și 8 se execută de $N/3$ ori (linia 6), rezultă o complexitate de $O(N^3)$. Iată că nici acest algoritm nu a adus îmbunătățiri și pare și ceva mai greu de implementat. El poate fi totuși modificat pentru a-i scădea complexitatea la $O(N^2)$.

Principalul neajuns al său este efectuarea ridicării la cub, care se face în $O(N^2)$. Dacă am putea să-l aflăm la fiecare pas pe $X^3$ fără a efectua înmulțiri, adică în timp liniar, atunci întregul algoritm ar avea complexitate pătratică. Bineînțeles, prima întrebare care vine pe buzele cititorului este „cum să ridicăm la cub fără să facem înmulțiri?". Să nu uităm însă ceva: că noi nu-l cunoaștem numai pe $X$. Îl cunoaștem și pe $X$ de la pasul anterior, care avea o cifră mai puțin (îl vom boteza $OldX$). Să presupunem că, printr-o metodă oarecare, am reușit să-l ridicăm pe $OldX$ la puterile a doua și a treia (și am obținut numerele $OldX2$ și $OldX3$). Cum putem, cunoscând aceste trei numere, precum și noua cifră ce se va adăuga la sfârșitul lui $X$ (să-i spunem $C$), să-l aflăm pe $X$, pătratul și cubul său? Nu e prea greu:

$$X = 10 \cdot OldX + C \tag{2.6}$$

$$\begin{aligned} X^2 &= (10 \cdot OldX + C)^2 = 100 \cdot OldX^2 + 20 \cdot C \cdot OldX + C^2 \\ &= 100 \cdot OldX^2 + (20 \cdot C) \cdot OldX + C^2 \end{aligned} \tag{2.7}$$

$$\begin{aligned} X^3 &= (10 \cdot OldX + C)^3 = 1.000 \cdot OldX^3 + 300 \cdot C \cdot OldX^2 + 30 \cdot C^2 \cdot OldX + C^3 \\ &= 1.000 \cdot OldX^3 + (300 \cdot C) \cdot OldX^2 + (30 \cdot C^2) \cdot OldX + C^3 \end{aligned} \tag{2.8}$$

Iată așadar că pentru a afla noile valori ale puterilor 1, 2 și 3 ale lui $X$, folosindu-le pe cele vechi, nu avem nevoie decât de adunări și de înmulțiri cu numere mici (de ordinul miilor). Toate aceste operații se fac în timp liniar, deci am reușit să găsim un algoritm pătratic. Iată mai jos sursa C:

```
void FindDigit(Huge L,Huge NewL2,Huge NewL3,
     Huge OldL,Huge OldL2,Huge OldL3,Huge Target)
{ Huge Aux;

  L[1]=10;
  do
    { L[1]--;
      /* Trebuie calculat L^3. Se stiu OldL (L/10)
         si noua cifra L[1]. Deci (OldL*10+L[1])^3=?
         Se aplica binomul lui Newton. */
      AtribHuge(NewL3,OldL3);Shl(NewL3,3);
      AtribHuge(Aux,OldL2);Mult(Aux,300*L[1]);
      Add(NewL3,Aux);
      AtribHuge(Aux,OldL);Mult(Aux,30*L[1]*L[1]);
      Add(NewL3,Aux);
      AtribValue(Aux,L[1]*L[1]*L[1]);
      Add(NewL3,Aux);
    }
  while (Sgn(NewL3,Target)==1);
  /* Aceeasi operatie pentru L^2 */
  AtribHuge(NewL2,OldL2);Shl(NewL2,2);
  AtribHuge(Aux,OldL);Mult(Aux,20*L[1]);
  Add(NewL2,Aux);
  AtribValue(Aux,L[1]*L[1]);
  Add(NewL2,Aux);
  /* Noile valori devin 'vechi' */
  AtribHuge(OldL2,NewL2);
  AtribHuge(OldL,L);
```

```
  AtribHuge(OldL3,NewL3);
}

void CubeRoot(Huge A, Huge X)
{ Huge Target,OldX,OldX2,OldX3,NewX2,NewX3;
  int i;

  /* Se initializeaza vectorii cu 0 (nici o cifra) */
  OldX[0]=OldX2[0]=OldX3[0]=X[0]=0;
  for (i=1;i<=(A[0]+2)/3;i++)
    { AtribHuge(Target,A);
      Shr(Target,3*((A[0]+2)/3-i));
      Shl(X,1);
      FindDigit(X,NewX2,NewX3,OldX,OldX2,OldX3,Target);
    }
}
```

Acum nu mai avem decât să scriem rutinele de intrare/ieșire și programul principal:

```
#include <stdio.h>
#include <mem.h>
#define NMax 1000
typedef int Huge[NMax+3];
Huge A,X; /* A[0] si X[0] indica numarul de cifre */

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

  A[0]=0;
  do A[++A[0]]=(C=fgetc(F))-'0';
  while (C!=EOF);
  A[0]--;
  fclose(F);
  /* Intoarce vectorul pe dos */
  for (i=1;i<=A[0]/2;i++)
    A[i]=(A[i]^A[A[0]+1-i])^(A[A[0]+1-i]=A[i]);
}

void WriteSolution(void)
{ FILE *F=fopen("output.txt","wt");
  int i=X[0];

```

```
25    while (!X[i]) i--;
26    while (i) fputc(X[i--]+'0',F);
27    fclose(F);
28  }
29
30  void main(void)
31  {
32    ReadData();
33    CubeRoot(A,X);
34    WriteSolution();
35  }
```

Pentru a extinde această metodă la rădăcini de orice ordin $K$, trebuie numai să ținem cont de expresia binomului lui Newton:

$$\begin{aligned} X^p &= (10 \cdot OldX + C)^p \\ &= \sum_{i=0}^{p} \mathbf{C}_p^i \cdot 10^i \cdot OldX^i \cdot C^{p-i} \end{aligned} \tag{2.9}$$

Presupunând că avem calculate toate puterile de la 1 la $p$ ale lui $OldX$, se poate calcula noua valoare a lui $X^p$ folosind numai adunări și înmulțiri cu scalari. În felul acesta se pot calcula în timp liniar valorile lui $X, X^2, X^3, \ldots, X^K$.

# Capitolul 3

# Lucrul cu structuri mari de date

De multe ori în practică, atât la concursuri cât și atunci când scriem programe care vehiculează un volum mai mare de date, avem nevoie de structuri de date de mari dimensiuni. După cum se știe, însă, compilatorul Borland Pascal nu permite definirea de structuri de date mai mari de 64 KB. Ce facem dacă, spre exemplu, avem nevoie de un vector cu sute de mii de elemente sau de o matrice pătratică de $400 \times 400$ elemente ?

O primă soluție este să schimbăm limbajul de programare folosit și să ne orientăm spre C / C++ sau alte limbaje în care gestiunea datelor voluminoase se face mai comod pentru utilizator. De fapt, programele scrise „la domiciliu" se scriu mai degrabă în C decât în Pascal, deoarece codul generat este mai eficient. Din nefericire, compilatorul de C este destul de lent, cel puțin în comparație cu cel de Pascal și, deși există destui concurenți care lucrează în C la olimpiadă, Pascal-ul mi se pare o alegere mai bună atunci când timpul de implementare contează decisiv.

În aceste condiții, se impune găsirea unor modalități de a ne supune rigorilor limbajului Pascal și de a „înghesui" cumva datele în memorie. Chiar dacă dispunem de memorie extinsă, segmentarea la 64 KB a datelor ridică destul de multe probleme. Vom trata pe rând câteva cazuri.

## 3.1 Vectori de tip boolean de mari dimensiuni

În Borland Pascal, variabilele de tip logic (`Boolean`) se reprezintă pe un octet. Se știe însă că variabilele booleene pot lua doar două valori, deci un singur bit ar fi suficient pentru a le reprezenta. Motivul acestei aparente „risipe" de memorie este viteza de rulare a programului. Regiștrii lucrează la nivel de octet, iar operațiile la nivel de bit sunt mai

costisitoare din punct de vedere al timpului. În plus, cei șapte biți care ar fi economisiți nu ar putea fi folosiți decât cel mult pentru alte variabile booleene.

În unele cazuri, însă, comprimarea variabilelor logice la un singur bit este necesară, această misiune revenindu-i programatorului. Iată un exemplu de problemă de concurs:

**ENUNȚ**: Se dau $N-1$ numere naturale distincte cuprinse între 1 și $N$. Să se tipărească cel de-al $N$-lea număr (cel care lipsește).

**Intrarea** se face din fișierul `INPUT.TXT`, care conține pe prima linie valoarea lui $N$ ($N \leq 500.000$), iar pe următoarele $N-1$ linii câte un număr cuprins între 1 și $N$.

**Ieșirea**: pe ecran se va tipări numărul care lipsește.

**Exemplu**: Pentru fișierul de intrare:

```
5
3
2
5
1
```

rezultatul tipărit pe ecran va fi 4.

**Complexitate cerută**: $O(N)$.

**Timp de implementare**: 30 minute.

**REZOLVARE**: O soluție foarte elegantă a problemei este următoarea: se știe că suma primelor $N$ numere naturale este

$$S_N = \frac{N(N+1)}{2} \tag{3.1}$$

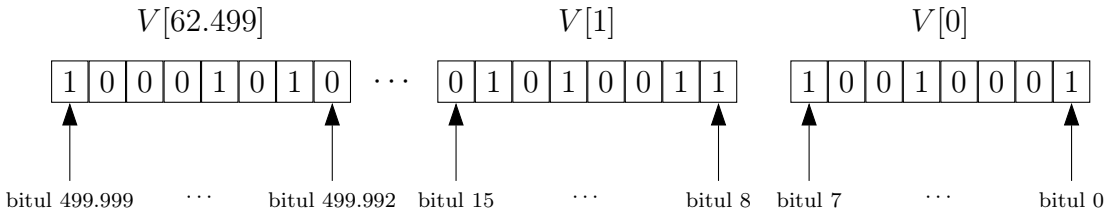
Se calculează suma $S'$ a numerelor din fișierul de intrare, iar numărul lipsă este $K = S_N - S'$. Această metodă are un dezavantaj care o face inutilizabilă: $S_{500.000}$ este aproximativ $125 \cdot 10^9$, adică un număr mult prea mare chiar și pentru tipul de date `Longint`. Ar trebui să se memoreze numerele foarte mari pe un vector de cifre, apoi ar trebui scrise funcțiile pentru adunarea și scăderea unor asemenea numere (vezi capitolul al II-lea), lucru destul de incomod dacă se ține seama de timpul de implementare.

Pentru a memora tot vectorul în memorie este nevoie de $500.000 \times 4 = 2.000.000$ octeți, adică foarte mult, iar găsirea valorii care lipsește ar presupune în cel mai bun

caz o sortare în $O(N \log N)$, care ar depăși complexitatea cerută. Ar mai fi soluția de a căuta pe rând fiecare număr în fișier și de a tipări primul număr pe care nu-l găsim. În cazul cel mai rău, însă, algoritmul ar face $N$ parcurgeri ale fișierului de intrare și $N^2$ comparații, deci ar depăși de asemenea complexitatea cerută, ca să nu mai vorbim de timpul de rulare.

Rezolvarea cea mai la îndemână este de a construi un vector de variabile booleene cu $N$ elemente. Se face apoi citirea datelor și se bifează în vector fiecare număr citit. Apoi se parcurge vectorul și se caută singurul element nebifat. Această versiune face o singură parcurgere a fișierului, adică minimul posibil.

Mai rămâne de văzut cum operăm cu un vector de 500.000 de variabile booleene. Dacă fiecare variabilă ar ocupa un octet, necesarul de memorie ar fi de 500.000 de octeți, care sunt greu de găsit în memoria de bază. Dacă însă alocăm câte un bit pentru fiecare element, necesarul de memorie este 500.000 / 8 = 62.500 octeți, adică o sumă rezonabilă care, mai mult, încape într-un singur segment de memorie și poate fi alocată static fără probleme. Cum se poate accesa și modifica valoarea unui bit din acest vector comprimat? Vom numerota vectorul nostru începând de la 0, deci ultimul său element va fi $V[62.499]$. Primii săi opt biți (biții 0...7, adică octetul $V[0]$) vor fi variabilele logice atașate numerelor 1...8, următorii opt biți (biții 8...15, adică octetul $V[1]$) vor fi variabilele logice atașate numerelor 9...16, și așa mai departe. Ultimii opt biți (biții 499.992...499.999, adică octetul $V[62.499]$) vor fi variabilele logice atașate numerelor 499.993...500.000. În general, bitul cu numărul $X$ indică dacă numărul $X + 1$ a fost găsit sau nu. Iată cum ar putea arăta vectorul la un moment oarecare al citirii din fișier:




Vectorul este reprezentat „pe dos", ceea ce ar putea să pară ciudat la prima vedere. Am făcut însă acest lucru deoarece, în cadrul octetului, biții sunt numerotați în ordine crescătoare de la dreapta spre stânga și am ținut să păstrăm aceeași ordine și pentru numerotarea octeților în vector.

Primul octet semnifică că numărul 1 a fost întâlnit, numerele 2, 3 și 4 nu au fost întâlnite încă, numărul 5 a fost găsit etc. Trebuie acum să vedem cum se face efectiv modificarea vectorului. Inițial toți biții au valoarea 0. Se citește din fișier un număr $X$ și trebuie ca al $X - 1$-lea bit să fie setat pe 1.

Mai întâi trebuie aflat în ce octet se află al $X-1$-lea bit. Se observă imediat că în octetul $Oct = (X-1)$ div 8. De exemplu, biții 0..7 se află în octetul 0, biții 7..15 în octetul 1 ș.a.m.d. Mai e nevoie să știm al câtulea bit este bitul nostru în cadrul octetului. El va fi pe poziția $B = (X-1)$ mod 8 (numărătoarea începe de la 0). În sfârșit, trebuie să setăm bitul respectiv la valoarea 1. În problema de mai sus, singura operație necesară este modificarea unui bit din 0 în 1. Vom trata însă cazul cel mai general, când se cere setarea unui bit la o anumită valoare (0 sau 1) fără a se ști ce valoare a avut el înainte.

Să pornim de la un exemplu particular urmând ca apoi să generalizăm rezultatul. Se cere să se seteze bitul 2 al octetului $A = 00110010$ la valoarea 1. Pentru aceasta, putem folosi o mască logică în care numai bitul 2 este setat pe 1, iar ceilalți sunt 0, adică masca $M = 00000100$. Între octeții $A$ și $M$ se poate face acum un SAU logic:

```
A  00110010 SAU
M  00000100
   --------
B  00110110
```

Se observă că $B = A$ cu excepția bitului 2, care a luat valoarea 1. Acest fapt este ușor de explicat: 0 este element neutru în raport cu operația SAU ($0$ SAU $X = X$, $\forall X$) și deci biții de valoare 0 din $M$ nu modifică biții corespunzători din $A$. În schimb, 1 SAU $X = 1$, $\forall X$, deci bitul 2 din $B$ va lua valoarea 1 indiferent de valoarea bitului corespunzător din $A$.

Revenind la cazul nostru, octetul `Oct` are o valoare oarecare cuprinsă între 0 și 255, iar noi trebuie să-i setăm bitul $B$ ($0 \leq B \leq 7$) la valoarea 1. Masca $M$ va avea toți biții de valoare 0, cu excepția bitului $B$ care va avea valoarea 1. Aceasta înseamnă ca masca $M$ are valoarea $M = 2^B =$ `1 shl B`. Operația pe care o avem de făcut este:

```
1 V[Oct] := V[Oct] or (1 shl B)
```

Să luăm acum un exemplu pentru a vedea cum se setează un bit oarecare la valoarea 0. Fie $A = 01101101$. Se cere să setăm bitul 5 la valoarea 0. De data aceasta, vom folosi o mască în care toți biții sunt 1, mai puțin al 5-lea și vom aplica operația ȘI logic:

```
A 01101101 ȘI
M 11011111
  --------
B 01001101
```

Se observă că $B = A$ cu excepția bitului 5, care a trecut de la valoarea 1 la valoarea 0. Aceasta deoarece 1 este element neutru în raport cu operația ȘI ($1$ ȘI $X = X$, $\forall X$) și deci biții de valoare 1 din $M$ nu modifică biții corespunzători din $A$. În schimb, 0 ȘI $X = 0$, $\forall X$, deci bitul 5 din $B$ va lua valoarea 0 indiferent de valoarea bitului corespunzător din $A$.

Să ne întoarcem la problema noastră: trebuie setat bitul $B$ din octetul `Oct` la valoarea 0. Pentru a construi masca $M$ remarcăm că un octet cu toți biții de valoare 1 este egal cu 255. Din 255 trebuie să scădem (`1 shl B`), deci operația necesară este:

```
1 V[Oct] := V[Oct] and (255 - (1 shl Bit))
```

Mai avem nevoie și să testăm valoarea unui bit. Să luăm de exemplu $A = 00101111$ și să aflăm ce valoare au biții 3 și 7. Folosim măștile $M_3 = 00001000$ și $M_7 = 10000000$, aplicând de fiecare dată operația ȘI logic.

```
A  00101111 ȘI          A   00101111 ȘI
M3 00001000             M7  10000000
   --------                 --------
   00001000                 00000000
```

În ce caz rezultatul poate fi diferit de 0? Șapte dintre biții măștii sunt 0, deci biții corespunzători din B vor fi oricum 0. Cel de-al optulea depinde de bitul respectiv din $A$: dacă acesta este 0, rezultatul va fi 0, dacă este 1, rezultatul va fi diferit de 0. Revenind la problemă, testul care trebuie făcut este:

```
1 V[Oct] and (1 shl Bit) <>  0
```

Cel mai bine este să se creeze o „interfață” care să aibă implementate cele două funcții (setarea și testarea unui bit). După aceasta nu se va mai accesa direct vectorul, ci numai prin intermediul acestor funcții. În felul acesta, dacă vor apărea erori de funcționare din cauza lucrului cu vectorul, vom ști unde să le căutăm.

Inițializarea vectorului se poate face prin două metode: una, mai elegantă, constă în folosirea funcției de atribuire pentru fiecare element al său în parte. A doua, mai rapidă, constă în suprascrierea cu valoarea 0 a tuturor celor 62.500 de elemente ale vectorului $V$, printr-un acces direct al memoriei (procedura `FillChar` din Pascal).

În program, pentru creșterea vitezei, s-au făcut următoarele modificări:

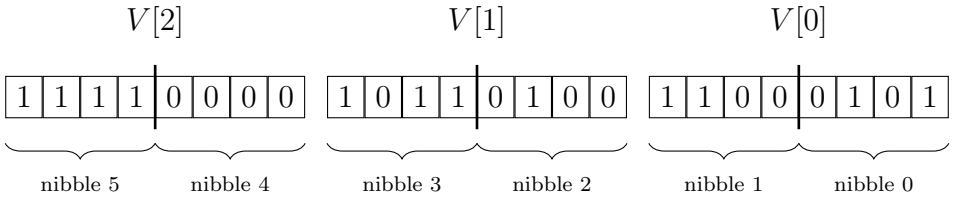
- $(X-1)$ div 8 înseamnă $(X-1)$ div $2^3$, adică `(X-1) shr 3`;
- $(X-1)$ mod 8 reprezintă ultimii 3 biți din reprezentarea binară a lui $X-1$, adică `(X-1) and 7` (deoarece 7 în binar este 00000111).
- `255 - (1 shl A) = 255 xor (1 shl A)`. Cu alte cuvinte, schimbarea unui singur bit din 1 în 0 se poate face atât prin scădere, cât și printr-un SAU exclusiv, a doua variantă fiind mai rapidă.

```
1  program VectorDeBiti;
2  type Vector=array[0..62499] of Byte;
3  var V:Vector;
4      N,X,i:LongInt;
5
6  function GetValue(X:LongInt):Boolean;
7  begin
8    GetValue:=V[Pred(X) shr 3] and (1 shl (Pred(X) and 7))<>0;
9  end;
10
11 procedure SetValue(X:LongInt;B:Boolean);
12 begin
13   if B
14     then V[Pred(X) shr 3]:=V[Pred(X) shr 3] or
15                             (1 shl (Pred(X) and 7))
16     else V[Pred(X) shr 3]:=V[Pred(X) shr 3] and
17                             ($FF xor (1 shl (Pred(X) and 7)));
18 end;
19
20 begin
21   FillChar(V,SizeOf(V),0);
22   Assign(Input,'input.txt');Reset(Input);
23   ReadLn(N);
24   for i:=1 to N-1 do
25     begin
26       ReadLn(X);
27       SetValue(X,True);
28     end;
29   Close(Input);
30   i:=0;
31   repeat Inc(i) until not GetValue(i);
32   WriteLn(i);
33 end.
```

## 3.2 Vectori de dimensiuni mari cu elemente de valori mici

Metoda descrisă la punctul anterior se poate aplica și în alte două cazuri particulare. Dacă elementele vectorului de care avem nevoie pot lua nu doar două valori, ci patru valori (0, 1, 2, 3), atunci ele se pot reprezenta pe doi biți. În concluzie, într-un octet putem „înghesui" patru elemente ale vectorului, iar pe un segment de memorie se pot reprezenta peste 250.000 elemente. În cazul în care elementele vectorului iau valori între 0 și 15, ele se reprezintă pe 4 biți (în limba engeză, grupul de 4 biți poartă un nume special - *nibble*), adică două elemente pe un octet, iar pe un segment încap peste 125.000 elemente. Vom trata numai al doilea caz, lăsându-l pe primul ca temă.

Vom folosi de asemenea un vector $V$ cu 62.500 elemente numerotate cu începere de la 0. Octetul 0 va fi folosit pentru a memora primele două elemente din vectorul dat, octetul 1 va memora al treilea și al patrulea element etc. Octetul 62.499 va memora elementele 124.999 și 125.000. Să vedem de exemplu cum se memorează vectorul (5, 12, 4, 11, 0, 15).




Avem nevoie de aceleași operații ca și în cazul precedent:

- Setarea valorii unui element, indiferent de valoarea sa precedentă;
- Aflarea valorii unui element;
- Inițializarea vectorului.

Pentru a seta valoarea elementului cu numărul $X$ ($1 \leq X \leq 125.000$), trebuie alterat nibble-ul cu numărul $X-1$. În ce octet se află acest nibble? În octetul $Oct = (X-1)$ div 2. Ce poziție ocupă nibble-ul în cadrul octetului? Poziția $Nib = (X-1)$ mod 2. Să luăm acum un caz particular și să vedem cum se face modificarea propriu-zisă, urmând ca apoi să generalizăm.

Se dă octetul $A = 01110010$ și se cere ca nibble-ul 1 (cel din stânga) să fie setat la valoarea 13 (în binar 1101). Se observă că nibble-ul stâng are deja o altă valoare, deci

în primul rând trebuie „curățată zona”, respectiv nibble-ul trebuie adus la valoarea 0. Cum? Probabil ați învățat deja, cu o mască logică. Cum toți patru biții trebuie puși pe 0, iar ceilalți patru trebuie să rămână nealterați, folosim masca $M_1 = 00001111$ și aplicăm operatorul ȘI logic:

```
A   01110010 ȘI
M1  00001111
    --------
A'  00000010.
```

Așadar nibble-ul 0 a rămas nemodificat, iar nibble-ul 1 are valoarea 0. Acum putem aduna pur și simplu nibble-ul de valoare 13. Pentru aceasta, luăm numărul 13 (în binar 1101) și îl deplasăm spre stânga cu 4 poziții, pentru a-l alinia în dreptul nibble-ului 1, după care efectuăm un SAU logic (se poate face și adunarea, dar ea este mai lentă din punct de vedere al calculatorului).

```
A'  00000010 SAU
M2  11010000
    --------
B   11010010
```

Să presupunem acum că voiam să setăm nibble-ul drept la aceeași valoare, 13. Ce aveam de făcut? „Curățam” jumătatea dreaptă a octetului printr-un ȘI logic cu masca $M_1 = 11110000$, apoi făceam un SAU logic cu nibble-ul 13:

```
A   01110010 ȘI
M1  11110000
    --------
A'  01110000 SAU
M2  00001101
    --------
B   01111101
```

Metoda generală este deci următoarea. Se construiește masca $M_1$ cu care se setează pe 0 nibble-ul dorit. Masca se obține scăzând din octetul „plin” 11111111 (zecimal 255, hexazecimal $FF) valoarea 1111 (zecimal 15, hexazecimal $0F), deplasată la stânga cu 4 poziții dacă `Nib = 1`. O primă formă a instrucțiunii de construire a măștii ar fi:

```
if Nib=1 then M1:=$FF - ($0F shl 4)
         else M1:=$FF - $0F
```

Pentru a evita instrucțiunea `if`, destul de mare consumatoare de timp, se poate calcula direct

```
M1:=$FF xor ($0F shl (Nib shl 2))
```

deoarece `Nib shl 2` este 0 dacă $Nib = 0$ și 4 dacă $Nib = 1$. S-a înlocuit scăderea cu operația SAU exclusiv, pentru motivul arătat la punctul 1.

Apoi se adună, după aceeași metodă, valoarea $K$ dorită ($0 \leq K \leq 15$):

```
V[Oct] = (V[Oct] and ($FF xor ($0F shl (Nib shl 2))))
         or (K shl (Nib shl 2))
```

Să vedem cum se află valoarea unui nibble. Să presupunem că se dă octetul $A =$ 01111010 și se cere să se afle nibble-ul 1 (cel din stânga). Pentru aceasta, se face un ȘI logic cu masca $M = 11110000$ (deoarece ultimii patru biți nu interesează), iar rezultatul se deplasează spre dreapta cu 4 biți:

```
A   01111010 ȘI
M   11110000
    --------
A'  01110000 >>>>
    --------
B   00000111
```

Deci nibble-ul stâng are valoarea 7. Revenind la cazul nostru, valoarea $K$ a nibble-ului $Nib$ din octetul $Oct$ este:

```
K = (V[Oct] and ($0F shl (Nib shl 2))) shr (Nib shl 2)
```

Se recomandă și în acest caz scrierea unor funcții și folosirea vectorului $V$ numai prin intermediul acestor funcții. În cazul inițializării vectorului, dacă toate elementele trebuie

puse la valoarea 0, se poate folosi totuși procedura `FillChar`, care e mult mai rapidă decât apelarea funcțiilor noastre pentru fiecare element în parte.

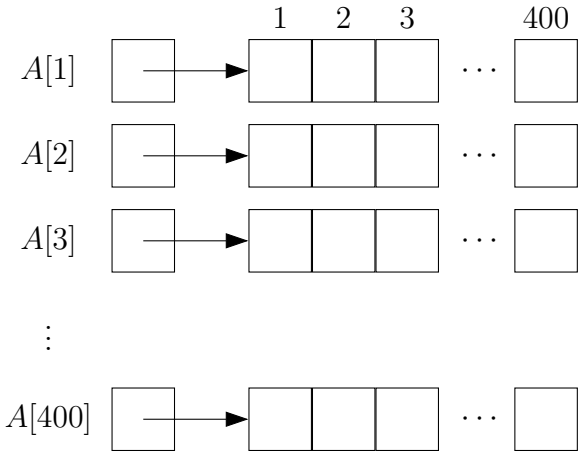
Prezentăm mai jos numai un program demonstrativ despre modul de lucru cu aceste funcții:

```
program Nibbles;
type Vector=array[0..62499] of Byte;
var V:Vector;

function GetValue(X:LongInt):Word;
var Oct,Nib:Word;
begin
  Oct:=Pred(X) shr 1;
  Nib:=Pred(X) and 1;
  GetValue:=(V[Oct] and ($0F shl (Nib shl 2)))
            shr (Nib shl 2);
end;

procedure SetValue(X:LongInt;K:Word);
var Oct,Nib:Word;
begin
  Oct:=Pred(X) shr 1;
  Nib:=Pred(X) and 1;
  V[Oct]:=(V[Oct] and ($FF xor ($0F shl (Nib shl 2))))
          or (K shl (Nib shl 2));
end;

begin
  FillChar(V,SizeOf(V),0);
  SetValue(12345,12);
  SetValue(12346,13); { Doi nibble de pe acelasi octet }
  WriteLn(GetValue(12345),' ',GetValue(12346));
end.
```

## 3.3 Alocarea dinamică a matricelor de mari dimensiuni

Să presupunem că avem nevoie de o matrice cu 400 linii și 400 coloane cu elemente de tip `Integer`. Necesarul de memorie este deci de $400 \times 400 \times 2 = 320.000$ octeți, adică

mult mai mult decât un segment. O cale foarte comodă de a rezolva această dificultate este de a declara matricea drept un vector de pointeri la vectori, ca în figura de mai jos:




Dimensiunea vectorului de pointeri este de $400 \times 4 = 1.600$ octeți (un pointer se reprezintă pe 4 octeți). Dimensiunea unei linii din matrice este de $400 \times 2 = 800$ octeți. Așadar, fiecare structură de date începe pe un segment. Vectorii sunt alocați dinamic la intrarea în program.

Metoda este comod de implementat (practic singura diferență este că elementul de la coordonatele $(i, j)$ din matrice nu va mai fi adresat cu `A[i,j]`, ci cu `A[i]^[j]`) și nu este consumatoare de timp (adresarea unui element mai presupune, pe lângă cele două incrementări datorate indicilor, și o indirectare datorată pointerului).

Iată un exemplu demonstrativ de folosire a acestei structuri de date, care calculează suma numerelor naturale de la 1 la 100.

```
1  program VPV;
2  type Vector=array[1..400] of Integer;
3       Matrix=array[1..400] of ^Vector;
4  var A:Matrix;
5      i:Integer;
6
7  begin
8    for i:=1 to 400 do New(A[i]);
9    A[1]^[1]:=1;
10   for i:=2 to 100 do A[i]^[i]:=A[i-1]^[i-1]+i;
11   WriteLn(A[100]^[100]);
12 end.
```

## 3.4 Fragmentarea matricelor de mari dimensiuni

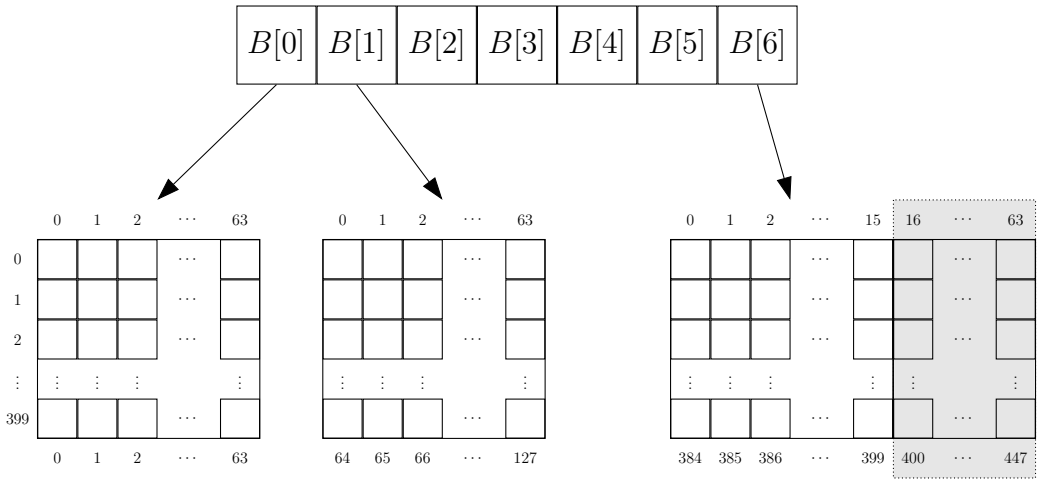
O altă soluție pentru a reprezenta în memorie structuri de date care depășesc un segment este de a le fragmenta în bucăți mai mici, fiecare din acestea nedepășind un segment. Pentru a accesa un element oarecare al structurii, depistăm întâi fragmentul din care el face parte, apoi îl localizăm în cadrul fragmentului.

Să considerăm același exemplu al unei matrice $A$ de dimensiuni $400 \times 400$ cu elemente de tip `Integer`. Spațiul necesar este de 320.000 octeți, adică mai mult de 4 segmente de câte 64 KB și mai puțin de 5. Să spunem că am dori să fragmentăm această matrice în 5 matrice $B[0], B[1], B[2], B[3], B[4]$, fiecare având câte 400 linii și 80 de coloane (vom numerota liniile de la 0 la 399 și coloanele de la 0 la 79). Cele 5 matrice se vor aloca dinamic, fiecare încăpând pe un segment (așadar vectorul $B$ este un vector de pointeri la matrice).

Unde se va regăsi elementul $A[i, j]$ ? Primele 80 de coloane din matricea $A$ se vor afla în matricea $B[0]$, următoarele 80 de coloane din $A$ se vor afla în matricea $B[1]$ ș.a.m.d. Coloana $j$ a matricei $A$ se va afla prin urmare în matricea $B[j \text{ div } 80]$. Linia $i$ va fi aceeași, iar coloana pe care se află elementul $A[i, j]$ în cadrul matricei $B[j \text{ div } 80]$ este $j \bmod 80$.

Acum se vede de ce, pentru a calcula mai rapid câtul și restul împărțirilor, este bine ca numărul de coloane la care se face fragmentarea să fie o putere a lui 2, astfel încât operațiile div și mod să se poată înlocui printr-un `shr` și un `and`. Pentru problema noastră, cea mai rezonabilă cifră este 64, ceea ce înseamnă că avem nevoie de $\lceil 400/64 \rceil = 7$ fragmente (prin rotunjire în sus a rezultatului). De fapt, prin alocarea a 7 segmente (numerotate de la 0 la 6) se creează $7 \times 64 = 448$ de coloane, cu 48 mai mult decât era necesar. Se pierde deci o cantitate de memorie de $48 \times 400 \times 2 = 38.400$ octeți. În cazul în care această memorie nu este vitală, se poate face „risipă”, câștigându-se în schimb viteză.

Iată cum ar arăta matricea fragmentată la 64 de coloane:




Prezentăm mai jos un scurt exemplu de fragmentare, care face același lucru ca și programul de la punctul anterior. S-au scris, ca și în cazurile precedente, două funcții, una pentru modificarea unui element și una pentru aflarea valorii lui. De asemenea, `X div 64` a fost înlocuit peste tot cu `X shr 6`, iar `X mod 64` cu `X and 63`.

```
program Fragment;
type FragType=array[0..399,0..63] of Integer;
     Matrix=array[0..6] of ^FragType;
var B:Matrix;
    i:Integer;

function GetValue(i,j:Integer):Integer;
begin
  GetValue:=B[j shr 6]^[i,j and 63];
end;

procedure SetValue(i,j,Value:Integer);
begin
  B[j shr 6]^[i,j and 63]:=Value;
end;

begin
  for i:=0 to 6 do New(B[i]);
  SetValue(1,1,1);
  for i:=2 to 100 do SetValue(i,i,GetValue(i-1,i-1)+i);
  WriteLn(GetValue(100,100));
end.
```

# Capitolul 4

# Heap-uri și tabele de dispersie

Vom încheia prezentarea structurilor de date mai speciale cu două structuri care se fac folositoare în problemele de căutare și sortare. Ele nu sunt dificil de implementat și se mulează peste orice structuri de date care conțin multe înregistrări ce pot fi ordonate după anumite criterii.

## 4.1 Heap-uri

Să pornim de la o problemă interesantă mai mult din punct de vedere teoretic:

**ENUNȚ**: Un vector se numește $k$-sortat dacă orice element al său se găsește la o distanță cel mult egală cu $k$ de locul care i s-ar cuveni în vectorul sortat. Iată un exemplu de vector 2-sortat cu 5 elemente:

$$V = (6 \quad 2 \quad 7 \quad 4 \quad 10) \tag{4.1}$$

$$V_{sortat} = (2 \quad 4 \quad 6 \quad 7 \quad 10) \tag{4.2}$$

Se observă că elementele 4 și 6 se află la două poziții distanță de locul lor în vectorul sortat, elementele 2 și 7 se află la o poziție distanță, iar elementul 10 se află chiar pe poziția corectă. Distanța maximă este 2, deci vectorul $V$ este 2-sortat. Desigur, un vector $k$-sortat este în același timp și un vector $(k+1)$-sortat, și un vector $(k+2)$-sortat etc., deoarece, dacă orice element se află la o distanță mai mică sau egală cu $k$ de locul potrivit, cu atât mai mult el se va afla la o distanță mai mică sau egală cu $k+1$, $k+2$ etc. În continuare, când vom spune că vectorul este $k$-sortat, ne vom referi la cel mai mic $k$ pentru care afirmația este adevărată. Prin urmare, un vector cu $N$ elemente poate fi

$N$-1 sortat în cazul cel mai defavorabil. Mai facem precizarea că un vector 0-sortat este un vector sortat în înțelesul obișnuit al cuvântului, deoarece fiecare element se află la o distanță egală cu 0 de locul său.

Problema este: dându-se un vector $k$-sortat cu $N$ elemente numere întregi, se cere să-l sortăm într-un timp mai bun decât $O(N \log N)$.

**Intrarea**: Fișierul `INPUT.TXT` conține pe prima linie valorile lui $N$ și $K$ ($2 \leq K < N$ și $N \leq 10.000$), despărțite printr-un spațiu. Pe a doua linie se dau cele $N$ elemente ale vectorului, despărțite prin spații.

**Ieșirea**: În fișierul `OUTPUT.TXT` se vor tipări pe o singură linie elementele vectorului sortat, separate prin spații.

**Exemplu**:

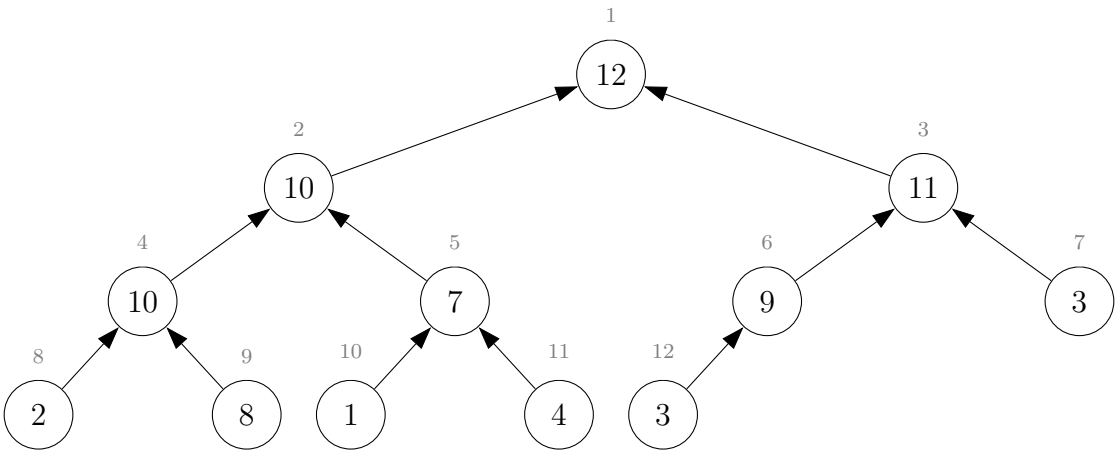
| **INPUT.TXT** | **OUTPUT.TXT** |
|---|---|
| `5 2`<br>`6 2 7 4 10` | `2 4 6 7 10` |

**Timp de implementare**: 45 minute.

**Timp de rulare**: 5 secunde.

**Complexitate cerută**: $O(N \log K)$.

**REZOLVARE**: Vom începe prin a defini noțiunea de *heap*. Un heap (engl. *grămadă*) este un vector care poate fi privit și ca un arbore binar, așa cum se vede în figura de mai jos:




Lângă fiecare nod din arbore se află câte un număr, reprezentând poziția în vector pe

care ar avea-o nodul respectiv. Pentru cazul considerat, vectorul echivalent ar fi:

$$V = (12 \quad 10 \quad 11 \quad 10 \quad 7 \quad 9 \quad 3 \quad 2 \quad 8 \quad 1 \quad 4 \quad 3) \tag{4.3}$$

Se observă că nodurile sunt parcurse de la stânga la dreapta și de sus în jos. O proprietate necesară pentru ca un arbore binar să se poată numi heap este ca toate nivelele să fie complete, cu excepția ultimului, care se completează începând de la stânga și continuând până la un punct. De aici de ducem că înălțimea unui heap cu $N$ noduri este

$$h = \lfloor \log_2 N \rfloor \tag{4.4}$$

(reamintim că înălțimea unui arbore este adâncimea maximă a unui nod, considerând rădăcina drept nodul de adâncime 0). Reciproc, numărul de noduri ale unui heap de înălțime $h$ este:

$$N \in [2^h, 2^{h+1} - 1] \tag{4.5}$$

Tot din această organizare mai putem deduce că tatăl unui nod $k > 1$ este nodul $\lceil k/2 \rceil$, iar fiii nodului $k$ sunt nodurile $2k$ și $2k + 1$. Dacă $2k = N$, atunci nodul $2k + 1$ nu există, iar nodul $k$ are un singur fiu; dacă $2k > N$, atunci nodul $k$ este frunză și nu are nici un fiu. Exemple: tatăl nodului 5 este nodul 2, iar fiii săi sunt nodurile 10 și 11. Tatăl nodului 6 este nodul 3, iar unicul său fiu este nodul 12. Tatăl nodului 9 este nodul 4, iar fii nu are, fiind frunză în heap.

Dar cea mai importantă proprietate a heap-ului, cea care îl face util în operațiile de căutare, este aceea că valoarea oricărui nod este mai mare sau egală cu valoarea oricărui fiu al său. După cum se vede mai sus, nodul 2 are valoarea 10, iar fiii săi - nodurile 4 și 5 - au valorile 10 și respectiv 7. Întrucât operatorul $\geq$ este tranzitiv, putem trage concluzia că un nod este mai mare sau egal decât oricare din nepoții săi și, generalizând, va rezulta că orice nod este mai mare sau egal decât toate nodurile din subarborele a cărui rădăcină este el.

Această afirmație nu decide în nici un fel între valorile a două noduri dispuse astfel încât nici unul nu este descendent al celuilalt. Cu alte cuvinte, nu înseamnă că orice nod de pe un nivel mic are valoare mai mare decât nodurile mai apropiate de frunze. Este cazul nodului 7, care are valoarea 3 și este mai mic decât nodul 9 de valoare 8, care este

totuși așezat mai jos în heap. În orice caz, o primă concluzie care rezultă din această proprietate este că rădăcina are cea mai mare valoare din tot heap-ul.

Structura de heap permite efectuarea multor operații într-un timp foarte bun:

- Căutarea maximului în $O(1)$;
- Crearea unei structuri de heap dintr-un vector oarecare în $O(N)$;
- Eliminarea unui element în $O(\log N)$;
- Inserarea unui element în $O(\log N)$;
- Sortarea în $O(N \log N)$
- Căutarea (singura care nu este prea eficientă) în $O(N)$.

Desigur, toate aceste operații se fac menținând permanent structura de heap a arborelui, adică respectând modul de repartizare a nodurilor pe nivele și „înălțarea" elementelor de valoare mai mare. Este de la sine înțeles că datele nu se vor reprezenta în memorie în forma arborescentă, ci în cea vectorială. Să le analizăm pe rând.

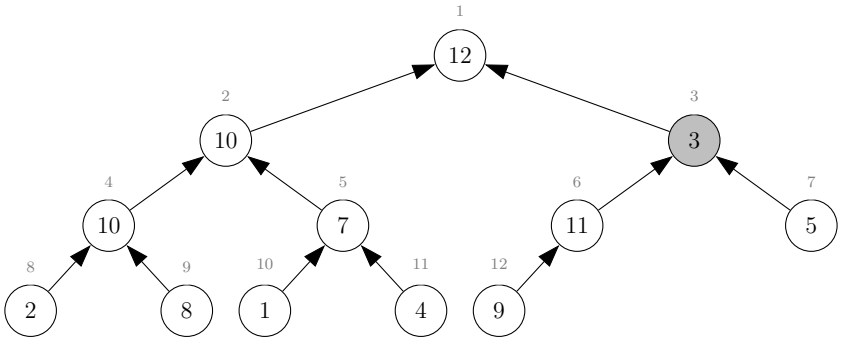
### 4.1.1 Căutarea maximului

Practic operația aceasta nu are de făcut decât să întoarcă valoarea primului element din vector:

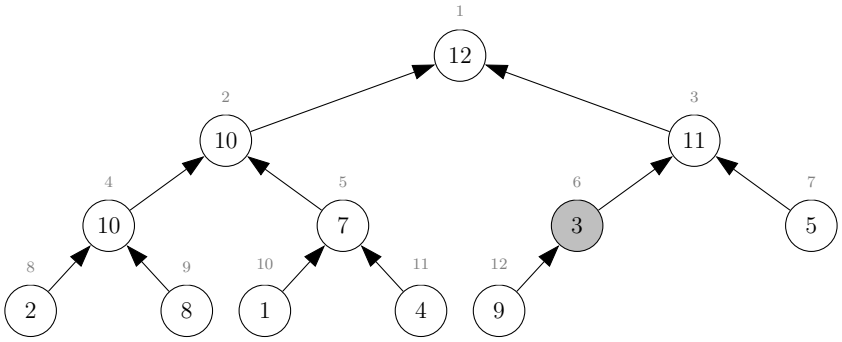
```
typedef int Heap[10001];
void Max(Heap H, int N)
{
  return H[1];
}
```

### 4.1.2 Crearea unei structuri de heap dintr-un vector oarecare

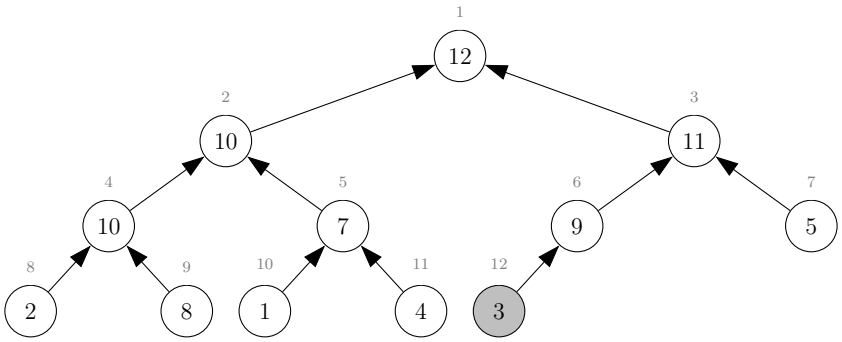
Pentru a discuta acest aspect, vom vorbi mai întâi despre două proceduri specifice heap-urilor, *Sift* (engl. *a cerne*) și *Percolate* (engl. *a se infiltra*). Să presupunem că un vector are o structură de heap, cu excepția unui nod care este mai mic decât unul din fiii săi. Este cazul nodului 3 din figura de mai jos, care are o valoare mai mică decât nodul 6:




Ce e de făcut? Desigur, nodul va trebui coborât în arbore, iar în locul lui vom aduce alt nod, mai exact unul dintre fiii săi. Întrebarea este: care din fiii săi? Dacă vom aduce nodul 7 în locul lui, acesta fiind mai mic decât nodul 6, inegalitatea se va păstra. Trebuie deci să schimbăm nodul 3 cu nodul 6:



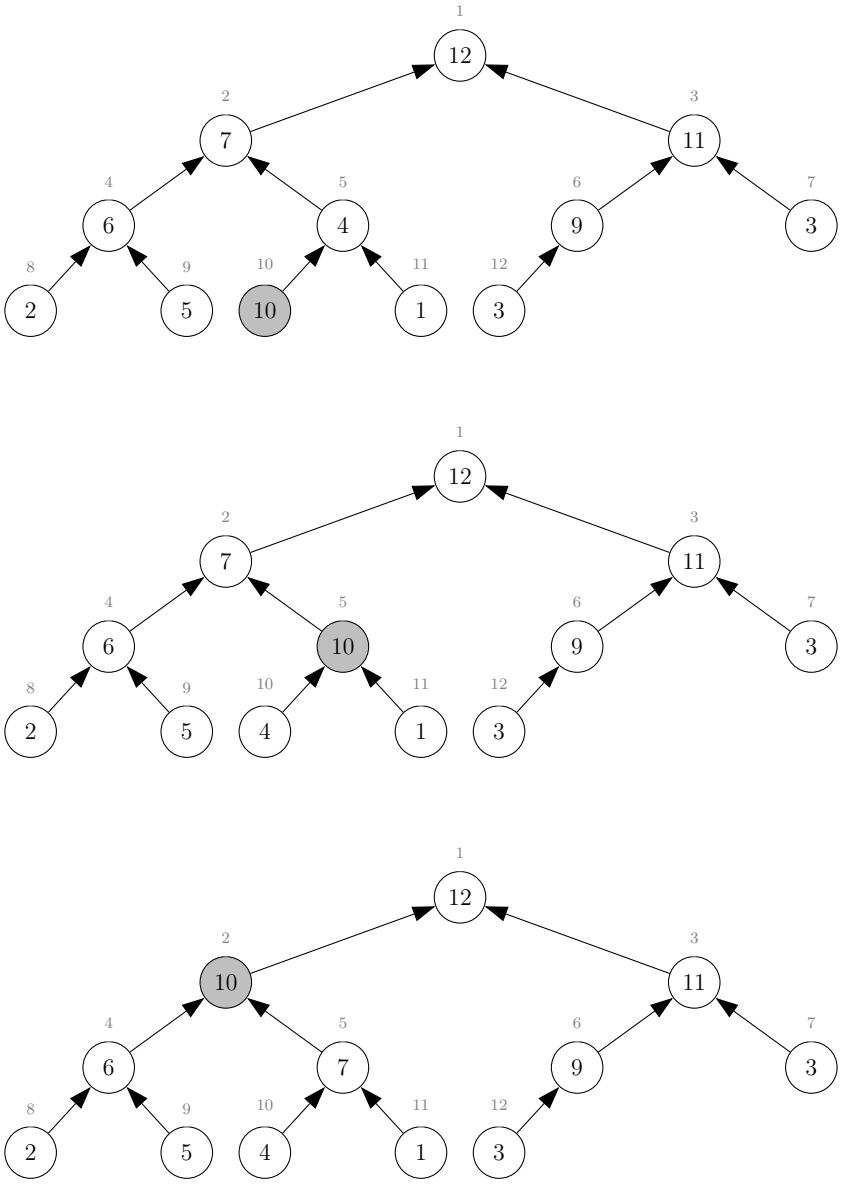

Problema nu este însă rezolvată, deoarece noul nod 6, proaspăt „retrogradat", este încă mai mic decât fiul său, nodul 12. De data aceasta avem un singur fiu, deci o singură opțiune: schimbăm nodul 6 cu nodul 12 și obținem o structură de heap corectă:




Procedura `Sift` primește ca parametri un heap $H$ cu $N$ noduri și un nod $K$ și presupune că ambii subarbori ai nodului $K$ au structură de heap corectă. Misiunea ei este să „cearnă" nodul $K$ până la locul potrivit, interschimbând mereu nodul cu cel mai mare fiu al său până când nodul nu mai are fii (ajunge pe ultimul nivel în arbore) sau până când fiii săi au valori mai mici decât el.

```
void Sift(Heap H, int N, int K)
{ int Son;

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

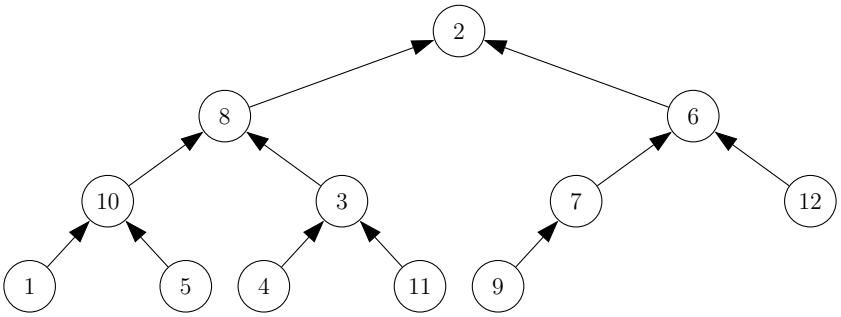
Procedura `Percolate` se va ocupa tocmai de fenomenul invers. Să presupunem că un heap are o „defecțiune” în sensul că observăm un nod care are o valoare mai mare decât tatăl său. Atunci, va trebui să interschimbăm cele două noduri. Este cazul nodului 10 din figura care urmează. Deoarece fiul care trebuie urcat este mai mare ca tatăl, care la rândul lui (presupunând că restul heap-ului e corect) este mai mare decât celălalt fiu al său, rezultă că după interschimbare fiul devenit tată este mai mare decât ambii săi fii. Trebuie totuși să privim din nou în sus și să continuăm să urcăm nodul în arbore fie până ajungem la rădăcină, fie până îi găsim un tată mai mare ca el. Iată ce se întâmplă cu nodul 10:



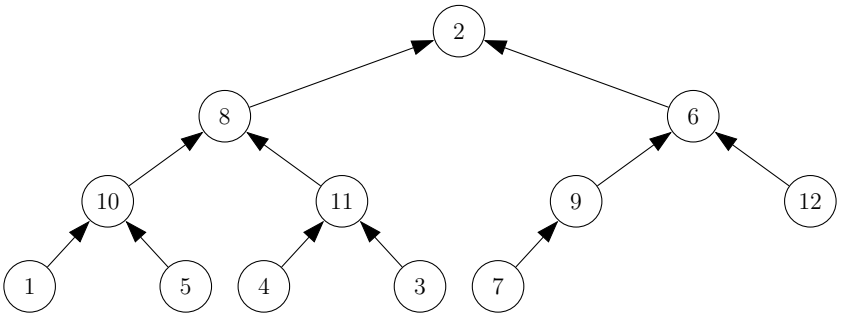

```
void Percolate(Heap H, int N, int K)
{ int Key;

  Key = H[K];
  while ((K>1) && (Key > H[K>>1]))
    { H[K]=H[K>>1];
      K>>=1;
    }
  H[K] = Key;
}
```

Acum ne putem ocupa efectiv de construirea unui heap. Am spus că procedura `Sift` presupune că ambii fii ai nodului pentru care este ea apelată au structură de heap.

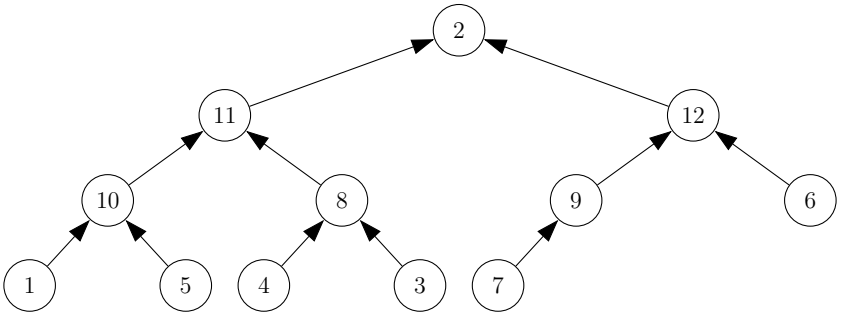
Aceasta înseamnă că putem apela procedura `Sift` pentru orice nod imediat superior nivelului frunzelor, deoarece frunzele sunt arbori cu un singur nod, și deci heap-uri corecte. Apelând procedura `Sift` pentru toate nodurile de deasupra frunzelor, vom obține deja o structură mai organizată, asigurându-ne că pe ultimele două nivele avem de-a face numai cu heap-uri. Apoi apelăm aceeași procedură pentru nodurile de pe al treilea nivel începând de la frunze, apoi pentru cele de deasupra lor și așa mai departe până ajungem la rădăcină. În acest moment, heap-ul este construit. Iată cum funcționează algoritmul pentru un arbore total dezorganizat:



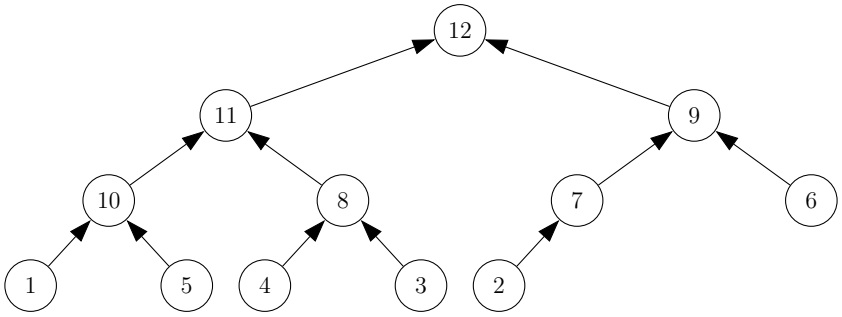

Nivelul frunzelor este organizat




Ultimele două nivele sunt organizate




Ultimele trei nivele sunt organizate



Structură de heap

```
void BuildHeap(Heap H, int N)
{ int i;

  for (i=N/2; i; Sift(H, N, i--));
}
```

S-a apelat căderea începând de la al $N/2$-lea nod, deoarece s-a arătat că acesta este ultimul nod care mai are fii, restul fiind frunze. Să calculăm acum complexitatea acestui algoritm. Un calcul sumar ar putea spune: există $N$ noduri, fiecare din ele se „cerne" pe $O(\log N)$ nivele, deci timpul de construcție a heap-ului este $O(N \log N)$. Totuși nu este așa. Presupunem că ultimul nivel al heap-ului este plin. În acest caz, jumătate din noduri vor fi frunze și nu se vor cerne deloc. Un sfert din noduri se vor afla deasupra lor și se vor cerne cel mult un nivel. O optime din noduri se vor putea cerne cel mult două nivele, și așa mai departe, până ajungem la rădăcina care se află singură pe nivelul ei și va putea cădea maxim $h$ nivele (reamintim că $h = \lfloor \log N \rfloor$). Rezultă că timpul total de calcul este dat de formula:

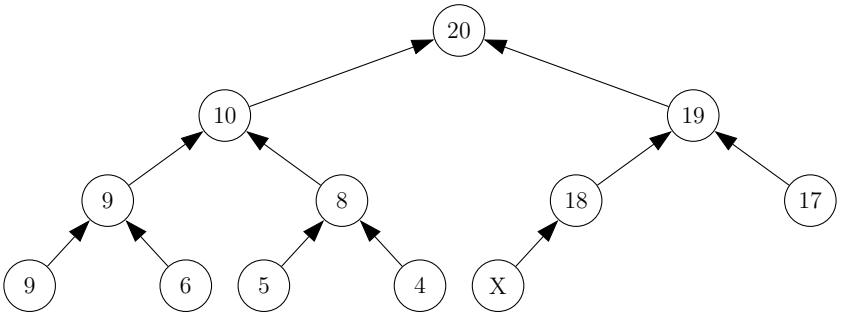
$$O\left(\sum_{k=0}^{\lfloor \log_2 N \rfloor} k \cdot \frac{N}{2^{k+1}}\right) \tag{4.6}$$

Demonstrarea egalității se poate face făcând substituția $N = 2^h$ și continuând calculele. Se va obține tocmai complexitatea $O(2^h)$; lăsăm această verificare ca temă cititorului.

### 4.1.3 Eliminarea unui element

Dacă eliminăm un element din heap, trebuie numai să refacem structura de heap. În locul nodului eliminat s-a creat un gol, pe care trebuie să îl umplem cu un alt nod. Care ar putea fi acela? Întrucât trebuie ca toate nivelele să fie complet ocupate, cu excepția

ultimului, care poate fi gol numai în partea sa dreaptă, rezultă că singurul nod pe care îl putem pune în locul celui eliminat este ultimul din heap. Odată ce l-am pus în gaura făcută, trebuie să ne asigurăm că acest nod „de umplutură" nu strică proprietatea de heap. Deci vom verifica: dacă nodul pus în loc este mai mare ca tatăl său, vom apela procedura `Percolate`. Altfel vom apela procedura `Sift`, în eventualitatea că nodul este mai mic decât unul din fiii săi. Iată exemplul de mai jos:




Să presupunem că vrem să eliminăm nodul de valoare 9, aducând în locul lui nodul de valoare $X$. Însă $X$ poate fi orice număr mai mic sau egal cu 18. Spre exemplu, $X$ poate fi 16, caz în care va trebui urcat deasupra nodului de valoare 10, sau poate fi 1, caz în care va trebui cernut până la nivelul frunzelor. Deoarece căderea și urcarea se pot face pe cel mult $\log N$ nivele, rezultă o complexitate a procedeului de $O(\log N)$.

```
1  void Cut(Heap H, int N, int K)
2  { H[K] = H[N--];
3
4    if ((K>1) && (H[K] > H[K>>1]))
5      Percolate(H, N, K);
6      else Sift(H, N, K)
7  }
```

### 4.1.4 Inserarea unui element

Dacă vrem să inserăm un nou element în heap, lucrurile sunt mult mai simple. Nu avem decât să-l așezăm pe a $N+1$-a poziție în vector și apoi să-l „promovăm" până la locul potrivit. Din nou, urcarea se poate face pe maxim $\log N$ nivele, de unde complexitatea logaritmică.

```
1  void Insert(Heap H, int N, int Key)
2  {
```

```
  H[++N] = Key;
  Percolate(H, N, N);
}
```

### 4.1.5 Sortarea unui vector (heapsort)

Acum, că am stabilit toate aceste lucruri, ideea algoritmului de sortare vine de la sine. Începem prin a construi un heap. Apoi extragem maximul (adică vârful heap-ului) și refacem heap-ul. Cele două operații luate la un loc durează $O(1)+O(\log N) = O(\log N)$. Apoi extragem din nou maximul, (care va fi al doilea element ca mărime din vector) și refacem din nou heap-ul. Din nou, complexitatea operației compuse este $O(\log N)$. Dacă facem acest lucru de $N$ ori, vom obține vectorul sortat într-o complexitate de $O(N \log N)$.

Partea cea mai frumoasă a acestui algoritm, la prima vedere destul de mare consumator de memorie, este că el nu folosește deloc memorie suplimentară. Iată explicația: când heap-ul are $N$ elemente, vom extrage maximul și îl vom ține minte undeva în memorie. Pe de altă parte, în locul maximului (adică în rădăcina arborelui) trebuie adus ultimul element al vectorului, adică $H[N]$. După această operație, heap-ul va avea $N-1$ noduri, al $N$-lea rămânând liber. Ce alt loc mai inspirat decât acest al $N$-lea nod ne-am putea dori pentru a stoca maximul? Practic, am interschimbat rădăcina, adică pe $H[1]$ cu $H[N]$. Același lucru se face la fiecare pas, ținând cont de micșorarea permanentă a heap-ului.

```
void HeapSort(Heap H, int N)
{ int i;

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

### 4.1.6 Căutarea unui element

Această operație este singura care nu poate fi optimizată (în sensul reducerii complexității sub $O(N)$). Aceasta deoarece putem fi siguri că un nod mai mic este descendentul unuia mai mare, dar nu putem ști dacă se află în subarborele stâng sau drept; din această cauză, nu putem face o căutare binară. Totuși, o oarecare îmbunătățire se poate aduce față de căutarea secvențială. Dacă rădăcina unui subarbore este mai mică decât valoarea căutată de noi, cu atât mai mult putem fi convinși că descendenții rădăcinii vor fi și mai mici, deci putem să renunțăm la a căuta acea valoare în tot subarborele. În felul acesta, se poate întâmpla ca bucăți mari din heap să nu mai fie explorate inutil. Pe cazul cel mai defavorabil, însă, parcurgerea întregului heap este necesară. Lăsăm scrierea unei proceduri de căutare pe seama cititorului.

Sperând că am reușit să explicăm modul de funcționare al unui heap, să încercăm să rezolvăm și problema propusă. Chiar faptul că ni se cere o complexitate de ordinul $O(N \log k)$ ne sugerează construirea unui heap cu $O(k)$ noduri. Să ne închipuim că am construi un heap $H$ format din primele $k+1$ elemente ale vectorului $V$. Diferența față de ce am spus până acum este că orice nod va trebui să fie **mai mic** decât fiii săi. Acest heap va servi deci la extragerea minimului.

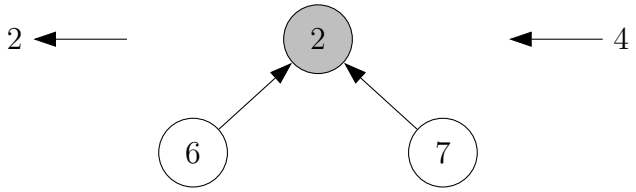
Deoarece vectorul este $k$-sortat, rezultă că elementul care s-ar găsi pe prima poziție în vectorul sortat se poate afla în vectorul nesortat pe oricare din pozițiile $1, 2, \cdots, k+1$. El se află așadar în heap-ul $H$; în plus, fiind cel mai mic, știm exact de unde să-l luăm: din vârful heap-ului. Deci vom elimina acest element din heap și îl vom trece „undeva" separat (vom vedea mai târziu unde). În loc să punem în locul lui ultimul element din heap, însă, vom aduce al $k+2$-lea element din vector și îl vom lăsa să se cearnă. Acum putem fi siguri că al doilea element ca valoare în vectorul sortat se află în heap, deoarece el se putea afla în vectorul nesortat undeva pe pozițiile $1, 2, \cdots, k+2$, toate aceste elemente figurând în heap (bineînțeles că minimul deja extras se exclude din discuție). Putem să mergem la sigur, luând al doilea minim direct din vârful heap-ului.

Vom proceda la fel până când toate elementele vectorului vor fi adăugate în heap. Din acel moment vom continua să extragem din vârful heap-ului, revenind la vechea modalitate de a umple locul rămas gol cu ultimul nod disponibil. Continuăm și cu acest procedeu până când heap-ul se golește. În acest moment am obținut vectorul sortat „undeva" în memorie. De fapt, dacă ne gândim puțin, vom constata că, odată ce primele $k+1$ elemente din vector au fost trecute în heap, ordinea lor în vectorul $V$ nu mai

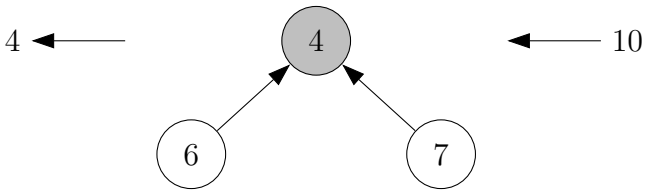
contează, ele putând servi chiar la stocarea minimelor găsite pe parcurs. Pe măsură ce aceste locuri se vor umple, altele noi se vor crea prin trecerea altor elemente în heap. Iată deci cum nici acest algoritm nu necesită memorie suplimentară.

Să urmărim evoluția metodei pe exemplul din enunț:

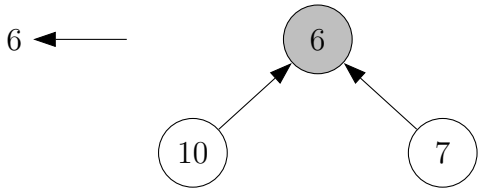
$$V = (6 \quad 2 \quad 7 \quad 4 \quad 10)$$



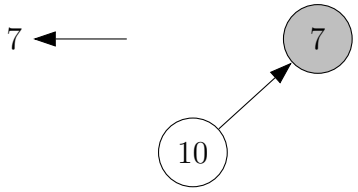

$$V = (2 \quad | \quad 2 \quad 7 \quad 4 \quad 10)$$




$$V = (2 \quad 4 \quad | \quad 7 \quad 4 \quad 10)$$




$$V = (2 \quad 4 \quad 6 \quad | \quad 4 \quad 10)$$



$$V = (2 \quad 4 \quad 6 \quad 7 \quad | \quad 10)$$


$$V = (2 \quad 4 \quad 6 \quad 7 \quad 10)$$

```c
#include <stdio.h>
#include <mem.h>
int V[10001], H[10001], N, K;

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

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

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

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

```
35            if (H[Son]>=H[X]) Son=0;
36          }
37          else Son=0;
38      }
39  }
40
41  void SortVector(void)
42  { int i;
43
44    /* Construieste heap-ul de K+1 elemente */
45    for (i=1; i<=K+1; H[i++]=V[i]);
46    for (i=(K+1) >> 1; i; Sift(i--, K+1));
47
48    for (i=1; i<=N; i++)
49      { V[i] = H[1]; // minimul trece in vector
50        /* Se adauga un element din vector sau din heap */
51        H[1] = (i<=N-K-1) ? V[i+K+1] : H[K+1];
52        /* Daca vectorul s-a terminat, heap-ul incepe
53           sa se micsoreze */
54        if (i>N-K-1) K--;
55        /* Cerne noul element */
56        Sift(1, K+1);
57      }
58  }
59
60  void WriteSolution(void)
61  { FILE *F=fopen("con","wt");
62    int i;
63
64    for (i=1; i<=N; fprintf(F, "%d ", V[i++]));
65    fprintf(F, "\n");
66    fclose(F);
67  }
68
69  void main(void)
70  {
71    ReadData();
72    SortVector();
73    WriteSolution();
74  }
```

## 4.2 Tabele HASH

În multe aplicații lucrăm cu structuri mari de date în care avem nevoie să facem căutări, inserări, modificări și ștergeri. Aceste structuri pot fi vectori, matrice, liste etc. În cazurile mai fericite ale vectorilor, aceștia pot fi sortați, caz în care localizarea unui element se face prin metoda înjumătățirii intervalului, adică în timp logaritmic. Chiar dacă nu avem voie să sortăm vectorul, tot se pot face anumite optimizări care reduc foarte mult timpul de căutare. De exemplu, probabil că mulți dintre cititori au idee despre ce înseamnă *indexarea* unei baze de date. Dacă avem o bază de date cu patru elemente de tip string, și anume

$$B = (bac, zugrav, abac, zarva)$$

putem construi un vector $Ind$ care să ne indice ordinea în care s-ar cuveni să fie așezate cuvintele în vectorul sortat. Ordinea alfabetică (din cartea de telefon) a cuvintelor este: „abac", „bac", „zarva", „zugrav", deci vectorul $Ind$ este:

$$Ind = (3, 1, 4, 2)$$

semnificând că primul cuvânt din vectorul sortat ar trebui să fie al treilea din vectorul $B$, respectiv „abac" și așa mai departe. În felul acesta am obținut un vector sortat, care presupune o indirectare a elementelor. Vectorul sortat este

$$B' = (B(Ind(1)), B(Ind(2)), B(Ind(3)), B(Ind(4)).$$

Această operație se numește indexare. Ce-i drept, construcția vectorului $Ind$ nu se poate face într-un timp mai bun decât $O(N \log N)$, dar după ce acest lucru se face (o singură dată, la începutul programului), căutările se pot face foarte repede. Dacă pe parcurs se fac adăugări sau ștergeri de elemente în/din baza de date, se va pierde câtva timp pentru menținerea indexului, dar în practică timpul acesta este mult mai mic decât timpul care s-ar pierde cu căutarea unor elemente în cazul în care vectorul ar fi neindexat. Nu vom intra în detalii despre indexare, deoarece nu acesta este obiectul capitolului de față.

În unele situații nu se poate face nici indexarea structurii de date. Să considerăm cazul unui program care joacă șah. În esență, modul de funcționare al acestui program se reduce la o rutină care primește o poziție pe tablă și o variabilă care indică dacă

la mutare este albul sau negrul, rutina întorcând cea mai bună mutare care se poate efectua din acea poziție. Majoritatea programelor de șah încep să *expandeze* respectiva poziție, examinând tot felul de variante ce pot decurge din ea și alegând-o pe cea mai promițătoare, așa cum fac și jucătorii umani. Pozițiile analizate sunt stocate în memorie sub forma unei liste simplu sau dublu înlănțuite. Memorarea nu se poate face sub forma unui vector, deoarece numărul de poziții analizate este de ordinul sutelor de mii sau chiar al milioanelor, din care câteva zeci de mii sunt reținute în permanență în memorie.

Să ne închipuim acum următoarea situație. Este posibil ca, prin expandarea unei configurații inițiale a tablei să se ajungă la aceeași configurație finală pe două căi diferite. Spre exemplu, dacă albul mută întâi calul la f3, apoi nebunul la c4, poziția rezultată va fi aceeași ca și când s-ar fi mutat întâi nebunul și apoi calul (considerând bineînțeles că negrul dă în ambele situații aceeași replică). Dacă configurația finală a fost deja analizată pentru prima variantă, este inutil să o mai analizăm și pentru cea de-a doua, pentru că rezultatul (concluzia la care se va ajunge) va fi exact același. Dar cum își poate da programul seama dacă poziția pe care are de gând s-o analizeze a fost analizată deja sau nu?

Cea mai simplă metodă este o scanare a listei de configurații examinate din memorie. Dacă în această listă se află poziția curentă de analizat, înseamnă că ea a fost deja analizată și vom renunța la ea. Dacă nu, o vom analiza acum. Ideea în mare a algoritmului este:

**Procedura 1** Analizează(Poziție $P$)

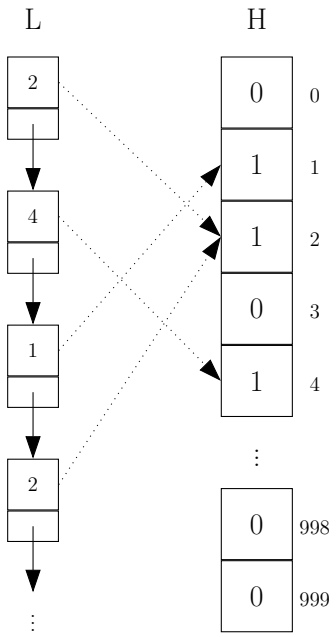
```
1: caută P în lista de poziții deja analizate
2: dacă P nu există în listă atunci
3:     expandează P și află cea mai bună mutare M
4:     adaugă P la lista de poziții analizate
5:     returnează M
6: altfel
7:     returnează valoarea M atașată poziției P găsite în listă
8: sfârșit dacă
```

Nu vom insista asupra a cum se expandează o poziție și cum se calculează efectiv cea mai bună mutare. Noi ne vom interesa de un singur aspect, și anume căutarea unei poziții în listă. Tehnica cea mai „naturală” este o parcurgere a listei de la cap la coadă, comparând pe rând poziția căutată cu fiecare poziție din listă. Dacă lista are memorate $N$ poziții, atunci în cazul unei căutări cu succes (poziția este găsită), numărul mediu de comparații făcute este $N/2$, iar numărul cel mai defavorabil ajunge până la

$N$. În cazul unei căutări fără succes (poziția nu există în listă), numărul de comparații este întotdeauna $N$. De altfel, cazul căutării fără succes este mult mai frecvent pentru problema jocului de șah, unde numărul de poziții posibile crește exponențial cu numărul de mutări. Același număr de comparații îl presupun și ștergerea unei poziții din listă (care presupune întâi găsirea ei) și adăugarea (care presupune ca poziția de adăugat să nu existe deja în listă).

Pentru îmbunătățirea **practică** a acestui timp sunt folosite **tabelele de dispersie** sau **tabelele hash** (engl. *hash = a toca, tocătură*). Menționăm de la bun început că tabelele hash nu au nici o utilitate din punct de vedere teoretic. Dacă suntem rău intenționați, este posibil să găsim exemple pentru care căutarea într-o tabelă hash să dureze la fel de mult ca într-o listă simplu înlănțuită, respectiv $O(N)$. Dar în practică timpul căutării și al adăugării de elemente într-o tabelă hash coboară uneori până la $O(1)$, iar în medie scade foarte mult (de mii de ori).

Iată despre ce este vorba. Să presupunem pentru început că în loc de poziții pe tabla de șah, lista noastră memorează numere între 0 și 999. În acest caz, tabela hash ar fi un simplu vector $H$ cu 1.000 de elemente booleene. Inițial, toate elementele lui $H$ au valoarea `False` (sau 0). Dacă numărul 473 a fost găsit în listă, nu avem decât să setăm valoarea lui $H(473)$ la `True` (sau 1). La o nouă apariție a lui 473 în listă, vom examina elementul $H(473)$ și, deoarece el este `True`, înseamnă că acest număr a mai fost găsit. Dacă dorim ștergerea unui element din hash, vom reseta poziția corespunzătoare din $H$. Practic, avem de-a face cu un exemplu rudimentar de ceea ce se cheamă **funcție de dispersie**, aidcă $h(x) = x$. O proprietate foarte importantă a acestei funcții este injectivitatea; este imposibil ca la două numere distincte să corespundă aceeași intrare în tabelă. Să încercăm o reprezentare grafică a metodei:




Iată primul set de proceduri de gestionare a unui Hash.

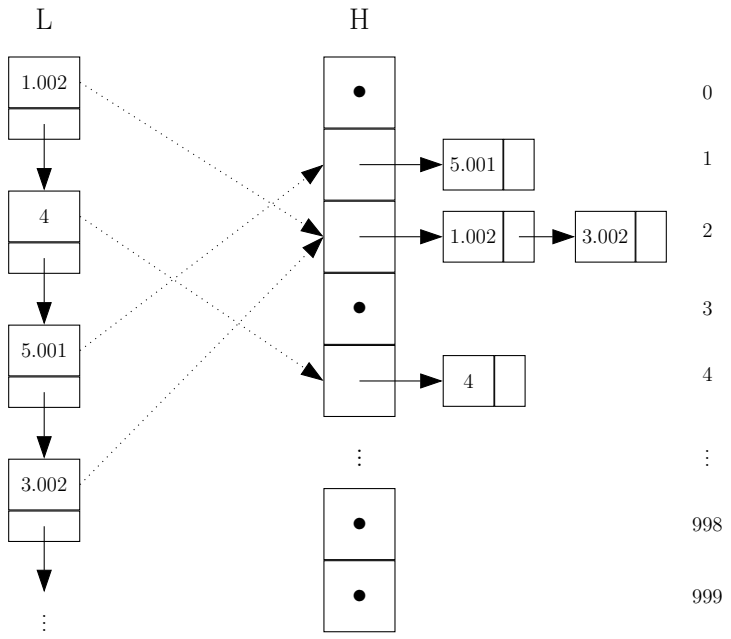
```
#define M 1000 // numarul de "intrari" //
typedef int Hash[M];
typedef int DataType;
Hash H;

void InitHash1(Hash H)
{ int i;

  for (i=0; i<M; H[i++]=0);
}

inline int h(DataType K)
{
  return K;
}

int Search1(Hash H, DataType K)
/* Intoarce -1 daca elementul nu exista in hash
   sau indicele in hash daca el exista */
{
  return H[h(K)] ? h(K) : -1;
}

void Add1(Hash H, DataType K)
{
```

```
26    H[h(K)]=1;
27  }
28
29  void Delete1(Hash H, DataType K)
30  {
31    H[h(K)]=0;
32  }
```

Prin „număr de intrări" în tabelă se înțelege numărul de elemente ale vectorului $H$; în general, orice tabelă hash este un vector. Pentru ca funcțiile să fie cât mai generale, am dat tipului de dată `int` un nou nume - `DataType`. În principiu, tabelele Hash se aplică oricărui tip de date. În realitate, fenomenul este tocmai cel invers: orice tip de date trebuie „convertit" printr-o metodă sau alta la tipul de date `int`, iar funcția de dispersie primește ca parametru un întreg. Funcțiile hash prezentate în viitor nu vor mai lucra decât cu variabile de tip întreg. Vom vorbi mai târziu despre cum se poate face conversia. Acum să generalizăm exemplul de mai sus.

Într-adevăr, cazul anterior este mult prea simplu. Să ne închipuim de pildă că în loc de numere mai mici ca 1.000 avem numere de până la 2.000.000.000. În această situație posibilitatea de a reprezenta tabela ca un vector caracteristic iese din discuție. Numărul de intrări în tabelă este de ordinul miilor, cel mult al zecilor de mii, deci cu mult mai mic decât numărul total de chei (numere) posibile. Ce avem de făcut? Am putea încerca să adăugăm un număr $K$ într-o tabelă cu $M$ intrări (numerotate de la 0 la $M-1$) pe poziția $K \bmod M$, adică $h(K) = K \bmod M$. Care va fi însă rezultatul? Funcția $h$ își va pierde proprietatea de injectivitate, deoarece mai multor chei le poate corespunde aceeași intrare în tabelă, cum ar fi cazul numerelor 1.234 și 2.001.234, ambele dând același rest la împărțirea prin $M = 1.000$. Nu putem avea însă speranța de a găsi o funcție injectivă, pentru că atunci numărul de intrări în tabelă ar trebui să fie cel puțin egal cu numărul de chei distincte. Vrând-nevrând, trebuie să rezolvăm **coliziunile** (sau **conflictele**) care apar, adică situațiile când mai multe chei distincte sunt repartizate la aceeași intrare.

Vom reveni ulterior la oportunitatea alegerii funcției modul și a numărului de 1.000 de intrări în tabelă. Deocamdată vom folosi aceste date pentru a explica modul de funcționare a tabelei hash pentru funcții neinjective. Să presupunem că avem două chei $K_1$ și $K_2$ care sunt repartizate de funcția $h$ la aceeași intrare $X$, adică $h(K_1) = h(K_2) = X$. Soluția cea mai comodă este ca $H(X)$ să nu mai fie un număr, ci o listă liniară simplu sau dublu înlănțuită care să conțină toate cheile găsite până acum și repartizate la aceeași intrare $X$. Prin urmare vectorul $H$ va fi un vector de liste:




$h(x) = x \bmod M$

Să analizăm acum complexitatea noilor proceduri de căutare, adăugare și ștergere. Căutarea nu se va mai face în toată lista, ci numai în lista corespunzătoare din $H$. Altfel spus, o cheie $K$ se va căuta numai în lista $H(h(K))$, deoarece dacă cheia $K$ a mai apărut, ea a fost în mod sigur repartizată la intrarea $H(h(K))$. De aceea, căutarea poate ajunge, în cazul cel mai defavorabil când toate cheile din listă se repartizează la aceeași intrare în hash, la o complexitate $O(N)$. Dacă reușim însă să găsim o funcție care să distrbuie cheile cât mai aleator, timpul de intrare se va reduce de $M$ ori. Avantajele sunt indiscutabile pentru $M = 10.000$ de exemplu.

Întrucât operațiile cu liste liniare sunt în general cunoscute, nu vom insista asupra lor. Prezentăm aici numai adăugarea și căutarea, lăsându-vă ca temă scrierea funcției de ștergere din tabelă.

```
#include <stdio.h>
#include <stdlib.h>
#define M 1000 // numarul de "intrari"
typedef struct _List {
          long P;
          struct _List * Next;
        } List;
typedef List * Hash[M];
Hash H;

```

```
11 void InitHash2(Hash H)
12 { int i;
13
14   for (i=0; i<M; H[i++]=NULL);
15 }
16
17 int h2(int K)
18 {
19   return K%M;
20 }
21
22 int Search2(Hash H, int K)
23 /* Intoarce 0 daca elementul nu exista in hash
24    sau 1 daca el exista */
25 { List *L;
26
27   for (L=H[h2(K)]; L && (L->P != K); L = L->Next);
28   return L!=NULL;
29 }
30
31 void Add2(Hash H, int K)
32 { List *L = malloc(sizeof(List));
33   L->P = K;
34   L->Next = H[h2(K)];
35   H[h2(K)] = L;
36 }
```

Am spus că funcțiile de dispersie sunt concepute să lucreze numai pe date de tip întreg; celelalte tipuri de date trebuie convertite în prealabil la tipuri de date întregi. Iată câteva exemple:

- Variabilele de tip string pot fi transformate în numere în baza 256 prin înlocuirea fiecărui caracter cu codul său ASCII. De exemplu, șirul „abac” poate fi privit ca un număr de 4 cifre în baza 256, și anume numărul (97 98 97 99). Conversia lui în baza 10 se face astfel:

$$X = ((97 \times 256 + 98) \times 256 + 97) \times 256 + 99 = 1.633.837.411 \qquad (4.7)$$

  Pentru stringuri mai lungi, rezultă numere mai mari. Uneori, ele nici nu mai pot fi reprezentate cu tipurile de date ordinale. Totuși, acest dezavantaj nu este supărător, deoarece majoritatea funcțiilor de dispersie presupun o împărțire cu rest, care, indiferent de mărimea numărului de la intrare, produce un număr controlabil.

- Variabilele de tip dată se pot converti la întreg prin formula:

$$A \times 366 + L \times 31 + Z \tag{4.8}$$

unde $A$, $L$ și $Z$ sunt respectiv anul, luna și ziua datei considerate. De fapt, această funcție aproximează numărul de zile scurse de la începutul secolului I. Ea nu are pretenții de exactitate (ca dovadă, toți anii sunt considerați a fi bisecți și toate lunile a avea 31 de zile), deoarece s-ar consuma timp inutil cu calcule mai sofisticate, fără ca dispersia însăși să fie îmbunătățită cu ceva. Condiția care trebuie neapărat respectată este ca funcția de conversie dată $\leftrightarrow$ întreg să fie injectivă, adică să nu se întâmple ca la două date $D_1$ și $D_2$ să li se atașeze același întreg $X$; dacă acest lucru se întâmplă, pot apărea erori la căutarea în tabelă (de exemplu, se poate raporta găsirea datei $D_1$ când de fapt a fost găsită data $D_2$). Pentru a respecta injectivitatea, s-au considerat coeficienții 366 și 31 în loc de 365 și 30. Dacă numărul de zile scurse de la 1 ianuarie anul 1 d.H. ar fi fost calculat cu exactitate, funcția de conversie ar fi fost și surjectivă, dar, după cum am mai spus, acest fapt nu prezintă interes.

- Analog, variabilele de tip oră se pot converti la întreg cu formula:

$$X = (H \times 60 + M) \times 60 + S \tag{4.9}$$

unde $H$, $M$ și $S$ sunt respectiv ora, minutul și secunda considerate, sau cu formula

$$X = ((H \times 60 + M) \times 60 + S) \times 100 + C \tag{4.10}$$

dacă se ține cont și de sutimile de secundă. De data aceasta, funcția este surjectivă (oricărui număr întreg din intervalul 0 - 8.639.999 îi corespunde în mod unic o oră).

- În majoritatea cazurilor, datele sunt structuri care conțin numere și stringuri. O bună metodă de conversie constă în alipirea tuturor acestor date și în convertirea la baza 256. Caracterele se convertesc prin simpla înlocuire cu codul ASCII corespunzător, iar numerele prin convertirea în baza 2 și tăierea în „bucăți” de câte opt biți. Rezultă numere cu multe cifre (prea multe chiar și pentru tipul `longint`), care sunt supuse unei operații de împărțire cu rest. Funcția de conversie trebuie să fie injectivă. De exemplu, în cazul tablei de șah despre care am amintit mai înainte, ea poate fi transformată într-un vector cu 64 de cifre în baza 16, cifra 0 semnificând un pătrat gol, cifrele 1-6 semnificând piesele albe (pion, cal, nebun, turn, regină, rege) iar cifrele 7-12 semnificând piesele negre. Prin trecerea acestui

vector în baza 256, rezultă un număr cu 32 de cifre. La acesta se mai pot adăuga alte cifre, respectiv partea la mutare (0 pentru alb, 1 pentru negru), posibilitatea de a efectua rocada mică/mare de către alb/negru, numărul de mutări scurse de la începutul partidei și așa mai departe.

Vom termina prin a prezenta două funcții de dispersie foarte des folosite.

### 4.2.1 Metoda împărțirii cu rest

Despre această metodă am mai vorbit. Funcția hash este

$$h(x) = x \bmod M \tag{4.11}$$

unde $M$ este numărul de intrări în tabelă. Problema care se pune este să-l alegem pe $M$ cât mai bine, astfel încât numărul de coliziuni pentru oricare din intrări să fie cât mai mic. De asemenea, trebuie ca $M$ să fie cât mai mare, pentru ca media numărului de chei repartizate la aceeași intrare să fie cât mai mică. Totuși, experiența arată că nu orice valoare a lui $M$ este bună.

De exemplu, la prima vedere s-ar putea spune că o bună valoare pentru $M$ este o putere a lui 2, cum ar fi 1.024, pentru că operația de împărțire cu rest se poate face foarte ușor în această situație. Totuși, funcția $h(x) = x \bmod 1.024$ are un mare defect: ea nu ține cont decât de ultimii 10 biți ai numărului $x$. Dacă datele de intrare sunt numere în mare majoritate pare, ele vor fi repartizate în aceeași proporție la intrările cu număr de ordine par, pentru că funcția h păstrează paritatea. Din aceleași motive, alegerea unei valori ca 1.000 sau 2.000 nu este prea inspirată, deoarece ține cont numai de ultimele 3-4 cifre ale reprezentării zecimale. Multe valori pot da același rest la împărțirea prin 1.000. De exemplu, dacă datele de intrare sunt anii de naștere ai unor persoane dintr-o agendă telefonică, iar funcția este $h(x) = x \bmod 1.000$, atunci majoritatea cheilor se vor îngrămădi (termenul este sugestiv) între intrările 920 și 990, restul rămânând nefolosite.

Practic, trebuie ca $M$ să nu fie un număr rotund în nici o bază aritmetică, sau cel puțin nu în bazele 2 și 10. O bună alegere pentru $M$ sunt numerele prime care să nu fie apropiate de nici o putere a lui 2. De exemplu, în locul unei tabele cu $M = 10.000$ de intrări, care s-ar comporta dezastruos, putem folosi una cu 9.973 de intrări. Chiar și această alegere poate fi îmbunătățită; între puterile lui 2 vecine, respectiv 8.192 și 16.384, se poate alege un număr prim din zona 11.000-12.000. Risipa de memorie de circa 1.000-2.000 de intrări în tabelă va fi pe deplin compensată de îmbunătățirea căutării.

### 4.2.2 Metoda înmulțirii

Pentru această metodă funcția hash este

$$h(x) = \lfloor M(x \times A \bmod 1) \rfloor \tag{4.12}$$

Aici $A$ este un număr pozitiv subunitar, $0 < A < 1$, iar prin $x \times A \bmod 1$ se înțelege partea fracționară a lui $x \times A$, adică $x \times A - \lfloor x \times A \rfloor$. De exemplu, dacă alegem $M = 1.234$ și $A = 0,3$, iar $x = 1.997$, atunci avem

$$h(x) = \lfloor 1.234 \times (599,1 \bmod 1) \rfloor = \lfloor 1.234 \times 0,1 \rfloor = 123 \tag{4.13}$$

Se observă că funcția $h$ produce numere între 0 și $M - 1$. Într-adevăr,

$$0 \leq x \times A \bmod 1 < 1 \implies 0 \leq M(x \times A \bmod 1) < M \tag{4.14}$$

În acest caz, valoarea lui $M$ nu mai are o mare importanță. O putem deci alege cât de mare ne convine, eventual o putere a lui 2. În practică, s-a observat că dispersia este mai bună pentru unele valori ale lui $A$ și mai proastă pentru altele. Donald Knuth propune valoarea

$$A = \frac{\sqrt{5} - 1}{2} \approx 0.618034 \tag{4.15}$$

Ca o ultimă precizare necesară la acest capitol, menționăm că funcția de căutare e bine să nu întoarcă pur și simplu 0 sau 1, după cum cheia căutată a mai apărut sau nu înainte între datele de intrare. E recomandabil ca funcția să întoarcă un pointer la zona de memorie în care se află prima apariție a cheii căutate. Vom da acum un exemplu în care această valoare returnată este utilă. Dacă, în cazul prezentat mai sus al unui program de șah, se ajunge la o anumită poziție $P$ după ce albul a pierdut dreptul de a face rocada, această poziție va fi reținută în hash. Reținerea nu se va face nicidecum efectiv (toată tabla), pentru că s-ar ocupa foarte multă memorie. Se va memora în loc numai un pointer la poziția respectivă din lista de poziții analizate. Pe lângă economia de memorie în cazul cheilor de mari dimensiuni, mai există și alt avantaj. Să ne închipuim că, analizând în continuare tabla, programul va ajunge la aceeași poziție $P$, dar în care albul are încă dreptul de a face rocada. E limpede că această variantă este mai promițătoare decât precedenta, deoarece albul are o libertate mai mare de mișcare. Se impune deci fie ștergerea vechii poziții $P$ din listă și adăugarea noii poziții, fie modificarea celei vechi prin

setarea unei variabile suplimentare care indică dreptul albului de a face rocada. Această modificare este ușor de făcut, întrucât căutarea în hash va returna chiar un pointer la poziția care trebuie modificată. Bineînțeles, în cazul în care poziția căutată nu se află în hash, funcția de căutare trebuie să întoarcă `NULL`.

În încheiere, prezentăm un exemplu de funcție de dispersie pentru cazul tablei de șah.

```
#define M 9973 // numarul de "intrari"
typedef struct {
          char b_T[8][8];
            /* tabla de joc, cu 0<= T[i][j] <=12 */
          char b_CastleW, b_CastleB;
            /*  ultimii doi biti ai lui b_CastleW
                indica daca albul are dreptul de a
                efectua rocada mare, respectiv pe cea
                mica. Analog pentru b_CastleB */
          char b_Side;
            /* 0 sau 1, dupa cum la mutare este albul
               sau negrul */
          char b_EP;
            /* 0..8, indicand coloana (0..7) pe care
               partea la mutare poate efectua o
               captura "en passant". 8 indica ca nu
               exista aceasta posibilitate */
          int b_NMoves;
            /* Numarul de mutari efectuate */
        } Board;
Board B;

int h3(Board *B)
{ int i,j;
  /* Valoarea initiala a lui S este un numar pe 17 biti care
     inglobeaza toate variabilele suplimentare pe langa T.
     S se va lua apoi modulo M */
  long S = (B->b_NMoves           /* 8 biti */
          +(B->b_CastleW << 8)    /* 2 biti */
          +(B->b_CastleB << 10)   /* 2 biti */
          +(B->b_Side << 12)      /* 1 bit */
          +B->b_EP<<13) % M;      /* 4 biti */

  for (i=0; i<=7; i++)
    for (j=0; j<=7; j++)
      S=(16*S+B->b_T[i][j])%M;

```

```
38    return S;
39  }
```

# Capitolul 5

# Despre algoritmi exponențiali și îmbunătățirea lor

Trebuie să spunem de la bun început că cea mai bună îmbunătățire care i se poate aduce unui algoritm în timp exponențial care rezolvă o anumită problemă este evitarea lui, adică găsirea - acolo unde este posibil - a unui algoritm polinomial care să rezolve aceeași problemă. Există cazuri în care acest lucru nu este posibil; în această situație, însă, orice îmbunătățire nu este decât o metodă de a ascunde gunoiul sub covor. Un algoritm exponențial rămâne exponențial, iar îmbunătățirile aduse îl pot face să meargă de două, de trei, de zece ori mai repede, dar nu-l pot transforma într-un algoritm polinomial. Creșterea cu două - trei unități a dimensiunii datelor de intrare va anihila saltul de la un calculator 486 la un Pentium.

În multe situații, nu se cunoaște nici un algoritm care să funcționeze în timp util și să furnizeze soluția optimă a unei probleme. În asemenea cazuri, dacă optimalitatea nu este strict necesară, se renunță la ea și se caută algoritmi care să producă soluții cât mai apropiate de cea optimă și care să meargă mult mai repede. Este intuitiv că, dacă impunem limite mai dure în ceea ce privește timpul de rulare al algoritmului, vom avea șanse mai mici să găsim o soluție apropiată de optim. Nu vom discuta în această carte despre cum se poate realiza un echilibru între abaterea soluției de la optim și timpul necesar pentru a o produce. Genul acesta de dileme apar și în cadrul concursului de informatică, dar acolo timpul nu permite o analiză laborioasă a problemei. Noi vom indica numai modul în care se poate „inventa" un algoritm polinomial în locul unuia exponențial și o metodă prin care acest algoritm poate fi îmbunătățit pentru ca rezultatele pe care acesta le scoate să nu fie departe de adevăr. Aceasta este una din tendințele din ultimii ani de la concursurile de informatică. Deoarece toți elevii cunosc problemele „clasice"

care s-ar putea da la concurs, se încearcă departajarea lor prin urmărirea modului în care ei se adaptează la probleme fără o soluție eficientă cunoscută.

Să pornim de la o problemă celebră, cea a comis-voiajorului.

**ENUNȚ**: Se dă un graf complet orientat cu $N \leq 30$ noduri. Fiecare muchie are un cost cuprins între 1 și 100. Se cere să se determine un ciclu hamiltonian de cost minim. Un ciclu hamiltonian este ciclul care parcurge fiecare nod exact o dată. Costul unui ciclu este suma costurilor muchiilor componente.

**Intrarea**: Datele de intrare se găsesc în fișierul `INPUT.TXT`, sub următoarea formă:

```
N
A[1,1] A[1,2] ... A[1,N]
...
A[N,1] A[N,2] ... A[N,N]
```

unde `A[i,j]` este lungimea muchiei care iese din nodul $i$ și intră în nodul $j$. Se garantează că $A[i,i] = 0, \forall 1 \leq i \leq N$.

**Ieșirea** se va face în fișierul `OUTPUT.TXT` pe două linii. Pe prima linie se va tipări costul minim găsit, iar pe a doua traseul parcurs ($N$ numere separate prin spații).

**Exemplu**:

| INPUT.TXT | OUTPUT.TXT |
|---|---|
| 4<br>0 3 4 1<br>3 0 2 3<br>5 2 0 6<br>8 1 3 0 | 9<br>1 4 2 3 |

**Timp de execuție**: 30 secunde

**Timp de implementare**: 30 minute

**REZOLVARE**: Problema este arhicunoscută și de asemenea este arhicunoscut faptul că ea nu admite o rezolvare polinomială. Totuși, dacă această problemă vă este dată la un concurs, nu poate constitui o scuză în fața comisiei argumentul că problema nu este polinomială. Ea trebuie făcută să meargă cât mai bine. Spre deosebire de laboratoarele NASA, unde orice greșeală într-o linie de cod poate distruge un modul spațial cu echipaj cu tot, la olimpiadă nu se merge pe principiul „totul sau nimic". Fiecare punct câștigat este bun câștigat.

Precizăm că metodele de mai jos se aplică mai degrabă în cazurile în care se cere o soluție optimă, dar se oferă punctaje parțiale și în cazul în care concurentul oferă o soluție cât de cât apropiată de cea optimă. Există șanse ca metodele de mai jos să asigure găsirea chiar a soluției optime pentru mare parte din teste, dar aceste șanse variază de la o problemă la alta.

În primul rând, care este deosebirea fundamentală dintre un algoritm backtracking și unul greedy? Algoritmul backtracking analizează pe rând fiecare soluție posibilă și o alege pe cea mai bună. În felul acesta, el nu poate scăpa soluția optimă. Din nefericire, în multe cazuri spațiul soluțiilor crește exponențial cu dimensiunea datelor de intrare; în aceste situații algoritmii backtracking nu mai sunt practici. În schimb, algoritmii greedy (engl. *greedy = lacom*) fac o parcurgere a datelor de intrare și, la fiecare pas al acestei parcurgeri, aleg o parte (destul de mică) din soluțiile posibile, iar pe restul le „aruncă". La pasul următor se aleg o parte din soluțiile rămase și așa mai departe, până când în final rămâne o singură soluție care se tipărește.

Criteriul în funcție de care se face trierea soluțiilor este cheia unui algoritm greedy. Dacă acest criteriu poate garanta că la fiecare pas al algoritmului soluția optimă (sau cel puțin una din soluțiile optime, dacă pot exista mai multe) rămâne între soluțiile care sunt păstrate, atunci algoritmul greedy funcționează perfect. Demonstrația este ușoară: soluția optimă nu este „aruncată" niciodată, iar la sfârșitul algoritmului rămâne o singură soluție, de unde rezultă că soluția rămasă este tocmai cea optimă. Asemenea cazuri de algoritmi pentru care s-a demonstrat că ei funcționează sunt: algoritmii lui Kruskal și Prim pentru găsirea arborelui parțial de cost minim al unui graf, algoritmul lui Dijkstra pentru determinarea drumurilor de cost minim de la un nod la toate celelalte într-un graf ș.a.m.d.

Putem extinde aceste noțiuni și la domeniul jocurilor logice. De exemplu, jocul Nim (pe care probabil îl cunoașteți cu toții) are o strategie sigură de câștig pentru anumite poziții, iar pentru celelalte se poate demonstra că nu există nici o strategie de câștig. În cazul în care strategia există, jucătorului i se oferă o mutare care îl duce spre victorie. Ce este în fond această mutare? Tocmai un criteriu de a tria anumite configurații, favorabile jucătorului, și de a le ignora pe celelalte.

Există însă probleme pentru care nu s-a găsit (iar uneori s-a și demonstrat că nu există) nici un algoritm greedy. Continuând paralela cu jocurile logice, există jocuri care nu au nici o strategie de câștig pentru unul din jucători. Este cazul jocului de șah. Pentru unele asemenea probleme (cum ar fi cea de față, a comis-voiajorului), care au aplicații largi în diferite domenii practice, se investesc sume mari în cercetare pentru a se găsi

algoritmi cât mai buni care să funcționeze într-un timp convenabil.

O situație asemănătoare apare la concursul de informatică, unde se cunoaște de la început timpul pus la dispoziție (atât cel pentru implementare, cât și cel pentru execuție) și se urmărește obținerea unui punctaj cât mai mare. Iată câteva metode destul de eficiente.

## 5.1 „Omorârea" backtracking-ului

Sintagma aceasta oarecum ilară, care stârnește mila pentru bietul backtracking, se aude foarte des pe la ieșirea din sălile de concurs, atunci când problema a fost mai „ciudată", în sensul că foarte puțini concurenți au descoperit vreo soluție eficientă la ea. Ce este de fapt „backtracking-ul omorât" și în ce situații este el preferabil?

Problema comis-voiajorului sub diverse forme sau alte probleme exponențiale au fost propuse în anii trecuți spre rezolvare la concursuri. Dacă vrem să aflăm soluția optimă (de cost minim), neavând altă soluție la îndemână, trebuie să recurgem la backtracking. Backtracking-ul, după cum se știe, examinează pe rând fiecare posibilă soluție. În cazul nostru, backtracking-ul nu are altceva de făcut decât să genereze pe rând toate permutările mulțimii $1, 2, \ldots, N$. Considerând fiecare permutare ca fiind un posibil ciclu hamiltonian (știm sigur că oricărei permutări îi corespunde un ciclu hamiltonian, deoarece graful este complet), mai trebuie doar să calculăm costul fiecărei permutări și să o afișăm pe cea de cost minim. Până aici, nimic deosebit. Iată și sursa Pascal:

```
1  program Hamilton;
2  {$B-,I-,R-,S-}
3  const NMax=30;
4  type Vector=array[1..NMax] of Integer;
5       Matrix=array[1..NMax,1..NMax] of Integer;
6  var A:Matrix;
7      Route,BestRoute:Vector;
8      Seen:set of 1..NMax;
9      N,Cost,MinCost:Integer;
10
11 procedure ReadData;
12 var i,j:Integer;
13 begin
14   Assign(Input,'input.txt');Reset(Input);
15   ReadLn(N);
16   for i:=1 to N do
```

```
17      begin
18        for j:=1 to N do Read(A[i,j]);
19        ReadLn;
20      end;
21    Close(Input);
22  end;
23
24  procedure Bkt(Level,Cost:Integer);
25  var i:Integer;
26  begin
27    if Level=N+1
28      then begin
29             Inc(Cost,A[Route[N],1]);
30             if Cost<MinCost
31               then begin
32                      BestRoute:=Route;
33                      MinCost:=Cost;
34                    end;
35           end
36      else if Cost<MinCost
37             then for i:=1 to N do
38               if not (i in Seen)
39                 then begin
40                        Seen:=Seen+[i];
41                        Route[Level]:=i;
42                        Bkt(Level+1,Cost+A[Route[Level-1],i]);
43                        Seen:=Seen-[i];
44                      end;
45  end;
46
47  procedure WriteSolution;
48  var i:Integer;
49  begin
50    Assign(Output,'output.txt');Rewrite(Output);
51    WriteLn(MinCost);
52    for i:=1 to N do Write(BestRoute[i],' ');
53    WriteLn;
54    Close(Output);
55  end;
56
57  begin
58    ReadData;
59    Route[1]:=1;
60    Seen:=[1];
61    MinCost:=MaxInt;
```

```
62    Bkt(2,0);
63    WriteSolution;
64  end.
```

Programul sub această formă nu se încadrează în timp nici măcar pentru $N = 15$ (cifra depinde și de calculatorul folosit pentru testare, dar nu variază cu mai mult de două-trei nivele; să spunem cu generozitate că pe un calculator performant programul ar putea merge până la $N = 18$ sau 20). Pe de altă parte, ideea în sine (algoritmul) de rezolvare nu mai poate fi mult îmbunătățită. Și cu toate acestea, programul trebuie să meargă până la $N = 30$. Ce putem face?

Desigur, nu există o rezolvare elegantă. Putem însă încerca fel de fel de metode de a trișa. Deoarece nu avem timp să examinăm toate soluțiile, trebuie să renunțăm la o parte din ele, cu riscul ca printre ele să se afle tocmai soluția căutată. Una dintre tehnici este omorârea backtracking-ului. După numărul și tipurile soluțiilor pe care le cer, algoritmii backtracking ar putea fi împărțiți în mai multe categorii:

- Cei care furnizează o singură soluție;
- Cei care, pe baza unei funcții care atașează un cost fiecărei soluții, furnizează soluția de cost minim;
- Cei care furnizează toate soluțiile.

Backtracking-ul omorât se aplică celui de-al doilea tip de cerințe. Putem face în așa fel încât să oprim programul exact la expirarea timpului permis pentru rulare și să afișăm cea mai bună soluție găsită până la momentul respectiv. Dacă am fost norocoși (termenul este cel mai potrivit în această situație), atunci programul nostru a apucat, în timpul pe care i l-am permis, să găsească soluția optimă. Dacă nu, putem totuși spera că a fost găsită o soluție cât de cât apropiată de cea optimă, pentru care putem eventual să primim măcar o parte din punctaj. După cum se vede, omorârea backtracking-ului nu promite marea cu sarea, dar este un artificiu binevenit, pentru că există trei variante:

- Programul se încadrează în timp, caz în care backtracking-ul omorât nu aduce nimic în plus;
- Programul nu se încadrează în timp, dar soluția optimă este găsită în timp, caz în care backtracking-ul simplu nu furnizează nici o soluție (și de obicei este oprit cu Ctrl-Break), pe când backtracking-ul omorât furnizează soluția;

- Programul nu se încadrează în timp și nici soluția optimă nu este găsită în timp, caz în care backtracking-ul simplu nu furnizează nici o soluție, pe când backtracking-ul omorât furnizează o soluție eventual apropiată de optim.

Din experiență se poate spune că membrii comisiei de corectare acordă jumătate din punctele pentru un test mai degrabă atunci când li se oferă o soluție neoptimă decât atunci când li se oferă o soluție optimă într-un timp depășit. Adesea programul este oprit îndată ce timpul de rulare expiră, și părerea autorului e că e mai bine așa.

„Omorârea" nu se pretează la celelalte două versiuni de backtracking. Atunci când există o singură soluție, nu se mai pune problema de a găsi una apropiată de ea. Ori găsim soluția, ori nimic. De exemplu, nu are sens să rezolvăm problema de mai sus cu un backtracking omorât dacă știm sigur că nu se acordă punctaje parțiale pentru soluții neoptime (dar în general acest lucru nu se știe sigur...). Rămâne bineînțeles posibilitatea de a opri programul imediat ce soluția a a fost găsită. În cazul în care se cer toate soluțiile, de asemenea programul nu poate fi oprit. Atunci când se poate, merită calculat numărul de soluții, pentru ca imediat ce am găsit toate soluțiile, să oprim programul. Putem aplica backtracking-ul omorât numai dacă știm că se acordă punctaj și pentru afișarea unei părți din soluții.

Mai rămâne de stabilit cum anume se face „omorârea" backtracking-ului (și de fapt a oricărui program). Există mai multe metode. Prima, asupra căreia nu vom insista deoarece ea are „efecte secundare" și, în plus, este foarte lentă, constă din două etape:

- Se setează ceasul sistem la ora 00:00:00 (miezul nopții) cu procedura `SetTime` din unitatea Dos a compilatorului Borland Pascal;
- Periodic se testează ora cu procedura `GetTime` și se oprește programul atunci când se apropie „ora critică" (în cazul nostru 00:00:30).

După cum se vede, marele neajuns al acestei metode este că „dă peste cap" ceasul sistem (lucru dezastruos mai ales pentru cei care vin la concurs fără ceas...). În afară de aceasta, procedurile `GetTime` și `SetTime` apelează la rândul lor întreruperile DOS, ceea ce consumă mult din timpul care și așa este limitat.

A doua metodă, care este mai rapidă și nu lasă urme, constă în captarea **întreruperii 8**, adică a **timer**-ului. Timer-ul este o rutină care se apelează automat la fiecare 55 de milisecunde, deci cam de 18,2 ori pe secundă. În principiu, ea nu face nimic altceva decât să incrementeze ceasul sistem cu 55 ms. Pe lângă aceasta, însă, putem adăuga și propriul

nostru cod, folosind procedurile Pascal `GetIntVec` și `SetIntVec`. Trebuie doar să avem grijă ca timer-ul scris de noi să-l apeleze și pe cel vechi, altfel ceasul sistem se va opri și cine știe ce altceva se mai poate întâmpla. Vom declara deci o variabilă `Time` care va fi decrementată la fiecare apel al întreruperii de ceas. Înainte de a intercepta întreruperea 8, vom inițializa variabila cu valoarea maximă dorită. Știm că timer-ul se apelează de 18,2 ori pe secundă, deci dacă limita de timp pentru un test este de 30 de secunde, valoarea inițială pentru variabila `Time` ar putea fi $18,2 \times 30 = 546$. Este bine să nu calculăm însă timpul la limită, deoarece avem nevoie de câteva fracțiuni și pentru tipărirea soluției în fișier, și poate pur și simplu ceasul comisiei de corectare o ia puțin înainte. De aceea, e mai sigură înmulțirea cu 17 în loc de 18,2.

În momentul în care, prin decrementări succesive, `Time` a ajuns la valoarea 0 (sau la o valoare negativă), programul trebuie oprit. Acest lucru presupune ieșirea din procedura de backtracking, afișarea soluției și restaurarea vechii întreruperi 8, pentru ca programul să nu lase „urme”. Iată deci o versiune a programului Pascal care va fi extrem de punctuală...

```
1  program Hamilton;
2  {$B-,I-,R-,S-}
3  uses Dos;
4  const NMax=30;
5        TimeLimit=30; { secunde }
6
7  type Vector=array[1..NMax] of Integer;
8       Matrix=array[1..NMax,1..NMax] of Integer;
9  var A:Matrix;
10     Route,BestRoute:Vector;
11     Seen:set of 1..NMax;
12     N,Cost,MinCost:Integer;
13     Time:Integer;  { Contorul }
14     OldTimer:procedure;
15
16 procedure MyTimer; interrupt;
17 { Se executa la fiecare 55 ms }
18 begin
19   Dec(Time);    { Ne facem treaba... }
20   Inline($9C); { ...pushf... }
21   OldTimer;     { ...si executam si vechiul timer }
22 end;
23
24 procedure ReadData;
25 var i,j:Integer;
```

```
26 begin
27   Assign(Input,'input.txt');Reset(Input);
28   ReadLn(N);
29   for i:=1 to N do
30     begin
31       for j:=1 to N do Read(A[i,j]);
32       ReadLn;
33     end;
34   Close(Input);
35 end;
36
37 procedure Bkt(Level,Cost:Integer);
38 var i:Integer;
39 begin
40   if Level=N+1
41     then begin
42            Inc(Cost,A[Route[N],1]);
43            if Cost<MinCost
44              then begin
45                     BestRoute:=Route;
46                     MinCost:=Cost;
47                   end;
48          end
49     else if (Time>0) and (Cost<MinCost)
50            then for i:=1 to N do
51              if not (i in Seen)
52                then begin
53                       Seen:=Seen+[i];
54                       Route[Level]:=i;
55                       Bkt(Level+1,Cost+A[Route[Level-1],i]);
56                       Seen:=Seen-[i];
57                     end;
58 end;
59
60 procedure WriteSolution;
61 var i:Integer;
62 begin
63   Assign(Output,'output.txt');Rewrite(Output);
64   WriteLn(MinCost);
65   for i:=1 to N do Write(BestRoute[i],' ');
66   WriteLn;
67   Close(Output);
68 end;
69
70 begin
```

```
  Time:=TimeLimit*17;
  { Captam intreruperea 8 (timer-ul) }
  GetIntVec(8,@OldTimer);
  SetIntVec(8,@MyTimer);
  ReadData;
  Route[1]:=1;
  Seen:=[1];
  MinCost:=MaxInt;
  Bkt(2,0);
  WriteSolution;
  { Restauram timer-ul }
  SetIntVec(8,@OldTimer);
end.
```

Și această a doua metodă are neajunsurile ei, deoarece presupune scrierea a destul de multe linii de program în plus. În afară de aceasta, instrucțiunea de decrementare a variabilei `Time`, precum și apelul suplimentar de procedură din cadrul întreruperii de ceas mai reduc puțin timpul dedicat calculelor efective. A treia variantă elimină și aceste deficiențe. Ea se bazează pe accesarea directă a locației de memorie $0000:$046C, unde se află, reprezentat pe 4 octeți, numărul de apeluri ale timer-ului (numărul de **tacți**) începând de la miezul nopții. Dacă declarăm o variabilă `Time` de tip `Longint` (deoarece acest tip de date ocupă 4 octeți) exact la această adresă, folosind clauza Pascal `absolute`, variabila se va incrementa la fiecare 55ms, scutindu-ne pe noi de această grijă. Dacă înmulțim variabila cu 55/1.000, aflăm exact numărul de secunde scurse de la miezul nopții. Dacă împărțim acest rezultat la 3.600, putem afla ora exactă ș.a.m.d. Lucrul care ne interesează pe noi este să setăm o „alarmă” care să oprească programul peste 30 de secunde. 30 de secunde înseamnă $30 \times 18,2$ incrementări ale variabilei `Time`. Folosind în loc de 18,2 valoarea 17 (pentru a păstra o rezervă), rezultă că trebuie să ne oprim atunci când `Time` are o valoare cu $30 \times 17$ mai mare decât la intrarea în program. Primul lucru pe care îl va face programul va fi să dea unei variabile `Alarm` valoarea `Time` + $30 \times 17$. Periodic (cel mai comod la intrarea în procedura backtracking) se va testa valoarea variabilei `Time` și atunci când ea este egală cu `Alarm`, se va ieși din program.

```
program Hamilton;
{$B-,I-,R-,S-}
const NMax=30;
      TimeLimit=30; { secunde }

type Vector=array[1..NMax] of Integer;
     Matrix=array[1..NMax,1..NMax] of Integer;
```

```
8  var A:Matrix;
9      Route,BestRoute:Vector;
10     Seen:set of 1..NMax;
11     N,Cost,MinCost:Integer;
12     Time:LongInt absolute $0000:$046C;
13     Alarm:LongInt;
14
15 procedure SetAlarm;
16 begin
17   Alarm:=Time+TimeLimit*17;
18   { Cifra corecta era nu 17, ci 18.2;
19     am pastrat insa o rezerva de siguranta }
20 end;
21
22 procedure ReadData;
23 var i,j:Integer;
24 begin
25   Assign(Input,'input.txt');Reset(Input);
26   ReadLn(N);
27   for i:=1 to N do
28     begin
29       for j:=1 to N do Read(A[i,j]);
30       ReadLn;
31     end;
32   Close(Input);
33 end;
34
35 procedure Bkt(Level,Cost:Integer);
36 var i:Integer;
37 begin
38   if Level=N+1
39     then begin
40            Inc(Cost,A[Route[N],1]);
41            if Cost<MinCost
42              then begin
43                     BestRoute:=Route;
44                     MinCost:=Cost;
45                   end;
46          end
47     else if (Time<Alarm) and (Cost<MinCost)
48            then for i:=1 to N do
49              if not (i in Seen)
50                then begin
51                       Seen:=Seen+[i];
52                       Route[Level]:=i;
```

```
53                      Bkt(Level+1,Cost+A[Route[Level-1],i]);
54                      Seen:=Seen-[i];
55                    end;
56  end;
57
58  procedure WriteSolution;
59  var i:Integer;
60  begin
61    Assign(Output,'output.txt');Rewrite(Output);
62    WriteLn(MinCost);
63    for i:=1 to N do Write(BestRoute[i],' ');
64    WriteLn;
65    Close(Output);
66  end;
67
68  begin
69    SetAlarm;
70    ReadData;
71    Route[1]:=1;
72    Seen:=[1];
73    MinCost:=MaxInt;
74    Bkt(2,0);
75    WriteSolution;
76  end.
```

Singura problemă pe care o poate ridica această ultimă versiune este următoarea: dacă programul este lansat în execuție la un moment foarte apropiat de miezul nopții, atunci variabila `Alarm` va avea o valoare mai mare decât numărul de tacți dintr-o zi. Variabila `Time` nu va ajunge niciodată la această valoare, deoarece la miezul nopții ea va lua din nou valoarea 0. Este totuși puțin probabil să vă fie corectat programul la miezul nopții...

## 5.2 Greedy euristic

După cum am spus, la unii algoritmi greedy, criteriul de departajare garantează că soluția optimă nu este niciodată scăpată din vedere. De și mai multe ori, totuși, criteriile de departajare nu pot promite acest lucru; în general elevii, la ieșirea din sălile de concurs, în cazul unei probleme mai controversate, își expun părerile și ideile față de colegii lor, apoi fiecare îi demonstrează celuilalt că algoritmul propus de el nu merge, prezentându-i un contraexemplu. Momentele cele mai picante se produc atunci când algoritmul pare să

nu fie corect, dar nici nu se poate găsi un contraexemplu.

În aceste situații, criteriile de departajare a soluțiilor la algoritmii greedy se numesc **funcții euristice**, iar algoritmul în sine se numește **greedy euristic** (în greaca veche, *heuriskein* însemna *a afla*). Dacă nu poate promite optimalitatea, funcția euristică trebuie în orice caz aleasă cât mai bine, respectiv trebuie să aibă șanse cât mai mari să rețină soluția optimă, sau măcar să rețină la fiecare pas soluții cât mai apropiate de cea optimă.

Un singur algoritm greedy euristic are șanse mici să găsească soluția optimă. Dar algoritmii greedy euristici au unele proprietăți interesante:

- Se încadrează cu ușurință în timpul de rulare.
- Sunt ușor de implementat.
- O modificare cât de mică a funcției euristice poate modifica radical algoritmul și soluția furnizată de el. Deoarece nu avem de unde ști care dintre funcțiile euristice este mai bună (acest lucru depinde de datele pe care este testată problema), ideal este să reținem ambele soluții furnizate și să o alegem pe cea mai bună.
- De multe ori, datele de intrare sunt vectori sau matrice; în unele situații, sensul în care sunt ele parcurse pentru determinarea soluției nu este important. Schimbând sensul de parcurgere, obținem de asemenea două soluții distincte pe care le putem compara.
- Aproape întotdeauna, funcțiile euristice conțin teste de genul:

```
if A<B then Actiune1 else Actiune2;
```

Din punct de vedere logic, dacă `A=B` se poate executa oricare din cele două acțiuni. Condiția `A<B` este echivalentă cu condiția `A<=B`. Totuși, din punct de vedere al calculatorului, cele două condiții sunt absolut diferite și pot produce soluții cu totul diferite. Iată de exemplu, două rutine care caută poziția $k$ pe care se află elementul minim într-un vector $V$ cu $N$ elemente:

```
k:=1;
for i:=2 to N do
  if V[i]<V[k] then k:=i;
```

respectiv

```
k:=1;
for i:=2 to N do
  if V[i]<=V[k] then k:=i;
```

Cele două versiuni vor găsi într-adevăr un indice $k$ astfel încât $V[k]$ să fie minim. Totuși, dacă există mai multe elemente de valoare minimă, atunci prima versiune va întoarce indicele cel mai mic, pe când ultima îl va întoarce pe cel mai mare.

Iată un exemplu de funcție euristică pentru problema comis-voiajorului: pornim din nodul 1 și, la fiecare pas, ne deplasăm în cel mai apropiat nod care nu a fost vizitat încă. După ce toate nodurile au fost vizitate, trebuie numai să ne deplasăm din ultimul nod vizitat în nodul 1. Luată în sine, această euristică nu este strălucită. Ea poate însă să fie „clonată” într-o multitudine de variante. În primul rând că nu este obligatoriu să pornim din nodul 1. Putem aplica același algoritm pornind pe rând din fiecare nod; la sfârșit tipărim soluția de cost minim. Prima variantă a programului Pascal este:

```
program Hamilton;
{$B-,I-,R-,S-}
const NMax=30;
type Vector=array[1..NMax] of Integer;
     Matrix=array[1..NMax,1..NMax] of Integer;
var A:Matrix;
    Route,BestRoute:Vector;
    N,Cost,MinCost,i:Integer;

procedure ReadData;
var i,j:Integer;
begin
  Assign(Input,'input.txt');Reset(Input);
  ReadLn(N);
  for i:=1 to N do
    begin
      for j:=1 to N do Read(A[i,j]);
      ReadLn;
    end;
  Close(Input);
end;

procedure Greedy1(Start:Integer;var R:Vector;
                  var Cost:Integer);
var i,j,Closest:Integer;
```

```
26      Seen:set of 1..NMax;
27  begin
28    R[1]:=Start;
29    Cost:=0;
30    Seen:=[Start];
31    for i:=2 to N do
32      begin
33        { Cauta nodul cel mai apropiat }
34        Closest:=MaxInt;
35        for j:=1 to N do
36          if (not (j in Seen)) and (A[R[i-1],j]<Closest)
37            then begin
38                   Closest:=A[R[i-1],j];
39                   R[i]:=j;
40                 end;
41        Inc(Cost,Closest);
42        Seen:=Seen+[R[i]];
43      end;
44    { Inchide ciclul }
45    Inc(Cost,A[R[N],Start]);
46  end;
47
48  procedure Update;
49  begin
50    if Cost<MinCost
51      then begin
52             MinCost:=Cost;
53             BestRoute:=Route;
54           end;
55  end;
56
57  procedure WriteSolution;
58  var i:Integer;
59  begin
60    Assign(Output,'output.txt');Rewrite(Output);
61    WriteLn(MinCost);
62    for i:=1 to N do Write(BestRoute[i],' ');
63    WriteLn;
64    Close(Output);
65  end;
66
67  begin
68    ReadData;
69    MinCost:=MaxInt;
70    for i:=1 to N do
```

```
71      begin
72        Greedy1(i,Route,Cost);
73        Update;
74      end;
75    WriteSolution;
76  end.
```

De sine stătătoare, funcția euristică nu este strălucită. Totuși, ea poate fi lesne modificată. Se observă că, în procedura `Greedy1`, instrucțiunea

```
for j:=1 to N do...
```

poate fi înlocuită cu

```
for j:=N downto 1 do...
```

, iar condiția

```
(A[R[i-1],j] < Closest)
```

cu

```
(A[R[i-1],j]<=Closest)
```

Făcând toate combinațiile posibile, rezultă alte trei proceduri, `Greedy2`, `Greedy3` și `Greedy4`, iar noua formă a programului principal este:

```
begin
  ReadData;
  MinCost:=MaxInt;

  for i:=1 to N do
    begin
      Greedy1(i,Route,Cost);
      Update;
```

```
9         Greedy2(i,Route,Cost);
10        Update;
11        Greedy3(i,Route,Cost);
12        Update;
13        Greedy4(i,Route,Cost);
14        Update;
15      end;
16
17    WriteSolution;
18  end.
```

După cum se vede, adaosul de proceduri face ca sursa să atingă dimensiuni impunătoare, dar efortul necesar pentru a o scrie este aproape aceeași ca și când ar fi existat o singură funcție euristică. Practic, trebuie scrisă una singură din cele patru funcții, restul rezumându-se la copierea unor blocuri cu ajutorul editorului Borland Pascal.

## 5.3 Decizia între greedy euristic și backtracking

Pentru a ne asigura și mai multe puncte din cele puse în joc, putem încerca următoarea combinație: pentru grafuri mici, care pot fi examinate exhaustiv în timp de câteva secunde, vom apela la algoritmul backtracking pentru rezolvarea problemei. Numai pentru valori mari, pentru care știm sigur că backtracking-ul depășește timpul admis, vom apela la funcțiile euristice. Ce înseamnă valori „mici" si „mari" se poate aproxima sau se poate determina după câteva teste. Aceste valori depind de problemă și de mașina folosită.

Deoarece primele teste pentru fiecare problemă (uneori o treime sau chiar jumătate din ele) sunt de dimensiuni mici, e bine dacă vi le puteți asigura printr-un backtracking care de regulă se implementează în 15-20 minute.

## 5.4 Combinația greedy euristic + backtracking

Urmărind prima rezolvare de la punctul (1) - varianta backtracking fără nici un fel de modificări - se observă că variabila `MinCost` se inițializează cu valoarea `MaxInt`. În felul acesta, prima soluție găsită este implicit cea mai bună și durează o vreme până când rezultatele încep să se apropie de optim. Pe de altă parte, evaluarea unui anumit lanț din graf și prelungirea lui cu noi noduri până la închiderea ciclului hamiltonian nu se fac decât dacă costul lanțului nu a depășit deja valoarea `MinCost`. De aici provine întrebarea

firească: ce-ar fi dacă, în loc să inițializăm variabila `MinCost` cu valoarea `MaxInt`, am lansa mai întâi unul sau mai multe greedy-uri euristice (depinde câte apucăm să scriem) pentru a da o valoare mai apropiată de adevăr variabilei `MinCost` ? Sigur, cu o floare nu se face primăvară, dar în cazul nostru se pot câștiga secunde prețioase. Se poate de asemenea ca după aceste greedy-uri să apelăm nu un backtracking simplu, ci unul omorât prin orice metodă, caz în care șansele se îmbunătățesc considerabil. Programul principal ar putea fi atunci:

```
begin
  SetAlarm;
  ReadData;
  MinCost:=MaxInt;
  for i:=1 to N do
    begin
      Greedy1(i,Route,Cost);
      Update;
      Greedy2(i,Route,Cost);
      Update;
      Greedy3(i,Route,Cost);
      Update;
      Greedy4(i,Route,Cost);
      Update;
    end;
  Route[1]:=1;
  Seen:=[1];
  Bkt(2,0);
  WriteSolution;
end.
```

## 5.5 Testarea aleatoare a posibilităților

Oricât ar părea de ciudat, și aceasta este o cale de a ieși din încurcătură. Ce-i drept, nu cea mai eficientă, dar atunci când imaginația vă joacă feste iar backtracking-ul nu vă surâde, puteți încerca chiar și o rezolvare care se bazează puternic pe funcția `Random`. În acest caz, tot ce aveți de făcut este să generați aleator cicluri hamiltoniene și să-i calculați fiecăruia costul. La sfârșit îl tipăriți pe cel de cost minim găsit. Bineînțeles, oprirea programului se va face printr-o „omorâre” de orice tip. Timpul de implementare al unei asemenea rezolvări este de ordinul minutelor. Această versiune găsește uneori soluția optimă, dar alteori este foarte departe de ea. De asemenea, are marele dezavantaj că la

două rulări consecutive nu generează același rezultat, deoarece procedura `Randomize` își extrage variabila `RandSeed` (folosită pentru a genera numere aleatoare) pe baza timerului... Personal nu o recomand, dar este destul de des folosită pe la concursuri. Și este momentul să amintim o urare ce li se adresează concurenților care intră în sala de corectare, respectiv „Să fie într-un timer bun!”.

```
program Hamilton;
{$B-,I-,R-,S-}
const NMax=30;
      TimeLimit=30; { secunde }

type Vector=array[1..NMax] of Integer;
     Matrix=array[1..NMax,1..NMax] of Integer;
var A:Matrix;
    Route,BestRoute:Vector;
    Seen:set of 1..NMax;
    N,Cost,MinCost:Integer;
    Time:LongInt absolute $0000:$046C;
    Alarm:LongInt;

procedure SetAlarm;
begin
  Alarm:=Time+TimeLimit*17;
  { Cifra corecta era nu 17, ci 18.2;
    am pastrat insa o rezerva de siguranta }
end;

procedure ReadData;
var i,j:Integer;
begin
  Assign(Input,'input.txt');Reset(Input);
  ReadLn(N);
  for i:=1 to N do
    begin
      for j:=1 to N do Read(A[i,j]);
      ReadLn;
    end;
  Close(Input);
end;

procedure RandomCycle;
var i,j:Integer;
begin
```

```
38    Route[1]:=Random(N)+1;
39    Seen:=[Route[1]];
40    Cost:=0;
41    for i:=2 to N do
42      begin
43        repeat Route[i]:=Random(N)+1;
44        until not (Route[i] in Seen);
45        Seen:=Seen+[Route[i]];
46        Inc(Cost,A[Route[i-1],Route[i]]);
47      end;
48    Inc(Cost,A[Route[N],Route[1]]);
49  end;
50
51  procedure Update;
52  begin
53    if Cost<MinCost
54      then begin
55             MinCost:=Cost;
56             BestRoute:=Route;
57           end;
58  end;
59
60  procedure WriteSolution;
61  var i:Integer;
62  begin
63    Assign(Output,'output.txt');Rewrite(Output);
64    WriteLn(MinCost);
65    for i:=1 to N do Write(BestRoute[i],' ');
66    WriteLn;
67    Close(Output);
68  end;
69
70  begin
71    SetAlarm;
72    Randomize;
73    ReadData;
74    MinCost:=MaxInt;
75    while Time<Alarm do
76      begin
77        RandomCycle;
78        Update;
79      end;
80    WriteSolution;
81  end.
```

# Capitolul 6

# Probleme de concurs

## 6.1 Problema 1

**ENUNȚ**: Se consideră următorul joc: Pe o tablă liniară cu $2N + 1$ căsuțe sunt dispuse $N$ bile albe (în primele $N$ căsuțe) și $N$ bile negre (în ultimele $N$ căsuțe), căsuța din mijloc fiind liberă. Bilele albe se pot mișca numai spre dreapta, iar cele negre numai spre stânga. Mutările posibile sunt:

1. O bilă albă se poate deplasa o căsuță spre dreapta, numai dacă aceasta este liberă;
2. O bilă albă poate sări peste bila aflată imediat în dreapta ei (indiferent de culoarea acesteia), așezându-se în căsuța de dincolo de ea, numai dacă aceasta este liberă;
3. O bilă neagră se poate deplasa o căsuță spre stânga, numai dacă aceasta este liberă;
4. O bilă neagră poate sări peste bila aflată imediat în stânga ei (indiferent de culoarea acesteia), așezându-se în căsuța de dincolo de ea, numai dacă aceasta este liberă.

Trebuie schimbat locul bilelor albe cu cele negre. Se mai cere în plus ca prima mutare să fie făcută cu o bilă albă.

**Intrarea**: De la tastatură se citește numărul $N \leq 1.000$.

**Ieșirea**: În fișierul `OUTPUT.TXT` se vor tipări două linii terminate cu ¡tt¿¡Enter¿¡/tt¿. Pe prima se va tipări numărul de mutări efectuate, iar pe a doua o succesiune de cifre cuprinse între 1 și 4, nedespărțite prin spații, corespunzătoare mutărilor ce trebuie făcute.

**Exemple**:

- $N = 1 \implies$ Ieșirea 141
- $N = 2 \implies$ Ieșirea 14322341

**Complexitate cerută**: $O(N^2)$.

**Timp de implementare**: 1h.

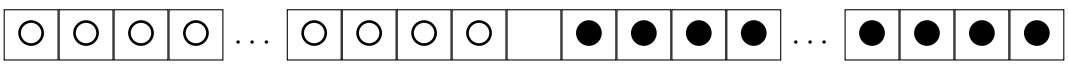
**Timp de rulare**: 10 secunde pentru un test.

**REZOLVARE**: La prima vedere, problema pare să se preteze la o rezolvare în timp exponențial, prin metoda „Branch and Bound". Un neajuns al enunțului pare să fie faptul că nu se specifică dacă numărul de mutări efectuate trebuie sau nu să fie minim. Pentru a ne lămuri, să privim în detaliu soluțiile pentru $N = 1$ și $N = 2$:

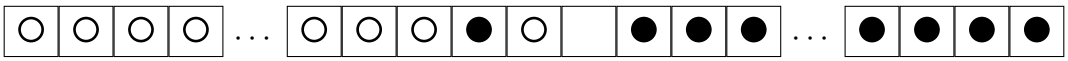
- Pentru $N = 1$, toate mutările sunt forțate ((a) - se mută bila albă, (b) - se sare cu cea neagră peste ea, (c) - se mută din nou bila albă); trebuie remarcat că după mutările (a) și (b) se obțin două configurații simetrice una în raport cu cealaltă (oglindite).
- Pentru $N = 2$, se poate începe sărind cu bila albă de la margine peste cealaltă, dar această mutare ar duce la blocarea jocului. Este deci obligatoriu să se înceapă prin a împinge bila albă centrală (a). Următoarea mutare este forțată ((b) - se sare cu bila neagră peste cea albă), apoi toate mutările sunt obligate (în sensul că dacă la orice pas se face altă mutare decât cea care conduce la soluție, jocul se blochează în câteva mutări): (c) - se împinge bila neagră, (d), (e) - se sare de două ori cu bilele albe, (f) - se împinge bila neagră, (g) - se sare cu bila neagră, (h) - se împinge bila albă. Trebuie din nou remarcat că după mutările (c) și (e) se obțin două configurații simetrice.

Așadar în ambele cazuri, soluția este unică. De fapt, există două soluții asemănătoare, una dacă se începe cu o mutare a bilei albe și una dacă se începe cu o mutare a bilei negre. Fiindcă enunțul impune ca prima mutare să se facă cu o bilă albă, soluția este unică. Se mai observă și că, atât pentru $N = 1$ cât și pentru $N = 2$ șirul de mutări este simetric. Pentru a indica efectiv modul de determinare a soluției (care va sugera și ideea de scriere a programului) și pentru a explica observațiile de mai sus, să generalizăm observațiile făcute pentru un $N$ oarecare.

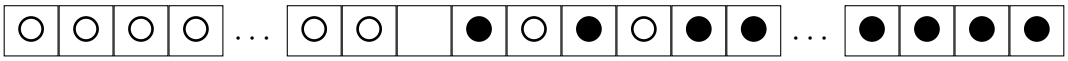
- Configurația inițială este:



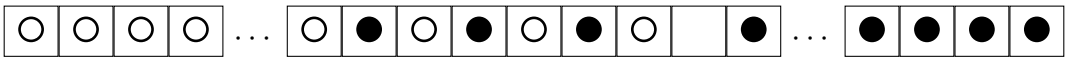
- Se împinge bila albă și se sare cu cea neagră peste ea (șirul de mutări 14):



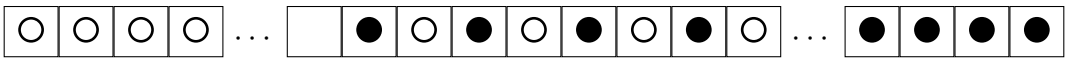
- Se împinge bila neagră și se sare de două ori cu cele albe peste ea (șirul de mutări 322):



- Se împinge bila albă și se sare de trei ori cu cele negre peste ea (șirul de mutări 1444):

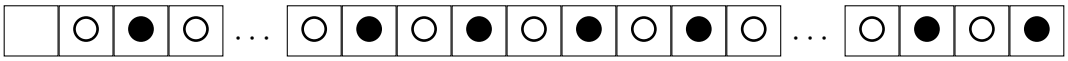


- Se împinge bila neagră și se sare de patru ori cu cele albe peste ea (șirul de mutări 32222):

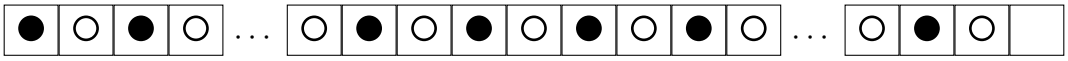


. . . . . .

- Se împinge bila albă (mutarea 1)

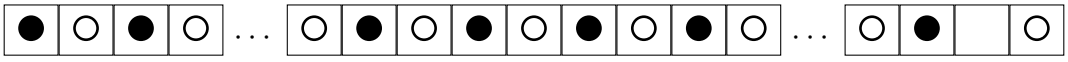


- Se sare de $N$ ori cu cele negre peste ea (șirul de mutări 44..44):



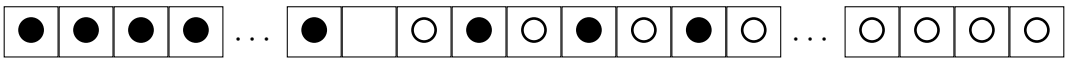
Ultimele două configurații sunt simetrice. În acest moment șirul de mutări se inversează:

- Se împinge bila albă (mutarea 1):

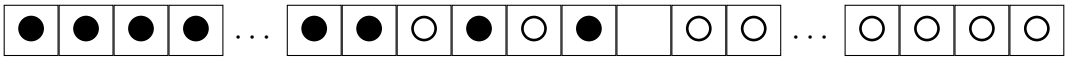


. . . . . .

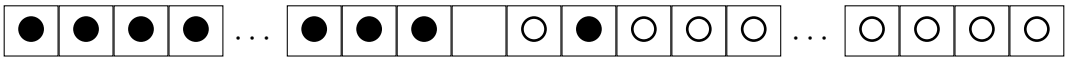
- Se sare de patru ori cu bilele albe și se împinge bila neagră (șirul de mutări 22223):



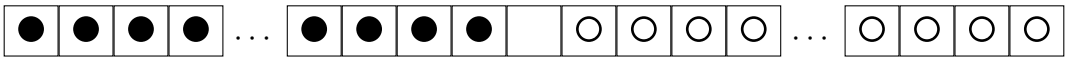
- Se sare de trei ori cu bilele negre și se împinge bila albă (șirul de mutări 4441):



- Se sare de două ori cu bilele albe și se împinge bila neagră (șirul de mutări 223):



- Se sare cu bila neagră și se împinge bila albă (șirul de mutări 41), obținându-se configurația finală:



În concluzie, șirul de mutări este: o împingere - un salt - o împingere - două salturi - o împingere - trei salturi - ... - o împingere - $N-1$ salturi - o împingere - $N$ salturi - o împingere - $N-1$ salturi - ... - o împingere - trei salturi - o împingere - două salturi - o împingere - un salt - o împingere, culorile alternând la fiecare pas.

Pentru a calcula numărul de mutări, putem să le numărăm pe măsură ce le efectuăm, dar deoarece se cere afișarea mai întâi a numărului de mutări și după aceea a mutărilor în sine, trebuie fie să stocăm toate mutările în memorie, fie să lucrăm cu fișiere temporare, ambele variante putând duce la complicații nedorite. Din fericire, numărul de mutări se poate calcula cu ușurință astfel: fiecare piesă albă trebuie mutată în medie cu $N$ pași către dreapta și fiecare piesă neagră trebuie mutată cu $N$ pași către stânga. Deci numărul total de pași este $2N(N+1)$. Din secvența generală de mutări expusă mai sus se observă că nu se fac decât $2N$ împingeri de piese (mutări de un singur pas), restul fiind salturi (mutări de câte doi pași). Deci numărul de mutări este:

$$2N + \frac{2N(N+1) - 2N}{2} = 2N + N^2 = N(N+2) \tag{6.1}$$

De aici deducem că nu există un algoritm mai bun decât $O(N^2)$, deoarece numărul de mutări este $O(N^2)$. Propunem ca temă cititorului să demonstreze că nu există decât

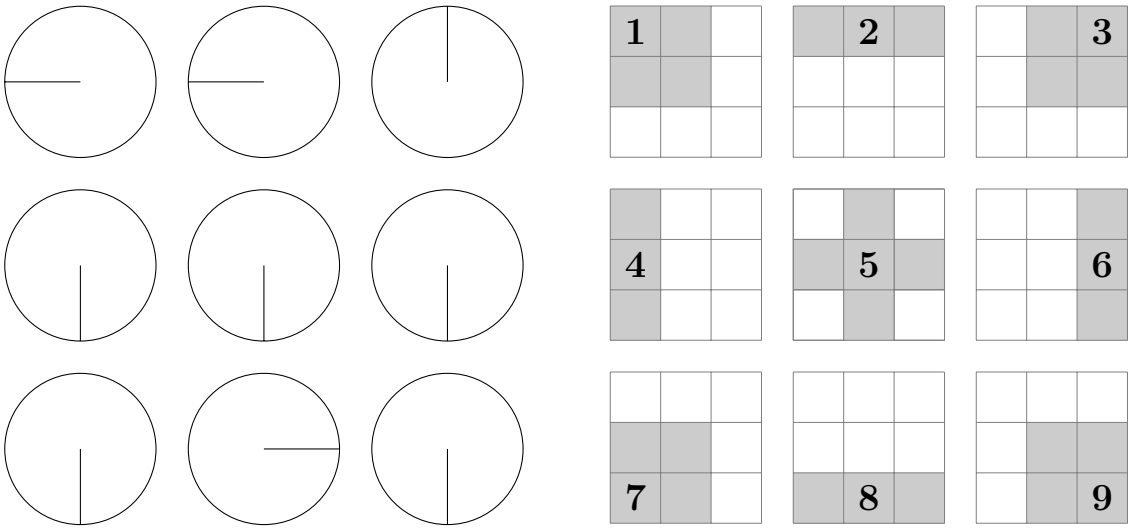
două succesiuni de mutări care rezolvă problema, din care una începe cu mutarea unei piese albe, iar cealaltă este oglindirea ei și începe cu mutarea unei piese negre, deci nu poate constitui o soluție corectă. Demonstrația începe prin a arăta că sunt necesare cel puțin $2N$ împingeri de piese. Această demonstrație explică de ce nu se cere un număr minim de mutări în enunț - cerința nu ar avea sens întrucât soluția este oricum unică. Acestea fiind zise, programul arată astfel:

```
#include <stdio.h>
int N;
FILE *OutF;

void Jump(int Level, char A, char B)
{ int i;
  putc(A,OutF);
  for (i=1;i<=Level;i++) putc(B,OutF);
  if (Level<N)
    {
      Jump(Level+1,'1'+'2'-A,'4'+'3'-B);
      for (i=1;i<=Level;i++) putc(B,OutF);
    }
  putc(A,OutF);
}

void main(void)
{
  printf("N=");scanf("%d",&N);
  OutF=fopen("output.txt","wt");
  fprintf(OutF,"%ld\n",(long)N*(N+2));
  Jump(1,'1','4');
  fprintf(OutF,"\n");
  fclose(OutF);
}
```

## 6.2 Problema 2

Problema următoare a fost propusă la a VI-a Olimpiadă Internațională de Informatică, Stockholm 1994. Este și ea un bun exemplu de situație în care putem cădea în plasa unei rezolvări „Branch and Bound” atunci când nu este cazul.

**ENUNȚ**: Se dă o configurație de $3 \times 3$ ceasuri, fiecare având un singur indicator care poate arăta numai punctele cardinale (adică orele 3, 6, 9 și 12). Asupra acestor ceasuri se poate acționa în nouă moduri distincte, fiecare acțiune însemnând mișcarea limbilor unui anumit grup de ceasuri în sens orar cu 90°. În figura de mai jos se dă un exemplu de configurație inițială a ceasurilor și se arată care sunt cele nouă tipuri de mutări (pentru fiecare tip de mutare se mișcă numai ceasurile reprezentate hașurat).




Se cere ca, într-un număr minim de mutări, să aducem toate indicatoarele la ora 12.

Intrarea se face din fișierul `INPUT.TXT`, care conține configurația inițială sub forma unei matrice $3 \times 3$. Pentru fiecare ceas se specifică câte o cifră: 0 = ora 12, 1 = ora 3, 2 = ora 6, 3 = ora 9.

Ieșirea se va face în fișierul `OUTPUT.TXT` sub forma unui șir de numere între 1 și 9, pe un singur rând, separate prin spațiu, reprezentând șirul de mutări care aduc ceasurile în starea finală. Se cere o singură soluție.

**Exemplu**: Pentru figura de mai sus, fișierul `INPUT.TXT` este

```
3 3 0
2 2 2
2 1 2
```

iar fișierul `OUTPUT.TXT` ar putea fi:

```
5 8 4 9
```

**Timp de implementare**: 1h - 1h 15min.

**Timp de rulare**: o secundă.

**Complexitate cerută**: $O(1)$ (timp constant).

**REZOLVARE**: Din câte am văzut pe la concursuri, peste jumătate din elevi s-ar apuca direct să implementeze o rezolvare Branch and Bound la această problemă, fără să-și mai bată capul prea mult. Există argumente în favoarea acestei inițiative:

- Mulți preferă să nu mai piardă timpul căutând o altă soluție, mai ales că problema seamănă mult cu „Lampa lui Dario Uri" (care de fapt este exact problema ceasurilor, dar în care ceasurile au doar două stări în loc de patru). În plus, se știe că pe cazul general al unei table $N \times N$, cele două probleme nu admit rezolvări polinomiale și atunci cea mai sigură soluție este prin tehnica Branch and Bound.
- De asemenea, se observă că numărul total de configurații posibile pentru o tablă cu 9 ceasuri este de $4^9$, adică aproximativ un sfert de milion. Un algoritm Branch and Bound ar furniza așadar o soluție în timp rezonabil. Raționamentul multor elevi este „decât să pierd timpul căutând o soluție mai bună, fără să am certitudinea că o voi găsi, mai bine folosesc timpul implementând un Branch care măcar știu sigur că merge".
- Problema cere o soluție într-un număr minim de pași, lucru care îi cam descurajează pe cei care încă ar vrea să caute alte rezolvări. „Alte rezolvări" înseamnă de obicei un Greedy comod de implementat, iar asupra rezolvărilor Greedy se poartă întotdeauna discuții interminabile pe culoarele sălilor de concurs referitor la „cât de bune sunt" (adică în cât la sută din cazuri furnizează soluția optimă).

Se pierd însă din vedere unele lucruri esențiale. În primul rând, tabla nu este de $N \times N$, ci are dimensiuni fixate, $3 \times 3$. În al doilea rând, implementarea unui Branch and Bound în timp de concurs este o aventură nu tocmai ușor de dus la bun sfârșit (personal mi-a fost frică să o încerc vreodată). În sfârșit, după cum se va vedea mai jos, problema șirului minim de transformări este o pseudo-problemă, deoarece soluția simplă este oricum unică.

Ce se înțelege prin „soluție simplă"? Să remarcăm două lucruri:

1. Aplicarea de patru ori a aceleiași mutări nu schimbă nimic în configurația ceasurilor. Într-adevăr, mutarea va afecta de fiecare dată același grup de ceasuri, iar aplicarea

de patru ori va roti fiecare indicator cu 360°, adică îl va aduce în poziția inițială. Din acest motiv, toate afirmațiile făcute în cele ce urmează vor fi valabile în algebra modulo 4.

2. Ordinea în care se aplică transformările nu contează.

În consecință, prin „soluție simplă" se înțelege un șir de mutări ordonat crescător în care nici o mutare nu apare de mai mult de trei ori. Să demonstrăm acum că soluția simplă este unică.

Fie $A \in \mathbb{M}_3(\mathbb{Z}_4)$ matricea citită de la intrare, unde $a_{i,j}$ arată de câte ori a fost rotit ceasul $C_{i,j}$ peste ora 12. Fie matricea $B \in \mathbb{M}_3(\mathbb{Z}_4), b_{i,j} = 4 - a_{i,j}$. Matricea $B$ arată de câte ori mai trebuie rotit fiecare ceas până la ora 12. O soluție înseamnă a efectua fiecare din cele 9 mutări de un număr de ori, $p_1, p_2, \ldots, p_9$. Cum afectează aceste mutări ceasurile? Se poate deduce ușor:

| Ceasul | Tipurile de mutări care îl afectează |
|---|---|
| $C_{1,1}$ | 1, 2, 4 |
| $C_{1,2}$ | 1, 2, 3, 5 |
| $C_{1,3}$ | 2, 3, 6 |
| $C_{2,1}$ | 1, 4, 5, 7 |
| $C_{2,2}$ | 1, 3, 5, 7, 9 |
| $C_{2,3}$ | 3, 5, 6, 9 |
| $C_{3,1}$ | 4, 7, 8 |
| $C_{3,2}$ | 5, 7, 8, 9 |
| $C_{3,3}$ | 6, 8, 9 |

Se obține deci un sistem de 9 ecuații cu 9 necunoscute:

$$P = \begin{pmatrix} p_1 + p_2 + p_4 & p_1 + p_2 + p_3 + p_5 & p_2 + p_3 + p_6 \\ p_1 + p_4 + p_5 + p_7 & p_1 + p_3 + p_5 + p_7 + p_9 & p_3 + p_5 + p_6 + p_9 \\ p_4 + p_7 + p_8 & p_5 + p_7 + p_8 + p_9 & p_6 + p_8 + p_9 \end{pmatrix} \equiv B \qquad (6.2)$$

Să presupunem că acest sistem admite două soluții $p_1, \ldots, p_9$ și $q_1, \ldots, q_9$. Atunci $P \equiv B \pmod 4$ și $Q \equiv B \pmod 4$, deci $P \equiv Q \pmod 4$ și, prin diferite combinații liniare ale celor 9 ecuații, se deduce $p_1 \equiv q_1$, $p_2 \equiv q_2$, ..., $p_9 \equiv q_9 \pmod 4$, adică cele două soluții sunt echivalente.

Odată ce am demonstrat că soluția este unică, algoritmul de găsire a ei este foarte simplu: găsim o soluție oarecare, o ordonăm crescător și eliminăm orice grup de 4 mutări

identice. Pentru a găsi o soluție oarecare, avem nevoie de niște mutări predefinite care să miște un singur ceas cu o singură poziție înainte, fără a afecta celelalte ceasuri. Aceste mutări vor fi reținute sub forma unui vector cu 9 componente, fiecare componentă indicând de câte ori se efectuează fiecare din cele 9 tipuri de mutări. Deoarece avem nevoie de 9 asemenea mutări predefinite, câte una pentru fiecare ceas, rezultatul va fi o matrice predefinită. De exemplu, pentru a determina secvența de mutări care rotește ceasul $C_{1,1}$ cu o poziție, trebuie rezolvat sistemul

$$\begin{pmatrix} p_1+p_2+p_4 & p_1+p_2+p_3+p_5 & p_2+p_3+p_6 \\ p_1+p_4+p_5+p_7 & p_1+p_3+p_5+p_7+p_9 & p_3+p_5+p_6+p_9 \\ p_4+p_7+p_8 & p_5+p_7+p_8+p_9 & p_6+p_8+p_9 \end{pmatrix} \equiv \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad (6.3)$$

lucru care nu este foarte ușor, dar se poate duce la bun sfârșit în timp de concurs. Soluția este $p_1 = 3, p_2 = 3, p_3 = 3, p_4 = 3, p_5 = 3, p_6 = 2, p_7 = 3, p_8 = 2, p_9 = 0$, adică mutarea 1 trebuie efectuată de trei ori, mutarea 2 de trei ori ș.a.m.d. Se obține prima linie din matricea predefinită, $(3, 3, 3, 3, 3, 2, 3, 2, 0)$. Mai trebuie rezolvate propriu-zis sistemele de ecuații pentru ceasurile $C_{1,2}$ și $C_{2,2}$, soluțiile celorlalte sisteme decurgând ușor prin simetrie. Soluțiile apar în textul sursă.

Odată determinate aceste șiruri elementare de mutări, vom lua pe rând fiecare ceas, vom aplica șirul elementar corespunzător de atâtea ori cât e nevoie pentru a-l aduce la ora 12 și vom aduna modulo 4 toate mutările făcute. Vectorul sumă care rezultă este tocmai soluția noastră.

Pentru exemplul din enunț, folosind constantele din programul sursă, obținem:

$$B = \begin{pmatrix} 1 & 1 & 0 \\ 2 & 2 & 2 \\ 2 & 3 & 2 \end{pmatrix} \quad (6.4)$$

și

$$
\begin{aligned}
1 \times (3,3,3,3,3,2,3,2,0) = \quad & (3,3,3,3,3,2,3,2,0)+ \\
1 \times (2,3,2,3,2,3,1,0,1) = \quad & (2,3,2,3,2,3,1,0,1) \\
0 \times (3,3,3,2,3,3,0,2,3) = \quad & (0,0,0,0,0,0,0,0,0) \\
2 \times (2,3,1,3,2,0,2,3,1) = \quad & (0,2,2,2,0,0,0,2,2) \\
2 \times (2,3,2,3,1,3,2,3,2) = \quad & (0,2,0,2,2,2,0,2,0) \\
2 \times (1,3,2,0,2,3,1,3,2) = \quad & (2,2,0,0,0,2,2,2,0) \\
2 \times (3,2,0,3,3,2,3,3,3) = \quad & (2,0,0,2,2,0,2,2,2) \\
3 \times (1,0,1,3,2,3,2,3,2) = \quad & (3,0,3,1,2,1,2,1,2) \\
2 \times (0,2,3,2,3,3,3,3,3) = \quad & (0,0,2,0,2,2,2,2,2) \\
\quad & \underline{\qquad\qquad\qquad\qquad} \\
\quad & (0,0,0,1,1,0,0,1,1)
\end{aligned}
$$

Prin urmare soluția simplă a exemplului este: 4 5 8 9.

```c
#include <stdio.h>
typedef int Matrix[9][9];
typedef int Vector[9];
const Matrix A=
  {{3,3,3,3,3,2,3,2,0},  // Mutarile care misca ceasul C11
   {2,3,2,3,2,3,1,0,1},  // .
   {3,3,3,2,3,3,0,2,3},  // .
   {2,3,1,3,2,0,2,3,1},  // .
   {2,3,2,3,1,3,2,3,2},  // .
   {1,3,2,0,2,3,1,3,2},  // .
   {3,2,0,3,3,2,3,3,3},  // .
   {1,0,1,3,2,3,2,3,2},  // .
   {0,2,3,2,3,3,3,3,3}}; // Mutarile care misca ceasul C33

void main(void)
{ FILE *F=fopen("input.txt","rt");
  Vector V={0,0,0,0,0,0,0,0,0}; // Vectorul suma
  int i,j,k;

  for (i=0;i<=8;i++)
    { fscanf(F,"%d",&k);
      for (j=0;j<=8;j++)
        V[j]=(V[j]+(4-k)*A[i][j])%4;
    }
```

```
25    fclose(F);
26
27    F=fopen("output.txt","wt");
28    for(i=0;i<=8;i++)
29      for(j=1;j<=V[i];j++)
30        fprintf(F,"%d ",i+1);
31    fclose(F);
32  }
```

## 6.3 Problema 3

Problema de mai jos este un exemplu de situație în care căutarea exhaustivă a soluției este cea mai bună alegere. Ea a fost propusă spre rezolvare la a VIII-a Olimpiadă Internațională de Informatică, Veszprem, Ungaria 1996.

**ENUNȚ**: Văzând succesul cubului său magic, Rubik a inventat versiunea plană a jocului, numit „pătrate magice". Se folosește o tablă compusă din 8 pătrate de dimensiuni egale. Cele opt pătrate au culori distincte, codificate prin numere de la 1 la 8, ca în figura următoare:

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 8 | 7 | 6 | 5 |

Configurația tablei se poate reprezenta într-un vector cu 8 elemente citind cele opt pătrate, începând din colțul din stânga sus și mergând în sens orar. De exemplu, configurația din figură se reprezintă prin vectorul (1, 2, 3, 4, 5, 6, 7, 8). Aceasta este configurația inițială a tablei.

Unei configurații i se pot aplica trei transformări elementare, identificate prin literele „A", „B" și „C":

- „A" schimbă între ele cele două linii ale tablei;
- „B" rotește circular spre dreapta întregul dreptunghi (cu o poziție);
- „C" rotește în sens orar cele patru pătrate centrale (cu o poziție);

Efectele transformărilor elementare asupra configurației inițiale sunt reprezentate în figura de mai jos:

| | | | |
|---|---|---|---|
| 8 | 7 | 6 | 5 |
| 1 | 2 | 3 | 4 |

**A**

| | | | |
|---|---|---|---|
| 4 | 1 | 2 | 3 |
| 5 | 8 | 7 | 6 |

**B**

| | | | |
|---|---|---|---|
| 1 | 7 | 2 | 4 |
| 8 | 6 | 3 | 5 |

**C**

Din configurația inițială se poate ajunge în orice configurație folosind doar combinații de tranformări elementare. Trebuie să scrieți un program care calculează o secvență de transformări elementare care să aducă tabla de la configurația inițială la o anumită configurație finală cerută.

**Intrarea**: Fișierul `INPUT.TXT` conține 8 întregi pe aceeași linie, separați prin spații, descriind configurația finală.

Ieșirea se va face în fișierul `OUTPUT.TXT`. Pe prima linie a acestuia se va tipări lungimea $L$ a secvenței de transformări, iar pe fiecare din următoarele $L$ linii se va tipări câte un caracter „A”, „B” sau „C”, corespunzător mutărilor care trebuie efectuate.

**Exemplu**:

| **INPUT.TXT** | **OUTPUT.TXT** |
|---|---|
| 2 6 8 4 5 7 3 1 | 7<br>B<br>C<br>A<br>B<br>C<br>C<br>B |

**Timp limită pentru un test**: 20 secunde.

**Timp de implementare**: 1h 30min - 1h 45min

**Note**:

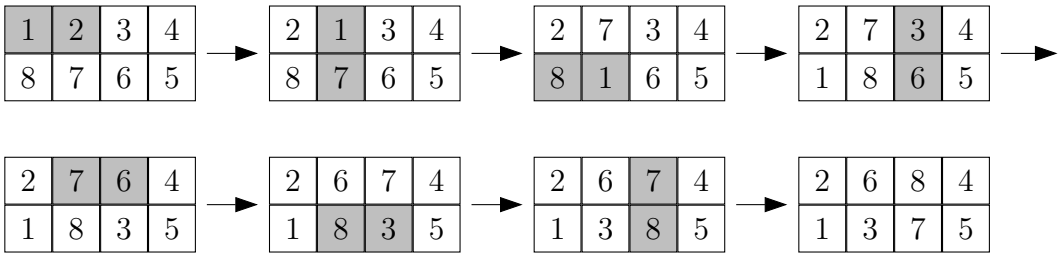
1. La concurs s-au acordat, pentru fiecare test, două puncte dacă se furniza o soluție și încă două dacă lungimea ei nu depășea 300 de mutări.
2. Concurenților li s-a furnizat un program auxiliar, `MTOOL.EXE`, cu care se puteau verifica soluțiile furnizate.

**REZOLVARE**: Și la această problemă se întrevăd două abordări, ca și în problema ceasurilor: una bazată pe mutări predefinite, iar cealaltă pe o căutare exhaustivă a

soluției. De data aceasta însă, prima este neinspirată. Să le analizăm pe rând pe fiecare, plecând de la următoarele considerente:

- Dacă se aplică de două ori la rând mutarea A, tabla rămâne nemodificată;
- Dacă se aplică de patru ori consecutiv una din mutările B sau C, tabla rămâne nemodificată;
- Ordinea în care se efectuează mutările contează.

Soluția pe care autorul a prezentat-o la concurs avea predefinite mai multe mutări care schimbau între ele oricare două pătrate vecine de pe tablă. Mergând din aproape în aproape, fiecare pătrat era adus în poziția corespunzătoare. Spre exemplu, succesiunea de mutări predefinite care duceau la configurația din exemplu este:




Această soluție funcționează instantaneu și este relativ ușor de implementat. Ea are însă defectul că soluția furnizată este extrem de lungă, ajungând frecvent la 500 de mutări. Din cele zece teste date, numai trei s-au încadrat în limita de 300 de mutări. Iată mai jos și sursa Pascal prezentată la concurs, care a câștigat numai 26 din cele 40 de puncte acordate pentru problemă:

```
1  program Magic;
2  {$B-,I-,R-,S-}
3  { Tabla este 1 2 3 4
4               A B C D }
5  const r12='BCBBB'; { Roteste in sens orar coloanele 12 }
6        r23='C';     { Roteste in sens orar coloanele 23 }
7        r34='BBBCB'; { Roteste in sens orar coloanele 34 }
8        r14='BBCBB'; { Roteste in sens orar coloanele 41 }
9
10       plBCD=r12+r23+r34+r14; { Permuta patratele BCD }
11       PL234=plBCD+'A'+plBCD+'A'; { Permuta coloanele 234 }
12
```

```
13        { SXY schimba intre ele coloanele X si Y }
14        S14='B'+PL234;
15        S12='BBB'+S14+'B';
16        S23='BB'+S14+'BB';
17        S34='B'+S14+'BBB';
18        S13=r12+r12+r23+r23+r12+r12;
19        S24='B'+S13+'BBB';
20
21        { RevXY schimba intre ele patratele vecine X si Y }
22        RevC3=plBCD+r23+r12+r12+r34+r34+'BBB'+plBCD+'A';
23        RevD4='BBB'+RevC3+'B';
24        RevB2='B'+RevC3+'BBB';
25        RevA1='BB'+RevC3+'BB';
26
27        Rev23='C'+RevC3+'CCC';
28        Rev12='B'+Rev23+'BBB';
29        Rev34='BBB'+Rev23+'B';
30        Rev14='BB'+Rev23+'BB';
31
32        RevAB='A'+Rev12+'A';
33        RevBC='A'+Rev23+'A';
34        RevCD='A'+Rev34+'A';
35        RevAD='A'+Rev14+'A';
36
37  type Matrix=array[1..2,1..4] of Integer;
38       Vector=array[1..60000] of Char;
39  var A,B:Matrix;
40      V:Vector;
41      N:Integer;
42
43  procedure MakeAMatrix;
44  begin
45    A[1,1]:=1;
46    A[1,2]:=2;
47    A[1,3]:=3;
48    A[1,4]:=4;
49    A[2,1]:=8;
50    A[2,2]:=7;
51    A[2,3]:=6;
52    A[2,4]:=5;
53  end;
54
55  procedure ReadBMatrix;
56  begin
57    Assign(Input,'input.txt');
```

```
58    Reset(Input);
59    Read(B[1,1]);
60    Read(B[1,2]);
61    Read(B[1,3]);
62    Read(B[1,4]);
63    Read(B[2,4]);
64    Read(B[2,3]);
65    Read(B[2,2]);
66    Read(B[2,1]);
67    Close(Input);
68  end;
69
70  procedure AddString(S:String);
71  { Adauga o secventa la sirul-solutie }
72  var i:Integer;
73  begin
74    for i:=1 to Length(S) do
75      begin
76        Inc(N);
77        V[N]:=S[i];
78      end;
79  end;
80
81  procedure FindElement(K:Integer;var X,Y:Integer);
82  { Cauta un element intr-o permutare }
83  var i,j:Integer;
84  begin
85    for i:=1 to 2 do
86      for j:=1 to 4 do
87        if A[i,j]=K then begin
88                           X:=i;
89                           Y:=j;
90                           Exit;
91                         end;
92  end;
93
94  procedure Switch(var X,Y:Integer);
95  { Schimba intre ele doua numere }
96  var IAux:Integer;
97  begin
98    IAux:=X;X:=Y;Y:=IAux;
99  end;
100
101 procedure Process;
102 { Transforma pozitia in pozitia B prin schimbari
```

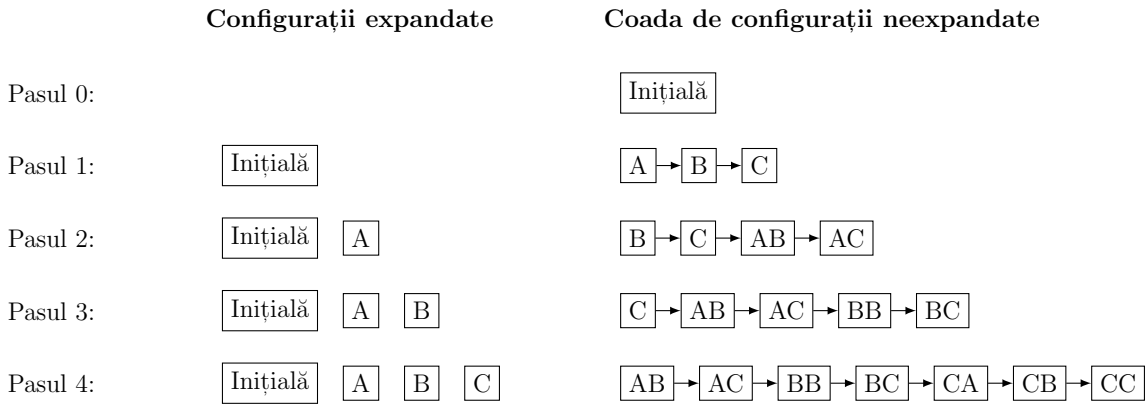
```
103   repetate ale elementelor vecine }
104 var i,j,k,l,m:Integer;
105 begin
106   for j:=1 to 4 do
107     for i:=1 to 2 do
108       begin
109         FindElement(B[i,j],k,l);
110         { Gaseste elementul care trebuie adus
111           pe pozitia (i,j) }
112         if k<>i then begin
113                        { Il aduce pe linia corecta }
114                        case l of
115                          1:AddString(RevA1);
116                          2:AddString(RevB2);
117                          3:AddString(RevC3);
118                          4:AddString(RevD4);
119                        end; {case}
120                        Switch(A[k,l],A[i,l]);
121                        k:=i;
122                      end;
123         for m:=l downto j+1 do
124           { Il aduce pe coloana corecta }
125           begin
126             if k=1
127               then case m of
128                      2:AddString(Rev12);
129                      3:AddString(Rev23);
130                      4:AddString(Rev34);
131                    end
132               else case m of
133                      2:AddString(RevAB);
134                      3:AddString(RevBC);
135                      4:AddString(RevCD);
136                    end;
137             Switch(A[k,m],A[k,m-1]);
138           end;
139       end;
140 end;
141
142 procedure Cut(K,D:Integer);
143 { Taie din vectorul V D pozitii incepand cu K }
144 var i:Integer;
145 begin
146   for i:=K to N-D do
147     V[i]:=V[i+D];
```

```
148   Dec(N,D);
149 end;
150
151 procedure Reduce;
152 { Reduce secventele de mutari identice }
153 var i:Integer;
154 begin
155   i:=1;
156   repeat
157     case V[i] of
158       'A':if (i<=N-1) and (V[i+1]='A')
159             then Cut(i,2)
160             else Inc(i);
161       'B':if (i<=N-3) and (V[i+1]='B')
162             and (V[i+2]='B') and (V[i+3]='B')
163             then Cut(i,4)
164             else Inc(i);
165       'C':if (i<=N-3) and (V[i+1]='C')
166             and (V[i+2]='C') and (V[i+3]='C')
167             then Cut(i,4)
168             else Inc(i);
169     end; {case}
170   until i=N;
171 end;
172
173 procedure WriteSolution;
174 var i:Integer;
175 begin
176   Assign(Output,'output.txt');
177   Rewrite(Output);
178   WriteLn(N);
179   for i:=1 to N do WriteLn(V[i]);
180   Close(Output);
181 end;
182
183 begin
184   N:=0;
185   MakeAMatrix;
186   ReadBMatrix;
187   Process;
188   Reduce;
189   WriteSolution;
190 end.
```

Singura soluție pare deci a fi una de tipul Branch and Bound, care nu este tocmai la îndemână. Cu toate acestea, numărul total de configurații posibile ale tablei este de numai 8! = 40.320. Într-adevăr, fiecare poziție de pe tablă se reprezintă printr-o permutare a mulțimii 1,2,3,4,5,6,7,8. Se poate face deci cu ușurință o căutare exhaustivă a soluției. Aceasta simplifică mult structurile de date folosite (implementarea Branch and Bound folosește structuri destul de încâlcite). În plus, practica arată că se poate ajunge în orice configurație în mai puțin de 25 de mutări.

Algoritmul de căutare este cunoscut sub numele de algoritmul lui Lee și are la bază următoarea idee: Se pornește cu configurația inițială, care este depusă într-o coadă. La fiecare pas se extrage prima configurație disponibilă din coadă, se efectuează pe rând fiecare din cele trei mutări și se obțin trei succesori. Aceștia sunt adăugați la sfârșitul cozii, dacă nu există deja în coadă. Acest pas se numește **expandare**. Expandarea continuă până când elementul selectat spre expandare este tocmai configurația finală.

Figura următoare indică modul de expandare a cozii, cu mențiunea că printr-o succesiune de litere ne-am referit la configurația care se obține efectuând mutările respective:




Se observă că, la pasul 2, în coadă au fost adăugate doar configurațiile „AB” și „AC”, iar configurația „AA” nu, deoarece prin efectuarea de două ori a mutării „A” se revine la configurația inițială, care a fost deja expandată. De asemenea, la pasul 3, după expandarea configurației „B” au fost adăugate în coadă numai configurațiile „BB” și „BC”, deoarece configurația „BA” este echivalentă cu configurația „AB”, aflată deja în listă.

Pseudocodul algoritmului este:

1: citește datele de intrare
2: inițializează coada cu configurația inițială
3: **cât timp** primul element al cozii nu este configurația finală **execută**

```
4:    expandează primul element al cozii
5:    pentru i = A, B, C execută
6:       dacă succesorul i nu a fost deja pus în coadă atunci
7:          adaugă succesorul i în coadă
8:       sfârșit dacă
9:    sfârșit pentru
10:   șterge primul element al cozii
11: sfârșit cât timp
12: reconstituie șirul de mutări
```

Algoritmul de mai sus garantează și găsirea soluției optime (în număr minim de mutări). Rămân de lămurit două lucruri: (1) Cum ne dăm seama dacă o configurație există deja în coadă și (2) cum se face reconstituirea soluției.

Pentru a afla dacă o configurație mai există în listă, cea mai simplă metodă ar fi o căutare secvențială a listei. Totuși, această versiune ar fi extrem de lentă, deoarece coada atinge rapid dimensiuni respectabile (de ordinul miilor de elemente). În plus, un element al listei ar reține configurația propriu-zisă (un vector cu opt elemente), ceea ce ar duce la un consum ridicat de memorie. Testul de egalitate a doi vectori ar fi și el costisitor din punct de vedere al timpului.

Există însă o altă metodă mai simplă. Am demonstrat că numărul de configurații posibile ale tablei este $8! = 40.320$. Dacă am putea găsi o funcție bijectivă $H : \mathbf{P_8} \to \{0, 1, \ldots, 40.319\}$, unde $\mathbf{P_8}$ este mulțimea permutărilor de 8 elemente, atunci ar fi suficient un vector caracteristic cu 40.320 elemente. De îndată ce introducem în coadă o nouă configurație $K$, nu avem decât să bifăm elementul corespunzător din vectorul caracteristic. Înainte de a adăuga o configurație în coadă, testăm dacă nu cumva elementul corespunzător ei a fost deja bifat, semn că nodul a mai fost vizitat.

Cum se construiește funcția $H$? Pentru orice permutare $p \in \mathbf{P_8}$, $H(p)$ este poziția lui $p$ în ordonarea lexicografică a lui $\mathbf{P_8}$ (începând de la 0):

$$H(1,2,3,4,5,6,7,8) = 0$$
$$H(1,2,3,4,5,6,8,7) = 1$$
$$H(1,2,3,4,5,7,6,8) = 2$$
$$\dots$$
$$H(8,7,6,5,4,3,1,2) = 40.318$$
$$H(8,7,6,5,4,3,2,1) = 40.319$$

Se observă că primele $7! = 5.040$ elemente din ordonare au pe prima poziție un 1, următoarele 5.040 au pe prima poziție un 2 etc. De asemenea, dintre elementele care au pe prima poziție un 1, primele $6! = 720$ au pe a doua poziție un 2, următoarele 720 au pe a doua poziție un 3 etc.

Să calculăm de exemplu $H(2,6,8,4,5,7,3,1)$. Prima cifră a permutării este 2, deci se adaugă $7! = 5.040$. Rămân cifrele 1, 3, 4, 5, 6, 7 și 8. A doua cifră a permutării este 6, a cincea ca valoare dintre cifrele rămase, deci se adaugă $4 \times 6! = 2.880$. Rămân cifrele 1, 3, 4, 5, 7 și 8 etc. Se aplică procedeul până la ultima cifră și rezultă:

| Cifre rămase | Permutarea | Valoarea adăugată |
|---|---|---|
| 1, 2, 3, 4, 5, 6, 7, 8 | 2 | $1 \times 7! = 5.040$ |
| 1, 3, 4, 5, 6, 7, 8 | 6 | $4 \times 6! = 2.880$ |
| 1, 3, 4, 5, 7, 8 | 8 | $5 \times 5! = 600$ |
| 1, 3, 4, 5, 7 | 4 | $2 \times 4! = 48$ |
| 1, 3, 5, 7 | 5 | $2 \times 3! = 12$ |
| 1, 3, 7 | 7 | $2 \times 2! = 4$ |
| 1, 3 | 3 | $1 \times 1! = 1$ |
| 1 | 1 | $0 \times 0! = 0$ |
| | | $H(p) = 8.585$ |

Reciproc se construiește permutarea când i se cunoaște valoarea atașată:

| Cifre nefolosite | $H(p)$ | | Cifra selectată | |
|---|---|---|---|---|
| 1, 2, 3, 4, 5, 6, 7, 8 | 8.585 | 8.585 div 7! = 1 | 2 | 8.585 mod 7! = 3.545 |
| 1, 3, 4, 5, 6, 7, 8 | 3.545 | 3.545 div 6! = 4 | 6 | 3.545 mod 6! = 665 |
| 1, 3, 4, 5, 7, 8 | 665 | 665 div 5! = 5 | 8 | 665 mod 5! = 65 |
| 1, 3, 4, 5, 7 | 65 | 65 div 4! = 2 | 4 | 65 mod 4! = 17 |
| 1, 3, 5, 7 | 17 | 17 div 3! = 2 | 5 | 17 mod 3! = 5 |
| 1, 3, 7 | 5 | 5 div 2! = 2 | 7 | 5 mod 2! = 1 |
| 1, 3 | 1 | 1 div 1! = 1 | 3 | 1 mod 1! = 0 |
| 1 | 0 | 0 div 0! = 0 | 1 | |

Rezultă $p = (2, 6, 8, 4, 5, 7, 3, 1)$.

Această metodă de căutare are și avantajul că în listă se va ține un singur număr pe doi octeți, făcându-se economie de memorie. Expandarea unui nod constă din trei pași:

1. Se extrage primul număr din listă și se reconstituie configurația atașată;
2. Se fac cele trei mutări, obținându-se trei succesori;
3. Pentru fiecare succesor se calculează funcția $H$ și dacă configurația nu este găsită în listă, este adăugată.

Pentru a face reconstituirea soluției avem nevoie de date suplimentare. Respectiv, vectorul caracteristic atașat permutărilor nu va mai reține doar dacă o poziție a fost „văzută" sau nu, ci și poziția din care ea provine (prin valoarea funcției $H$). Trebuie de asemenea reținut tipul mutării (A, B sau C) prin care s-a ajuns în acea configurație. Cei doi vectori se numesc `Father` și `MoveKind`. Inițial, toate elementele vectorului `Father` au eticheta „Unknown", semnificând că nodurile nu au fost încă vizitate, cu excepția elementului atașat configurației inițiale, care poartă eticheta specială „Root" (rădăcină).

Pseudocodul pentru expandarea unui nod arată cam așa:

```
1: K ← primul număr din coadă
2: P ← H^{-1}(K)
3: află cei trei succesori Q_A, Q_B, Q_C
4: pentru i = A, B, C execută
5:    dacă Father[H(Q_i)] = Unknown atunci
6:       Father[H(Q_i)] ← K
7:       MoveKind[H(Q_i)] ← i
8:       adaugă H(Q_i) în coadă
9:    sfârșit dacă
10: sfârșit pentru
11: șterge K din coadă
```

Reconstituirea soluției se face recursiv: se pornește de la configurația finală și se merge înapoi (folosind informația din vectorul `Father`) până la configurația inițială, măsurându-se astfel numărul de mutări. La revenire se tipăresc toate mutările efectuate (folosind informația din vectorul `MoveKind`).

```
#include <stdio.h>
#include <stdlib.h>
#define Unknown 0xFFFF
#define Root 0xFFFE

typedef huge unsigned Vector[40320];
typedef char CharVector[40320];
typedef int Perm[8];
typedef struct list { unsigned X; struct list * Next; } List;

Perm StartPerm={1,2,3,4,5,6,7,8}, EndPerm;
const Perm Moves[3]=
   {{7,6,5,4,3,2,1,0},
    {3,0,1,2,5,6,7,4},
    {0,6,1,3,4,2,5,7}};
/* Cele trei tipuri de mutari */
Vector Father; /* Legaturile de tip tata */
CharVector MoveKind;
unsigned StartValue,EndValue;
/* Valorile atasate configuratiilor initiala si finala */
List *Head, *Tail; /* Coada de expandat */

/**** Bijectia care asociaza un numar unei permutari ****/

unsigned Perm2Int(Perm P)
{ int i,j,k,Fact=5040;
  unsigned Sum=0;

  for (i=0;i<=6;i++)
    { k=P[i]-1;
      for (j=0;j<i;j++)
        if (P[j]<P[i]) k--;
      Sum+=k*Fact;
      Fact/=(7-i);
    }
  return Sum;
}

void Int2Perm(unsigned Sum, Perm P)
```

```
{ int i,j,k,Order,Fact=5040;
  Perm Used={0,0,0,0,0,0,0,0};

  for (i=0;i<=7;i++)
    { Order=Sum/Fact;
      j=-1;
      for (k=0;k<=Order;k++)
        do j++; while (Used[j]);
      Used[j]=1;
      P[i]=j+1;
      Sum%=Fact;
      if (i!=7) Fact/=(7-i);
    }
}

/**** Lucrul cu liste ****/

void InitList(void)
{
  Head=Tail=malloc(sizeof(List));
  Tail->X=StartValue;
  Tail->Next=NULL;
}

void AddToTail(unsigned K)
{
  Tail->Next=malloc(sizeof(List));
  Tail=Tail->Next;
  Tail->X=K;
  Tail->Next=NULL;
}

void Behead(void)
/* Sterge capul listei */
{ List *LCor=Head;

  Head=Head->Next;
  free(LCor);
}

/**** Cautarea propriu-zisa ****/

void MakeMove(Perm P,Perm Q,int Kind)
/* Kind = 0, 1 sau 2 */
{ int i;
```

```
  for (i=0;i<=7;i++)
    Q[i]=P[Moves[Kind][i]];
}

void Expand(void)
{ List *LCor;
  Perm P1,P2;
  unsigned i,XSon,Done;

  InitList();
  do {
    Int2Perm(Head->X,P1);
    for(i=0;i<=2;i++)
      { MakeMove(P1,P2,i);
        XSon=Perm2Int(P2);
        if (Father[XSon]==Unknown)
          { Father[XSon]=Head->X;
            MoveKind[XSon]=i+65;
            AddToTail(XSon);
          }
      }
    Done=(Head->X==EndValue);
    Behead(); }
  while (!Done);
}

/* Intrarea si iesirea */

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

  for (i=0;i<=7;i++) fscanf(F,"%d",&EndPerm[i]);
  fclose(F);
  StartValue=Perm2Int(StartPerm);
  EndValue=Perm2Int(EndPerm);
  for (i=0;i<40320;) Father[i++]=Unknown;
  Father[StartValue]=Root;
}

void WriteMove(FILE *F,unsigned K,int Len)
{
  if (K!=StartValue)
    { WriteMove(F,Father[K],Len+1);
      fprintf(F,"%c\n",MoveKind[K]);
```

```
130     }
131     else fprintf(F,"%d\n",Len);
132 }
133
134 void Restore(void)
135 { FILE *F=fopen("output.txt","wt");
136
137   WriteMove(F,EndValue,0);
138   fclose(F);
139 }
140
141 void main(void)
142 {
143   InitData();
144   Expand();
145   Restore();
146 }
```

## 6.4 Problema 4

Continuăm cu o problemă care a fost de asemenea dată spre rezolvare la a VIII-a Olimpiadă Internațională de Informatică, Veszprem 1996. Problema în sine nu a fost foarte grea și mulți elevi au luat punctaj maxim. Totuși, enunțul permite unele modificări interesante care practic schimbă cu totul problema.

**ENUNȚ**: Să considerăm următorul joc de două persoane. Tabla de joc constă într-o secvență de întregi pozitivi. Cei doi jucători mută pe rând. Mutarea fiecărui jucător constă în alegerea unui număr de la unul din cele două capete ale secvenței. Numărul ales este șters de pe tablă. Jocul se termină când toate numerele au fost selectate. Primul jucător câștigă dacă suma numerelor alese de el este mai mare sau egală cu cea a numerelor alese de cel de-al doilea jucător. În caz contrar, câștigă al doilea jucător.

Dacă tabla conține inițial un număr par de elemente, atunci primul jucător are o strategie de câștig. Trebuie să scrieți un program care implementează strategia cu care primul jucător câștigă jocul. Răspunsurile celui de-al doilea jucător sunt date de un program rezident. Cei doi jucători comunică prin trei proceduri ale modulului `Play` care v-a fost pus la dispoziție. Procedurile sunt `StartGame`, `MyMove` și `YourMove`. Primul jucător începe jocul apelând procedura fără parametri `StartGame`. Dacă alege numărul de la capătul din stânga, el va apela procedura `MyMove('L')`. Analog, apelul de procedură `MyMove('R')` trimite un mesaj celui de-al doilea jucător prin care îl informează că a ales numărul de la capătul din dreapta. Cel de-al doilea jucător, deci computerul, mută imediat, iar primul jucător poate afla mutarea

acestuia executând procedura `YourMove(C)`, unde `C` este o variabilă de tip `Char` (în C/C++ apelul este `YourMove(&C)`). Valoarea lui `C` este `'L'` sau `'R'`, după cum numărul ales este de la capătul din stânga sau din dreapta.

**Intrarea**: Prima linie din fișierul `INPUT.TXT` conține dimensiunea inițială $N$ a tablei. $N$ este par și $2 \leq N \leq 100$. Următoarele $N$ linii conțin fiecare câte un număr, reprezentând conținutul tablei de la stânga la dreapta. Fiecare număr este cel mult 200.

**Ieșirea**: Când jocul se termină, programul trebuie să scrie rezultatul final în fișierul text `OUTPUT.TXT`. Fișierul conține două numere pe prima linie, reprezentând suma numerelor alese de primul, respectiv de cel de-al doilea jucător. Programul trebuie să joace un joc corect și ieșirea trebuie să corespundă jocului jucat.

**Exemplu**:

| **INPUT.TXT** | **OUTPUT.TXT** |
|---|---|
| 6<br>4<br>7<br>2<br>9<br>5<br>2 | 15 14 |

**Timp limită de execuție**: 20 secunde pentru un test.

Acesta a fost enunțul original, la care va trebui să facem câteva modificări, în parte deoarece nu putem folosi modulul `Play`, în parte pentru a face problema mai restrictivă:

- Mutările vor fi anunțate pe ecran prin tipărirea unui caracter `'L'` sau `'R'`;
- Mutările celui de-al doilea jucător vor fi comunicate de un partener uman, prin introducerea de la tastatură a unui caracter `'L'` sau `'R'`;
- Rezultatul final se va tipări pe ecran, sub aceeași formă (pereche de numere).
- Timpul de gândire pentru fiecare mutare trebuie să fie cât mai mic (practic răspunsul să fie instantaneu);
- **Complexitatea totală** a calculelor efectuate să fie $O(N)$.
- **Timpul de implementare** a fost cam de 1h 40 min. Propunem reducerea lui la 30 minute.

**REZOLVARE**: Este ușor de demonstrat că o rezolvare „greedy” a problemei (la fiecare

mutare jucătorul 1 alege numărul mai mare) nu atrage întotdeauna câștigul. Iată un contraexemplu:

| 7 | 10 | 1 | 2 |
|---|---|---|---|

La prima mutare, jucătorul 1 poate să aleagă fie numărul 7, fie numărul 2. Dacă se va „lăcomi" la 7, jucătorul 2 va lua numărul 10 și inevitabil va câștiga. Soluția pentru primul jucător este să ia numărul 2, apoi, indiferent de ce va juca partenerul său, va putea lua numărul 10 și va câștiga.

Iată o soluție izbitor de simplă de complexitate $O(N)$: La citirea datelor se face suma elementelor aflate pe poziții pare și a celor aflate pe poziții impare. Să presupunem că suma elementelor de ordin par este mai mare sau egală cu cea a elementelor de ordin impar (cazul invers se tratează analog). Atunci, dacă primul jucător ar putea să aleagă toate elementele de ordin par (care sunt într-adevăr $N/2$, adică atâtea câte are el dreptul să aleagă), ar câștiga jocul. Jucătorul 1 poate începe jocul prin a lua primul sau ultimul element din secvență, deci îl va alege pe ultimul, care are număr de ordine par. Al doilea jucător are de ales între primul și al $N-1$-lea element, ambele având număr de ordine impar. Indiferent ce variantă o va adopta, primul jucător va avea din nou acces la un element de pe o poziție pară. Dacă jucătorul 2 alege elementul din stânga (primul), atunci jucătorul 1 va putea lua elementul de după el (al doilea), iar dacă jucătorul 2 alege elementul din dreapta (al $N-1$-lea), atunci jucătorul 1 va putea lua elementul dinaintea el (al $N-2$-lea). Deci primul jucător nu are altceva de făcut decât să repete mutările făcute de cel de-al doilea. Să privim de exemplu desfășurarea jocului pe tabla dată în enunț:

| Jucătorul 1 | 1 | 2 | 3 | 4 | 5 | 6 | Jucătorul 2 |
|---|---|---|---|---|---|---|---|
| 0 | 4 | 7 | 2 | 9 | 5 | 2 | 0 |
| 2 | 4 | 7 | 2 | 9 | 5 | | 0 |
| 2 | 4 | 7 | 2 | 9 | | | 5 |
| 11 | 4 | 7 | 2 | | | | 5 |
| 11 | | 7 | 2 | | | | 9 |
| 18 | | | 2 | | | | 9 |
| 18 | | | | | | | 11 |

$$7+9+2 > 4+2+5$$

Programul în sine nici nu are nevoie să mai rețină vectorul de numere în memorie, din moment ce primul jucător nu are altceva de făcut decât să imite mutările celui de-al doilea. Un

calcul al sumelor la citirea datelor este suficient. Complexitatea $O(N)$ este optimă, deoarece vectorul trebuie parcurs cel puțin o dată pentru citirea configurației inițiale a tablei.

```
#include <stdio.h>

void main(void)
{ FILE *F=fopen("input.txt","rt");
  int SEven,SOdd,N,i,K;

  fscanf(F,"%d\n",&N);
  for (i=1, SEven=SOdd=0; i<=N; i++)
    { fscanf(F, "%d\n", &K);
      if (i&1) SOdd+=K;
        else SEven+=K;
    }
  fclose(F);

  printf("Mutarea mea: %c\n", SEven>=SOdd ? 'R' : 'L');
  for (i=1; i<N/2; i++)
    { printf("Mutarea dvs. (L/R) ? ");
      printf("Mutarea mea: %c\n", getchar());
      getchar(); /* Caracterul newline */
    }
  printf("Mutarea dvs. (L/R) ? ");
  getchar();

  if (SEven>=SOdd)
    printf("%d %d\n", SEven, SOdd);
    else printf("%d %d\n", SOdd, SEven);
}
```

O a doua variantă a enunțului aduce unele condiții suplimentare:

- Se cere să se tipărească numai diferența maximă de scor pe care o poate obține primul jucător, considerând că ambii parteneri joacă perfect;
- **Complexitatea cerută** este $O(N^2)$.
- **Timpul de implementare** este de 45 minute, maxim 1h.

**REZOLVARE**: Trebuie mai întâi să lămurim ce se înțelege prin „joc perfect”. Jucătorul 1 are întotdeauna victoria la îndemână (metoda este arătată mai sus), dar nu la orice scor. Jucătorul 2 urmărește să minimizeze diferența de scor. Fie $D$ diferența de scor cu care se

termină un joc. $D$ poate lua diferite valori pentru aceeași configurație inițială a tablei, în funcție de mutările făcute de cei doi jucători. Fie $D_{MAX}$ diferența maximă de scor pe care o poate obține primul jucător indiferent de mutările celui de-al doilea. Exact această valoare trebuie aflată. $D_{MAX}$ nu este propriu-zis o diferență maximă. Jucătorul 1 poate să câștige și la diferențe mai mari decât $D_{MAX}$, dar trebuie ca jucătorul 2 să-l „ajute". Să reluăm exemplul cu 4 numere:

| 7 | 10 | 1 | 2 |
|---|---|---|---|

În acest caz, primul jucător are asigurat scorul 12-8 (deci diferența 4). Pentru aceasta, el începe prin a lua numărul 2, apoi, orice ar replica celălalt, va lua numărul 10, jucătorului 2 revenindu-i așadar numerele 1 și 7. El poate obține și scorul 17-3 (jucătorul 1 ia numărul 7, celălalt ia 2, jucătorul 1 ia 10, iar celălalt ia 1), dar aceasta se întâmplă numai dacă jucătorul 2 face o greșeală. După cum am arătat mai sus, dacă primul jucător începe luând numărul 7, el pierde în mod normal partida. Iată deci că în acest caz $D_{MAX} = 4$.

Pentru a putea afla diferența maximă de scor, este bine să privim mereu în adâncime. Există patru variante în care ambii parteneri pot face câte o mutare:

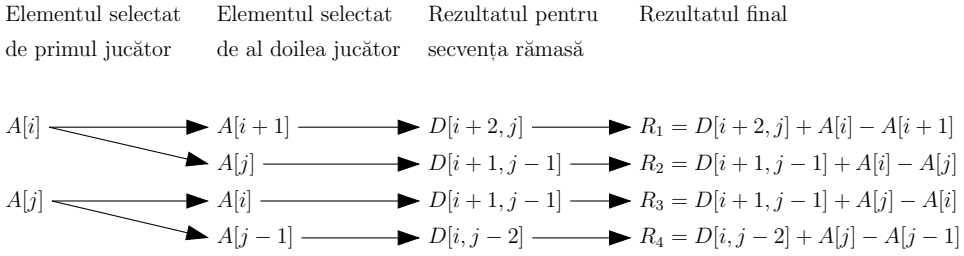
1. Ambii jucători aleg numere din partea stângă;
2. Ambii aleg numere din partea dreaptă;
3. Primul jucător alege numărul din stânga, iar celălalt pe cel din dreapta;
4. Primul jucător alege numărul din dreapta, iar celălalt pe cel din stânga;

În urma oricărei variante de mutare, secvența se scurtează cu două elemente. Dacă am putea cunoaște dinainte care este rezultatul jocului pentru fiecare din secvențele scurte, am putea să decidem care variantă de joc este cea mai convenabilă pentru secvența inițială, ținând cont și de modul de joc al jucătorului al doilea. Tocmai de aici vine și ideea de rezolvare. Să notăm cu $A[1], A[2], ..., A[N]$ secvența citită la intrare. Vom construi o matrice $D$ cu $N$ linii și $N$ coloane, unde $D[i, j]$ este diferența maximă pe care o poate obține jucătorul 1 pentru secvența $A[i], A[i+1], \ldots, A[j]$. Bineînțeles, sunt luate în considerare numai secvențele de lungime pară. Scopul nostru este să-l aflăm pe $D[1, N]$.

Elementele matricei pe care le putem afla fără multă bătaie de cap sunt $D[1, 2]$, $D[2, 3]$, $\ldots$, $D[N-1, N]$. Într-adevăr, dintr-o secvență de numai două numere, primului jucător îi revine cel mai mare, iar celui de-al doilea - cel mai mic. Așadar

$$D[i, i+1] = |A[i] - A[i+1]| \tag{6.5}$$

Cum calculăm $D[i, j]$ dacă cunoaștem valorile matricei $D$ pentru toate subsecvențele incluse în secvența $A[i], A[i+1], ..., A[j]$? După cum am mai spus, avem patru variante:




Trebuie să ținem minte că, dacă primul jucător optează să-l aleagă pe $A[i]$ (una din primele două variante), atunci jucătorul 2 va juca în așa fel încât pierderea să fie minimă, iar scorul final va fi $\min(R_1, R_2)$. Dacă jucătorul 1 alege varianta 3 sau 4, scorul final va fi $\min(R_3, R_4)$. Dar jucătorul 1 este primul la mutare, deci va alege varianta care îi maximizează profitul. Rezultatul este

$$D[i, j] = \max(\min(R_1, R_2), \min(R_3, R_4)) \tag{6.6}$$

adică

$$\begin{aligned} D[i, j] = \max(&A[i] + \min(D[i+2, j] - A[i+1], D[i+1, j-1] - A[j]), \\ &A[j] + \min(D[i+1, j-1] - A[i], D[i, j-2] - A[j-1])) \end{aligned} \tag{6.7}$$

Matricea $D$ se completează pe diagonală, pornind de la diagonala principală și mergând până în colțul de N-E. Iată cum arată matricea atașată datelor de intrare din enunț:

$$D = \begin{pmatrix} X & 3 & X & 10 & X & 7 \\ X & X & 5 & X & 9 & X \\ X & X & X & 7 & X & 4 \\ X & X & X & X & 4 & X \\ X & X & X & X & X & 3 \\ X & X & X & X & X & X \end{pmatrix} \tag{6.8}$$

Pentru exemplul din enunț, răspunsul este deci $D_{MAX} = 7$. Cum elementele matricei sunt parcurse cel mult o dată, rezultă o complexitate de $O(N^2)$.

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```
#define NMax 101

int D[NMax][NMax], A[NMax], N;

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

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

int Min(int A, int B)
{
  return A<B ? A : B;
}

int Max(int A, int B)
{
  return A>B ? A : B;
}

void FindMax(void)
{ int i,j,k;

  for (i=1;i<N;i++)
    D[i][i+1]=abs(A[i]-A[i+1]);
  for (k=3;k<=N-1;k++)
    for (i=1;i+k<=N;i++)
      { j=i+k;
        D[i][j]=Max(A[i]+Min(D[i+2][j]-A[i+1],
                             D[i+1][j-1]-A[j]),
                    A[j]+Min(D[i+1][j-1]-A[i],
                             D[i][j-2]-A[j-1]));
      }
  printf("Diferenta maxima este %d\n",D[1][N]);
}

void main(void)
{
  ReadData();
  FindMax();
}
```

Programul prezentat mai sus poate fi optimizat, dacă timpul o permite și dacă acest lucru este necesar. Lăsăm cititorul să încerce să rezolve aceeași problemă folosind o cantitatate de memorie direct proporțională cu $N$.

## 6.5 Problema 5

Această problemă a fost propusă la Olimpiada Națională de Informatică, Slatina 1995, la clasa a XI-a. Pe atunci programarea dinamică era o tehnică de programare destul de puțin cunoscută de către majoritatea elevilor.

**ENUNȚ**: O regiune deșertică este reprezentată printr-un tablou de dimensiuni $M \times N$ ($1 \leq M \leq 100$, $1 \leq N \leq 100$). Elementele tabloului sunt numere naturale mai mici ca 255, reprezentând diferențele de altitudine față de nivelul mării (cota 0). Să se stabilească:

a) Un traseu pentru a traversa deșertul de la nord la sud (de la linia 1 la linia $M$), astfel:

- Se pornește dintr-un punct al liniei 1;
- Deplasarea se poate face în una din direcțiile: E, SE, S, SV, V;
- Suma diferențelor de nivel (la urcare și la coborâre) trebuie să fie minimă.

b) Un traseu pentru a traversa deșertul de la nord la sud în condițiile punctului (a), la care se adaugă condiția:

- Lungimea traseului să fie minimă.

**Intrarea**: Fișierul de intrare `INPUT.TXT` conține un singur set de date cu următoarea structură:

| | |
|---|---|
| linia 1: | $M$ $N$ |
| liniile $2 \ldots M+1$: | elementele tabloului (pe linii) separate prin spații |

**Ieșirea**: Fișerul de ieșire `OUTPUT.TXT` va conține rezultatele în următorul format:

```
(a)
<suma diferențelor de nivel>
TRASEU: (i_1, j_1)->(i_2, j_2)->...->(i_k, j_k)
(b)
<suma diferențelor de nivel> <lungime traseu>
TRASEU: (i_1, j_1)->(i_2, j_2)->...->(i_p, j_p)
```

unde $i_x$ și $j_x$ sunt linia și coloana fiecărei celule vizitate.

**Exemplu**:

| **INPUT.TXT** | **OUTPUT.TXT** |
|---|---|
| 4 4 | (a) |
| 10 7 2 5 | 5 |
| 13 20 25 3 | TRASEU: (1,3)->(2,4)->(3,3)->(3,2)->(4,1) |
| 2 4 2 20 | (b) |
| 5 10 9 11 | 9 3 |
| | TRASEU: (1,3)->(2,4)->(3,3)->(4,2) |

Acesta a fost enunțul original. Iată acum completările propuse și o precizare importantă:

- **Timpul de implementare**: 45 minute - 1h (la concurs a fost cam 1h 30 min);
- **Timpul de rulare**: 2-3 secunde;
- **Complexitatea cerută**: $O(N^2)$;
- La punctul (b), condiția nou adăugată este mai puternică decât cea de la punctul (a). Cu alte cuvinte, în primul rând contează lungimea drumului și abia apoi, dintre toate drumurile de lungime minimă, trebuie ales cel pentru care suma denivelărilor este minimă. Pentru a vă convinge că ordinea în care sunt impuse condițiile este importantă, să privim exemplul de mai sus. Dacă este mai importantă minimizarea sumei denivelărilor, atunci minimul este 5, iar drumul este soluția de la punctul (a). Dacă este mai importantă minimizarea lungimii drumului, atunci lungimea minimă este 3, iar din toate drumurile de lungime 3, cel mai puțin costisitor este cel indicat la punctul (b).

**REZOLVARE**: Vom lăsa punctul (b) al acestei probleme în seama cititorului, întrucât el nu este altceva decât o simplificare a punctului (a). Să ne ocupăm acum de punctul (a). Vom numi efort diferența de altitudine (în modul) la deplasarea cu un pas. Scopul este deci găsirea unor drumuri de efort total minim. Matricea de altitudini o vom nota cu $Alt$.

O primă posibilitate de abordare a problemei este „greedy", dar aceasta nu e cea mai fericită alegere, chiar dacă este una comodă. Ideea de bază este următoarea: Încercăm să pornim din colțul de NV și să ne deplasăm la fiecare pas pe acea direcție pentru care efortul este minim, până ajungem la ultima linie. Apoi pornim din a doua coloană a primei linii și aplicăm aceeași tactică, apoi din a treia coloană și așa mai departe până la colțul de NE. În final tipărim soluția cea mai bună găsită. Iată însă un exemplu pe care această metodă dă greș:

$$Alt = \begin{pmatrix} 2 & 1 & 2 \\ 10 & 1 & 10 \\ 10 & 10 & 10 \end{pmatrix} \tag{6.9}$$

Pe această matrice, algoritmul greedy va găsi traseele (1,1) → (2,2) → (3,2) de efort total 10, (1,2) → (2,2) → (3,2) de efort total 9 și (3,1) → (2,2) → (3,2) de efort total 10. Așadar, rezultatul optim ar fi 9, ceea ce este fals deoarece alegerea traseului (1,1) → (2,1) → (3,1) ar duce la un efort total de 8, deci mai mic.

Motivul pentru care acest algoritm nu funcționează cum trebuie este că el nu privește în perspectivă. În cazul de mai sus, coborârea în „văile" de altitudine 1 era o primă mutare tentantă, dar fără nici un rezultat, deoarece până la urmă tot era necesară suirea la altitudinea 10. Soluția corectă este ca, pentru a afla efortul minim cu care se poate ajunge la o locație oarecare, să analizăm toate drumurile care duc la acea locație. Dacă am cunoaște efortul minim cu care se poate ajunge la fiecare din vecinii din E, NE, N, NV, și V ai unei celule, atunci putem cu ușurință, pe baza unor comparații, să deducem din ce parte este cel mai avantajos să venim în respectiva celulă și cu ce efort minim.

Mai concret, vom construi o matrice cu aceleași dimensiuni ca și matricea *Alt*, pe care o vom denumi *Eff*. În această matrice, $Eff[i,j]$ reprezintă efortul minim necesar pentru a ajunge de pe un punct oarecare de pe linia 1 în celula $(i,j)$. Deducem că $Eff[1,j] = 0, \forall 1 \leq j \leq N$. Noi trebuie să completăm matricea *Eff*, apoi să căutăm minimul dintre toate elementele de pe linia $M$ (care este chiar efortul minim căutat) și să reconstituim traseul de urmat.

Ca să vedem cum anume se face completarea matricei, facem mai întâi observația că, odată ce am ajuns pe o linie, putem fie să coborâm direct pe linia imediat inferioară, fie să ne deplasăm câțiva pași numai spre stânga sau numai spre dreapta, apoi să coborâm pe linia următoare. În orice locație $(X,Y)$ a matricei putem veni dinspre E, NE, N, NV, sau V. Pentru acești cinci vecini presupunem deja calculate eforturile minime necesare, respectiv $Eff[X,Y+1], Eff[X-1,Y+1], Eff[X-1,Y], Eff[X-1,Y-1], Eff[X,Y-1]$. Atunci, în funcție de direcția din care venim, efortul depus până la punctul $(X,Y)$ va fi:

| | | |
|---|---|---|
| dinspre est: | $Eff[X,Y+1] + \lvert Alt[X,Y+1] - Alt[X,Y]\rvert$ | (1) |
| dinspre nord-est: | $Eff[X-1,Y+1] + \lvert Alt[X-1,Y+1] - Alt[X,Y]\rvert$ | (2) |
| dinspre nord: | $Eff[X-1,Y] + \lvert Alt[X-1,Y] - Alt[X,Y]\rvert$ | (3) |
| dinspre nord-vest: | $Eff[X-1,Y-1] + \lvert Alt[X-1,Y-1] - Alt[X,Y]\rvert$ | (4) |
| dinspre vest: | $Eff[X,Y-1] + \lvert Alt[X,Y-1] - Alt[X,Y]\rvert$ | (5) |

În principiu, nu avem decât să calculăm minimul dintre aceste expresii ca să aflăm valoarea lui $Eff[X,Y]$. În felul acesta, matricea *Eff* se va completa pe linie, de sus în jos. Totuși, apare o problemă: pentru a-l afla pe $Eff[X,Y]$ avem nevoie de $Eff[X,Y-1]$ (dacă ne deplasăm spre est), iar pentru a-l afla pe $Eff[X,Y-1]$ avem nevoie de $Eff[X,Y]$ (dacă ne deplasăm spre vest)! Bineînțeles, avem sentimentul că ne învârtim după propria coadă. Totuși, dezlegarea nu e complicată, ținând cont de observația făcută mai sus, că pe aceeași linie deplasarea se face într-o singură direcție. Este suficient să parcurgem fiecare linie de două ori: prima oară o parcurgem de la stânga la dreapta, în ipoteza că deplasarea pe linia respectivă se face spre

est, apoi încă o dată de la dreapta la stânga, în ipoteza că deplasarea pe linia respectivă se face spre vest. La prima parcurgere, vom minimiza efortul pentru fiecare căsuță cu expresia (5), iar la a doua - cu expresia (1). Minimizarea cu expresiile (2), (3) și (4) se poate face la oricare din parcurgeri, deoarece elementele liniei superioare nu se mai modifică.

După cum am spus, efortul minim se obține căutând minimul de pe ultima linie a matricei *Eff* (aceasta deoarece nu contează în ce punct de pe ultima linie este sosirea). Punctul în care se atinge acest minim este tocmai punctul de sosire. Reconstituirea efectivă a drumului se face în sens invers: se pleacă din punctul de sosire $(i_k, j_k)$ și se caută un punct vecin lui pe una din cele cinci direcții permise, $(i_{k-1}, j_{k-1})$ astfel încât

$$Eff[i_k, j_k] = Eff[i_{k-1}, j_{k-1}] + |Alt[i_k, j_k] - Alt[i_{k-1}, j_{k-1}]| \tag{6.10}$$

Cu alte cuvinte, se testează pentru care din expresiile (1) - (5) se verifică egalitatea. Se reia, recursiv, același procedeu pentru locația $(i_{k-1}, j_{k-1})$.

Iată cum se completează matricea *Eff* pentru exemplul dat și cum se reconstituie drumul:

$$Alt = \begin{pmatrix} 10 & 7 & 2 & 5 \\ 13 & 20 & 25 & 3 \\ 2 & 4 & 2 & 20 \\ 5 & 10 & 9 & 11 \end{pmatrix} \tag{6.11}$$

$$Eff = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 3 & 10 & 15 & 1 \\ 6 & 4 & 2 & 18 \\ 5 & 10 & 9 & 11 \end{pmatrix} \tag{6.12}$$

Minimul de pe linia a 4-a a matricei *Eff* este $Eff[4, 1] = 5$, deci sosirea se face în colțul de SV. Din ce parte am ajuns aici? Se testează toți vecinii și se constată că $Eff[4, 1] = Eff[3, 2] + |Alt[4, 1] - Alt[3, 2]|$, deci s-a venit de la locația (3,2). Apoi se constată că:

$$\begin{aligned} Eff[3, 2] &= Eff[3, 3] + |Alt[3, 2] - Alt[3, 3]| \\ Eff[3, 3] &= Eff[2, 4] + |Alt[3, 3] - Alt[2, 4]| \\ Eff[2, 4] &= Eff[1, 3] + |Alt[2, 4] - Alt[1, 3]| \end{aligned} \tag{6.13}$$

Din aceste relații rezultă că traseul urmat este $(1, 3) \to (2, 4) \to (3, 3) \to (3, 2) \to (4, 1)$.

Pentru o mai mare ușurință a implementării, se vor adăuga două coloane fictive la matricea *Alt*: coloanele 0 și $N + 1$. Facem acest lucru pentru a ne putea referi la celula $(X, Y - 1)$ atunci

când $(X, Y)$ este o celulă din prima coloană (respectiv la celula $(X, Y+1)$ atunci când $(X, Y)$ este o celulă de pe ultima coloană) fără a primi un mesaj de eroare. Trebuie însă să fim atenți ca nu cumva noile coloane adăugate să perturbe datele de ieșire și să rezulte că traseul optim trece prin coloana 0 sau $N+1$. Pentru a scăpa de grija celulelor de pe aceste două coloane și a ne asigura că ele nu vor putea fi selectate pentru traseul optim, le vom atribui altitudini foarte mari. Deoarece diferența maximă de nivel la fiecare pas este 255, rezultă că efortul total maxim ce se poate obține este $255 \times 99 = 25.245$. Așadar, o altitudine a coloanelor laterale de 30.000 este suficientă.

```
#include <stdio.h>
#include <math.h>
#define NMax 101
#define Infinity 30000
typedef int Matrix[NMax][NMax+1];

Matrix Alt, Eff;
int M, N;
FILE *OutF;

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

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

void Optimize(int X1, int Y1, int X2, int Y2)
/* Testeaza daca in (X1,Y1) se poate ajunge
   cu efort mai mic dinspre (X2,Y2) */
{
  if (Eff[X2][Y2]+abs(Alt[X1][Y1]-Alt[X2][Y2])<Eff[X1][Y1])
    Eff[X1][Y1]=Eff[X2][Y2]+abs(Alt[X1][Y1]-Alt[X2][Y2]);
}

void Traverse(void)
{ int i,j;

  for (j=1; j<=N;) Eff[1][j++]=0;
  for (i=1; i<=M; i++)
```

```
35      Eff[i][0]=Eff[i][N+1]=Infinity; /* Bordeaza matricea */
36    for (i=2; i<=M; i++)
37      {
38        for (j=1; j<=N; j++)
39          {
40            Eff[i][j]=Infinity;
41            Optimize(i, j, i-1, j);       /* De la N  */
42            Optimize(i, j, i-1, j-1);     /* De la NV */
43            Optimize(i, j, i-1, j+1);     /* De la NE */
44            Optimize(i, j, i, j-1);       /* De la V  */
45          }
46        for (j=N; j; j--)
47          Optimize(i, j, i, j+1);         /* De la E  */
48      }
49  }
50
51  void GoBack(int X, int Y)
52  /* Reconstituie drumul */
53  {
54    if (X>1)
55      if (Eff[X][Y]==Eff[X][Y-1]
56                     +abs(Alt[X][Y-1]-Alt[X][Y]))
57      GoBack(X, Y-1);
58      else if (Eff[X][Y]==Eff[X-1][Y-1]
59                          +abs(Alt[X-1][Y-1]-Alt[X][Y]))
60      GoBack(X-1, Y-1);
61      else if (Eff[X][Y]==Eff[X-1][Y]
62                          +abs(Alt[X-1][Y]-Alt[X][Y]))
63      GoBack(X-1, Y);
64      else if (Eff[X][Y]==Eff[X-1][Y+1]
65                          +abs(Alt[X-1][Y+1]-Alt[X][Y]))
66      GoBack(X-1, Y+1);
67      else if (Eff[X][Y]==Eff[X][Y+1]
68                          +abs(Alt[X][Y+1]-Alt[X][Y]))
69      GoBack(X, Y+1);
70    if (X>1) fprintf(OutF, "->");
71    fprintf(OutF,"(%d,%d)", X, Y);
72  }
73
74  void WriteSolution(void)
75  { int j,k;
76
77    OutF=fopen("output.txt", "wt");
78    /* Cauta punctul de sosire */
79    fputs("(a)\n",OutF);
```

```
80    for (j=2, k=1; j<=N; j++)
81      if (Eff[M][j]<Eff[M][k]) k=j;
82    fprintf(OutF, "%d\n", Eff[M][k]);
83    fputs("TRASEU: ",OutF);
84    GoBack(M, k);
85    fprintf(OutF,"\n");
86    fclose(OutF);
87  }
88
89  void main(void)
90  {
91    ReadData();
92    Traverse();
93    WriteSolution();
94  }
```

## 6.6 Problema 6

Propunem în continuare o problemă care s-a dat la Olimpiada Națională de Informatică, Suceava 1996, la clasa a XII-a. Menționăm că un singur concurent a reușit să o ducă la bun sfârșit în timpul concursului. Problema se numește „Cartierul Enicbo".

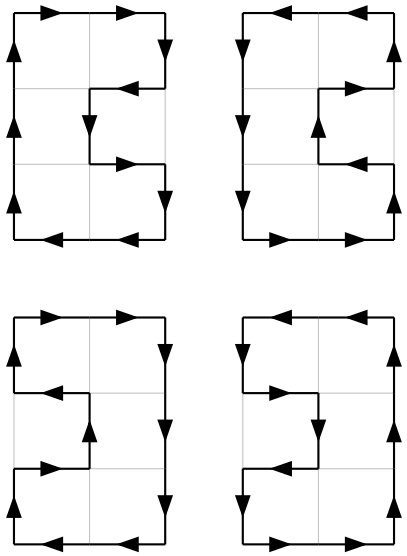
**ENUNȚ**: În orașul Acopan s-a construit un nou cartier. Noul cartier are patru bulevarde paralele și un număr de $N$ străzi perpendiculare pe ele. Există deci în total $4N$ intersecții. Furgoneta oficiului poștal trebuie să distribuie poșta în fiecare zi; în acest scop, furgoneta pleacă de la oficiul poștal aflat la intersecția bulevardului 1 cu strada 1 și, urmând rețeaua stradală, trece exact o dată prin fiecare intersecție astfel încât să încheie traseul în punctul de plecare.

Conducerea oficiului poștal roagă participanții la olimpiadă să o ajute să afle în câte moduri distincte se poate stabili traseul furgonetei.

**Intrarea**: Programul va citi de la tastatură valoarea lui $N$ ($2 \leq N \leq 200$).

**Ieșirea**: Pe ecran se va afișa soluția (numărul de trasee distincte pentru valoarea respectivă a lui $N$).

**Exemplu**: Pentru $N = 3$ există 4 soluții (se citește de la tastatură numărul 3 și se afișează pe ecran numărul 4). Iată soluțiile efective:



**Timp de execuție**: 30 secunde pentru un text

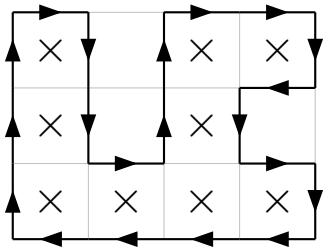
**Timp de implementare**: 1h 30 min.

**Complexitate cerută**: $O(N^2)$

**REZOLVARE**: Primul lucru care ne vine în gând este „se cere numărul de cicluri hamiltoniene într-un graf, deci problema e exponențială". Rezolvarea backtracking nu e deloc greu de implementat, dar nu are nici o șansă să meargă pentru valori mari ale lui $N$. Afirmația de mai sus este corectă, dar incompletă; din această cauză concluzia este falsă. Se scapă din vedere faptul că graful nu este oarecare, ci are un aspect foarte particular.

Și în această problemă vom încerca să utilizăm soluțiile locale (pentru valori mici ale lui $N$) pentru aflarea soluției globale. Respectiv, vom rezolva problema pentru $N = 2$, apoi o vom extinde pentru $N = 3, 4$ și așa mai departe. Pentru început, însă, încercăm să simplificăm enunțul, reducând problema la una echivalentă, dar mai simplă.

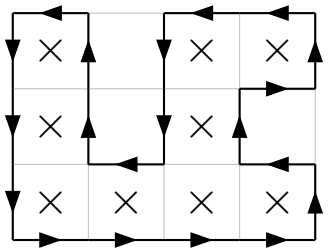
Să considerăm o posibilă soluție pentru $N = 5$:



În loc să lucrăm cu segmente în această rețea, vom lucra cu ochiuri. Furgoneta parcurge un ciclu, deci închide în circuitul ei un număr de ochiuri. Am marcat aceste ochiuri cu un „×" în figura de mai sus. Așadar, oricărui drum al furgonetei i se poate atașa o matrice cu 3 linii și $N - 1$ coloane, în care unele celule sunt bifate cu „×", iar altele nu. Să vedem în primul rând

care este corespondența între numărul de circuite hamiltoniene și numărul de matrice de acest tip.

Se observă că pentru orice circuit există un altul căruia îi este atașată aceeași matrice. Circuitul pereche este tocmai circuitul parcurs în sens invers, care închide în interior aceleași ochiuri de rețea:



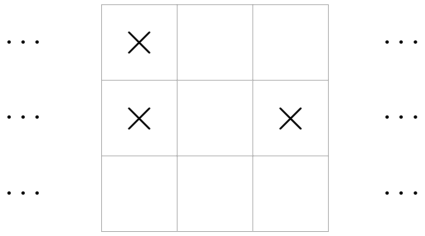
Acest lucru se întâmplă deoarece transformarea graf-matrice ignoră sensul de parcurgere a circuitului hamiltonian. De aici rezultă că pentru a calcula numărul de circuite hamiltoniene trebuie să calculăm numărul de matrice și să-l înmulțim cu 2.

În continuare, să analizăm câteva proprietăți ale matricelor în discuție.

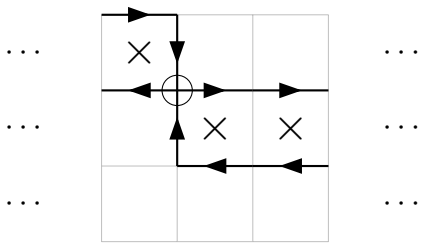
**Proprietatea 1.** *Elementele bifate cu „×” în matrice formează o singură figură conexă.*

*Demonstraţie.* Dacă figura nu ar fi conexă, adică dacă ar exista mai multe figuri, ele nu ar putea fi înconjurate de furgonetă într-un singur drum. De remarcat că **toate** pătratele înconjurate de furgonetă trebuie bifate cu ×, deci furgoneta nu poate înconjura pătrate nebifate. □

De aceea, traseul de mai jos (care prezintă o porțiune oarecare de circuit) este imposibil.



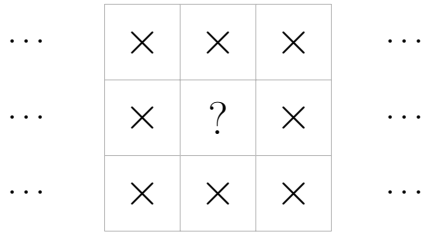
Conexitatea se referă numai la vecinătatea pe latură, nu și pe colț. Spre exemplu, figura de mai jos este incorectă, deoarece, pentru a o înconjura, furgoneta trebuie să treacă de două ori prin punctul încercuit:



**Proprietatea 2.** *Elementele bifate cu „×” formează o structură aciclică.*

*Demonstraţie.* Dacă structura ar fi ciclică, ar rezulta că elementele bifate cu „×” închid între ele elemente nebifate, pe care furgoneta însă nu poate să le ocolească. □

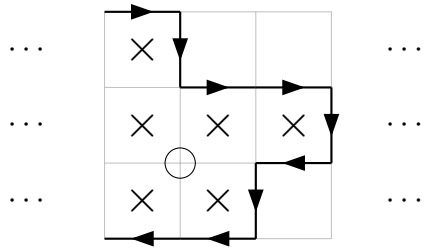
Iată un exemplu de ciclicitate:




Practic, situația de mai sus obligă furgoneta să facă două drumuri: unul pe exterior și unul în jurul ochiului marcat cu „?”.

**Proprietatea 3.** *Nici un nod interior al rețelei nu poate avea toate cele patru ochiuri vecine marcate cu „×”.*

*Demonstraţie.* Dacă ar exista un asemenea nod, el nu ar putea fi parcurs de furgonetă, deci ciclul nu ar mai fi hamiltonian. Este cazul nodului încercuit în figura următoare: □

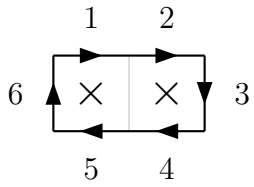



**Proprietatea 4.** *Structura elementelor bifate cu „×” în cadrul matricei este arborescentă.*

*Demonstraţie.* Rezultă imediat din punctele anterioare: figura este conexă și aciclică. □

**Proprietatea 5.** *Numărul de celule bifate este* $P = 2N - 1$.

*Demonstraţie.* Folosim inducția matematică. Să presupunem că structura noastră ar avea un singur pătrat bifat. Atunci structura ar avea patru laturi „la vedere”. Traseul furgonetei care ocolește structura ar avea patru laturi. O structură de două pătrate (desigur lipite) va avea șase laturi la vedere:




Să ne imaginăm acum că orice structură cu $k$ pătrate are $S_k$ laturi la vedere. Trebuie să demonstrăm că toate structurile cu $k+1$ pătrate au același număr de laturi la vedere și să aflăm efectiv acest număr, $S_{k+1}$. Cel de-al $k+1$-lea pătrat trebuie alipit la structura deja existentă în așa fel încât să nu se închidă nici un ciclu. El se va lipi deci de o latură la vedere a unui pătrat din structură. În acest fel, va dispărea o latură la vedere, dar vor apărea trei în loc. Numărul de laturi la vedere va crește prin urmare cu 2. Această cifră nu depinde de locul în care este alipit al $k+1$-lea pătrat, nici de forma structurii deja existente, deci am demonstrat că toate structurile arborescente cu $k$ pătrate au același număr de laturi la vedere. Pentru a afla efectiv acest număr, pornim de la relațiile recurente stabilite prin inducție și eliminăm recurența:

$$\left.\begin{array}{r} S_{k+1} = S_k + 2 \\ S_1 = 4 \end{array}\right\} \implies S_k = 2k + 2 \tag{6.14}$$

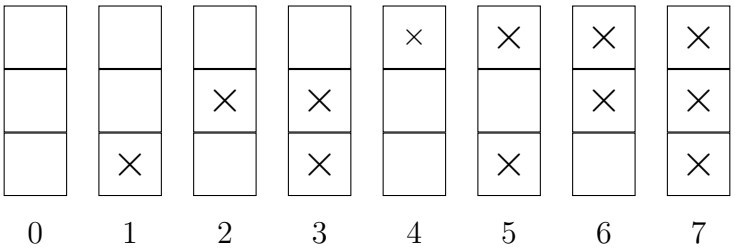
Deoarece numărul total de noduri al rețelei este de $4N$, rezultă că structura noastră trebuie să aibă $4N$ laturi la vedere. Notând cu $P$ numărul de pătrate bifate din matrice și rezolvând ecuația de mai jos, rezultă valoarea lui $P$:

$$2P + 2 = 4N \implies P = 2N - 1 = 2(N-1) + 1 \tag{6.15}$$

Cum numărul de coloane al matricei este $N-1$, deducem că în medie pe fiecare coloană se vor afla două pătrate bifate, cu excepția uneia pe care se vor afla trei pătrate bifate. □

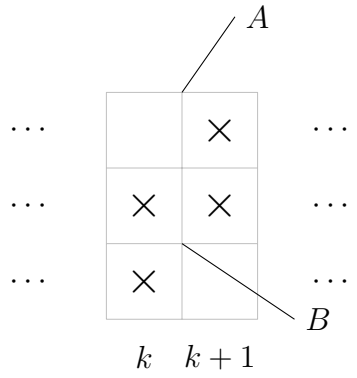
La nivel local, proprietatea este de asemenea respectată: numărul de pătrate bifate din primele $k$ coloane ale matricei este $2k$, existând posibilitatea să mai fie un pătrat suplimentar. De exemplu, în figura dată mai sus pentru $N = 5$, în primele două coloane se află patru elemente „×”, deci o medie de două pătrate pe fiecare coloană. În primele trei coloane există șapte elemente „×”, adică o medie de două pătrate pe coloană și un surplus de un pătrat. Lăsăm ca temă cititorului să demonstreze că în primele $k$ coloane există întotdeauna fie $2k$, fie $2k+1$ pătrate. Orice număr mai mare duce la ciclicitatea figurii, orice număr mai mic duce la neconexitatea ei.

Pe fiecare coloană există opt combinații posibile de elemente bifate și nebifate, pe care le vom codifica cu numere de la 0 la 7, conform unei numărători binare:

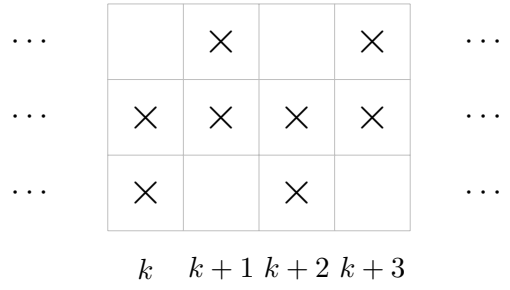



Să vedem acum care dintre aceste combinații rămân valabile. O coloană de tipul 0 nu poate exista, deoarece ea ar „rupe" matricea în două bucăți separate, deci proprietatea de conexitate nu ar fi respectată.

Dacă pe coloana $k$ se află o combinație de tipul 3, ce s-ar putea afla pe coloana $k+1$?




Pentru ca punctul $A$ să se afle pe traseul furgonetei, este obligatoriu să bifăm pătratul de sub el. Pentru a menține conexitatea figurii apărute, trebuie bifat și pătratul din centrul coloanei $k+1$. Cea de-a treia celulă a coloanei $k+1$ nu poate fi bifată, deoarece punctul $B$ ar fi înconjurat din patru părți de celule bifate, lucru care s-a stabilit că este imposibil. Se vede că singura combinație posibilă pentru coloana $k+1$ este 6. Ce combinație putem pune pe coloana $k+2$? Printr-un raționament analog, deducem că numai combinația 3:




Iată că, pentru a putea respecta condițiile de corectitudine a matricei, am fi nevoiți să continuăm la nesfârșit cu coloane cu combinațiile 3-6-3-6 etc. Deci niciuna din aceste combinații nu poate apărea în matrice.

În continuare, vom defini mai multe șiruri de forma $S_k(i,t)$, unde:

- $k$ este numărul unei coloane;
- $i$ este un număr de combinație (respectiv 1, 2, 4, 5 sau 7);
- $t$ este un număr care poate avea valoarea 0 sau 1.

$S_k(i,t)$ semnifică „numărul de matrice (corecte) cu $k$ coloane astfel încât pe coloana cu numărul $k$ să se afle combinația $i$, iar surplusul de pătrate bifate peste media de două pătrate pe fiecare coloană să fie $t$". De exemplu, $S_7(5,1)$ reprezintă numărul de matrice corecte (care respectă regulile de construcție) cu 7 coloane, astfel încât pe ultima coloană să se afle combinația 5 și să existe un surplus de 1 pătrat (adică numărul total de pătrate să fie $2 \times 7 + 1 = 15$).

Facem observația că pe a $N-1$-a coloană se pot afla doar combinațiile 5 sau 7 (pentru a acoperi colțurile de NE și SE ale grafului), iar surplusul de pătrate trebuie să fie 1 (deoarece în $N-1$ coloane trebuie să se afle $P = 2(N-1)+1$ pătrate bifate). Deci scopul nostru este să calculăm suma $S_{N-1}(5,1) + S_{N-1}(7,1)$ și să o înmulțim cu 2 ca să aflăm numărul de cicluri hamiltoniene.

De asemenea, remarcăm că șirurile $S_k(1,1)$, $S_k(2,1)$ și $S_k(4,1)$ nu sunt definite. Aceasta deoarece combinațiile 1, 2 și 4 au un singur pătrat bifat pe coloană, adică mai puțin decât media de două pătrate. Este imposibil ca după adăugarea unei asemenea coloane să mai existe un surplus. (deoarece ar rezulta că în primele $k$-1 coloane exista un surplus de două pătrate). La polul opus, șirul $S_k(7,0)$ nu este definit, deoarece combinația 7 are toată coloana bifată, adică peste medie, deci nu se poate să nu apară un surplus de pătrate bifate.

Mai trebuie stabilite formulele de recurență între șirurile $S_k(1,0)$, $S_k(2,0)$, $S_k(4,0)$, $S_k(5,0)$, $S_k(5,1)$ și $S_k(7,1)$. Termenii inițiali ai recurenței sunt:

- $S_1(1,0) = 0$ deoarece matricea nu poate începe cu combinația 1
- $S_1(2,0) = 0$ deoarece matricea nu poate începe cu combinația 2
- $S_1(4,0) = 0$ deoarece matricea nu poate începe cu combinația 4
- $S_1(5,0) = 1$ deoarece există o singură matrice de o coloană cu combinația 5
- $S_1(5,1) = 0$ deoarece combinația 5 are două pătrate, deci nu există surplus
- $S_1(7,1) = 1$ deoarece există o singură matrice de o coloană cu combinația 7

Pentru a stabili relația de recurență pentru șirul $S_k(1,0)$, ne întrebăm: cărei coloane îi poate urma coloana $k$ de tip 1 astfel încât să nu mai existe surplus? Dacă observăm că pe coloana $k$

avem un singur element bifat (deci sub medie), rezultă că pe coloana $k-1$ exista un surplus de un pătrat. Deci coloana $k-1$ putea fi de tipul 5 sau 7, acestea fiind singurele tipuri de coloană după care poate exista un surplus. Rezultă formula:

$$S_k(1,0) = S_{k-1}(5,1) + S_{k-1}(7,1) \tag{6.16}$$

Printr-o simetrie perfectă se calculează aceeași formulă și pentru șirul $S_k(4,0)$:

$$S_k(4,0) = S_{k-1}(5,1) + S_{k-1}(7,1) \tag{6.17}$$

La șirul $S_k(2,0)$, mai trebuie făcută observația că o coloană de tip 2 nu poate urma unei coloane de tip 5, deoarece se strică conexitatea figurii. Rezultă:

$$S_k(2,0) = S_{k-1}(7,1) \tag{6.18}$$

Șirul $S_k(5,0)$ provine din adăugarea unei coloane de tipul 5 după o coloană de tipul 1, 4 sau 5. Coloana $k-1$ nu poate fi de tipul 2 deoarece figura rezultată nu este conexă, nici de tipul 7 deoarece ar rezulta că în primele $k-2$ coloane media de celule bifate este mai mică decât 2.

$$S_k(5,0) = S_{k-1}(1,0) + S_{k-1}(4,0) + S_{k-1}(5,0) \tag{6.19}$$

Șirul $S_k(5,1)$ provine din adăugarea unei coloane de tipul 5 după o coloană de tipul 5 sau 7, deoarece tipul de coloană 5 are două pătrate bifate, deci conservă surplusul:

$$S_k(5,1) = S_{k-1}(5,1) + S_{k-1}(7,1) \tag{6.20}$$

În sfârșit, o coloană de tip 7 poate urma oricărui tip de coloană pentru care surplusul este 0, adică:

$$S_k(7,1) = S_{k-1}(1,0) + S_{k-1}(2,0) + S_{k-1}(4,0) + S_{k-1}(5,0) \tag{6.21}$$

Acestea sunt formulele de recurență. Rezultatul care trebuie afișat pe ecran este $2[S_{N-1}(5,1) + S_{N-1}(7,1)]$, deoarece după $N-1$ coloane surplusul trebuie să fie 1, iar colțurile matricei trebuie să fie bifate. Se observă că $S_k(1,0) = S_k(4,0) = S_k(5,1)$. Practic, problema se

reduce la trei șiruri. Notăm:

$$\begin{aligned} A_k &= S_k(5,0) \\ B_k &= S_k(1,0) = S_k(4,0) = S_k(5,1) \\ C_k &= S_k(7,1) \implies S_k(2,0) = C_{k-1} \end{aligned} \tag{6.22}$$

De aici rezultă grupul de relații:

$$\begin{cases} A_k &= A_{k-1} + 2B_{k-1} \\ B_k &= B_{k-1} + C_{k-1} \\ C_k &= A_{k-1} + 2B_{k-1} + C_{k-2} = A_k + C_{k-2} \end{cases} \tag{6.23}$$

și

$$\begin{cases} A_1 &= 1 \\ B_1 &= 0 \\ C_1 &= 1 \end{cases} \tag{6.24}$$

Noi avem nevoie de valoarea

$$2(B_{N-1} + C_{N-1}) = 2B_N \tag{6.25}$$

Programul de mai jos nu face decât să implementeze calculul acestor șiruri recurente. Trebuie avut grijă însă cu reprezentarea internă a numerelor, deoarece pentru $N = 200$ valorile ajung la 81 de cifre. Este deci necesară reprezentarea numerelor ca șiruri de cifre.

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

typedef int Huge[85];
Huge A,B,C,C2,HTemp;
int N,k;

void Atrib(Huge H, int V)
/* H <- V */
{
  memset(H,0,sizeof(Huge));
  H[0]=1;
```

```
13    H[1]=V;
14  }
15
16  void Add(Huge A, Huge B)
17  /* A <- A+B */
18  { int i,T=0;
19
20    if (B[0]>A[0])
21      { for (i=A[0]+1;i<=B[0];) A[i++]=0;
22        A[0]=B[0];
23      }
24      else for (i=B[0]+1;i<=A[0];) B[i++]=0;
25    for (i=1;i<=A[0];i++)
26      { A[i]+=B[i]+T;
27        T=A[i]/10;
28        A[i]%=10;
29      }
30    if (T) A[++A[0]]=T;
31  }
32
33  void WriteHuge(Huge H)
34  { int i;
35
36    for (i=H[0];i;printf("%d",H[i--]));
37    printf("\n");
38  }
39
40  void main(void)
41  {
42    printf("N=");scanf("%d",&N);
43    Atrib(A,1);
44    Atrib(B,0);
45    Atrib(C,1);
46    Atrib(C2,0);
47    for (k=2;k<=N;k++)
48      { memmove(HTemp,C,sizeof(Huge));
49        Add(A,B);Add(A,B);   /* A(k) = A(k-1) + 2*B(k-1) */
50        Add(B,C);            /* B(k) = B(k-1) + C(k-1)   */
51        memmove(C,A,sizeof(Huge));
52        Add(C,C2);           /* C(k) = A(k) + C(k-2)     */
53        memmove(C2,HTemp,sizeof(Huge)); /* noul C(K-2) */
54      }
55    Add(B,B);              /* Rezultatul este 2*B(n)   */
56    WriteHuge(B);
57  }
```

## 6.7 Problema 7

Problema programării unui turneu de fotbal a fost dată spre rezolvare la a III-a Balcaniadă de Informatică, Varna 1995. Vom prezenta mai întâi enunțul nemodificat al problemei, după care vom adăuga câteva detalii care o vor face mai „provocatoare".

**ENUNȚ**: Una din sarcinile Ministerului Sporturilor dintr-o țară balcanică este de a organiza un campionat de fotbal cu $N$ echipe (numerotate de la 1 la $N$). Campionatul constă din $N$ etape (dacă $N$ este impar) sau $N-1$ etape (dacă $N$ este par); orice două echipe dispută între ele un joc și numai unul. Scrieți un program care realizează o programare a campionatului.

**Intrarea**: Numărul de echipe $N$ ($2 \leq N \leq 24$) va fi dat la intrarea standard (tastatură).

**Ieșirea** se va face în fișierul text `OUTPUT.TXT` sub forma unui tabel cu numere întregi având $N$ linii. Al $j$-lea element din linia $i$ este numărul echipei care joacă cu echipa $i$ în etapa $j$ (evident, dacă $i$ joacă cu $k$ în etapa $j$, atunci în tabel $k$ joacă cu $i$ în aceeași etapă). Dacă echipa $i$ este liberă în etapa $j$, atunci al $j$-lea element al liniei $i$ este zero.

**Exemple**:

| **INPUT.TXT** | **OUTPUT.TXT** |
|---|---|
| 3 | 2 3 0<br>1 0 3<br>0 1 2 |
| 4 | 2 3 4<br>1 4 3<br>4 1 2<br>3 2 1 |

**Timp de implementare**: circa 1h 45'

**Timp de execuție**: 33 secunde

Iată și modificările pe care le propunem pentru a aduce cu adevărat problema la nivel de concurs:

- Limita pentru numărul de echipe este $N \leq 200$;
- **Timpul de implementare** este de 45 minute;
- **Timpul de execuție** este de 2-3 secunde;
- **Complexitatea cerută** este $O(N^2)$.

**REZOLVARE**: Vom lăsa pe seama cititorului conceperea, implementarea și testarea unei rezolvări backtracking (adoptată de majoritatea în timpul concursului). De altfel, la Balcaniadă concurenții au avut la dispoziție calculatoare 486 / 50 Mhz, iar un backtracking îngrijit funcționa cam până la $N = 18$ în timpul impus, ceea ce asigura cam 75% din punctajul maxim. În afară de aceasta, se mai puteau face și alte lucruri nu tocmai elegante, având în vedere că datele de ieșire nu erau foarte mari. Pentru $N > 18$, mulți concurenți lăsau programul să meargă până când termina (câteva minute bune sau chiar mai mult), apoi scriau rezultatele într-un fișier temporar. Matricea rezultată era inclusă ca o constantă în codul sursă. Programul care era dat comisiei de corectare se prefăcea că „se gândește" timp de câteva secunde, apoi scria pur și simplu matricea în fișierul de ieșire. Cu aceasta s-au luat punctaje foarte apropiate de maxim. Totuși, pentru $N = 24$ un backtracking ar fi stat foarte mult pentru a găsi soluția. Tot timpul acesta, calculatorul era blocat, neputând fi folosit. De aceea, mulți concurenți nu au luat punctajul maxim, preferând să renunțe la ultimele teste și să rezolve celelalte probleme. Oricum, backtracking-ul este în acest caz o soluție pentru care raportul punctaj obținut / timp consumat este foarte convenabil.

Există însă și o soluție care poate asigura un punctaj maxim fără bătăi de cap și fără să folosească „date preprocesate". Ea are complexitatea $O(N^2)$ și nu face decât o singură parcurgere a matricei de ieșire. Ce-i drept, autorul a pierdut cam trei ore pentru a o găsi și implementa în timp de concurs, compromițând aproape rezolvarea celorlalte două probleme din ziua respectivă, dar comisia de corectare a fost plăcut impresionată, programul fiind singurul care mergea instantaneu. Rămâne ca voi să alegeți între eleganță și eficiență...

Dacă ținem cont și de restricțiile suplimentare propuse, rezolvarea backtracking nu mai este valabilă. În această situație, iată care este metoda care stă la baza rezolvării în timp pătratic. În primul rând se reduce cazul când $N$ este impar la un caz când $N$ este par, prin mărirea cu 1 a lui $N$ și introducerea unei echipe fictive. În fiecare etapă se consideră că echipa care trebuia să joace cu echipa fictivă stă de fapt „pe bară". Iată de exemplu cum se rezolvă cazul $N = 3$:

(a) Se programează un campionat cu 4 echipe, echipa 4 fiind echipa fictivă:

| **Echipa** | **Etapa 1** | **Etapa 2** | **Etapa 3** |
|---|---|---|---|
| **1** | 2 | 3 | 4 |
| **2** | 1 | 4 | 3 |
| **3** | 4 | 1 | 2 |
| **4** | 3 | 2 | 1 |

(b) Se neglijează ultima linie din tabel, meciurile echipei fictive nefiind importante:

| **Echipa** | **Etapa 1** | **Etapa 2** | **Etapa 3** |
|---|---|---|---|
| **1** | 2 | 3 | 4 |
| **2** | 1 | 4 | 3 |
| **3** | 4 | 1 | 2 |

(c) Peste tot unde apare cifra 4, ea este înlocuită cu 0:

| **Echipa** | **Etapa 1** | **Etapa 2** | **Etapa 3** |
|---|---|---|---|
| **1** | 2 | 3 | 0 |
| **2** | 1 | 0 | 3 |
| **3** | 0 | 1 | 2 |

Mai rămâne să vedem cum se tratează cazul când $N$ este par. E destul de greu de dat o demonstrație matematică metodei care urmează; de fapt, nici nu există una, problema fiind rezolvată în timp de concurs prin inducție incompletă (adică s-a constatat cu creionul pe hârtie că metoda merge pentru $N = 4, 6, 8$ și 10, apoi s-a scris programul care să facă același lucru și s-a constatat că merge și pentru valori mai mari). Să tratăm și aici cazul $N = 8$, apoi să generalizăm procedeul.

Vor fi șapte etape și, fără a reduce cu nimic generalitatea problemei, putem presupune că echipa 1 joacă pe rând cu echipele 2, 3, 4, 5, 6, 7 și 8. Deocamdată tabelul arată astfel:

| **Echipa** | **Etapa 1** | **Etapa 2** | **Etapa 3** | **Etapa 4** | **Etapa 5** | **Etapa 6** | **Etapa 7** |
|---|---|---|---|---|---|---|---|
| **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **2** | 1 | | | | | | |
| **3** | | 1 | | | | | |
| **4** | | | 1 | | | | |
| **5** | | | | 1 | | | |
| **6** | | | | | 1 | | |
| **7** | | | | | | 1 | |
| **8** | | | | | | | 1 |

Echipa 2 are deja programat un meci cu echipa 1 în prima etapă și mai are de programat meciurile cu echipele 3, 4, 5, 6, 7 și 8 în celelalte etape. Așezarea se poate face oricum, cu condiția ca echipele 1 și 2 să nu își aleagă același partener în aceeași etapă. De exemplu, putem programa meciul 2-3 în etapa a 3-a, meciul 2-4 în etapa a 4-a, ..., iar meciul 2-8 în etapa a 2-a. Astfel am completat linia a doua a tabloului:

| Echipa | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 |
|---|---|---|---|---|---|---|---|
| **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **2** | 1 | 8 | 3 | 4 | 5 | 6 | 7 |
| **3** | | 1 | 2 | | | | |
| **4** | | | 1 | 2 | | | |
| **5** | | | | 1 | 2 | | |
| **6** | | | | | 1 | 2 | |
| **7** | | | | | | 1 | 2 |
| **8** | | 2 | | | | | 1 |

Echipa 3 are programate meciurile cu 1 și 2 și mai are de programat meciurile cu echipele 4, 5, 6, 7 și 8. Pentru a nu crea conflicte (adică două echipe să nu-și aleagă același adversar), putem aplica același procedeu: pornim de la etapa a 5-a și completăm toate celulele goale ale liniei a 3-a cu numerele de la 4 la 8, mergând circular spre dreapta:

| Echipa | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 |
|---|---|---|---|---|---|---|---|
| **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **2** | 1 | 8 | 3 | 4 | 5 | 6 | 7 |
| **3** | 7 | 1 | 2 | 8 | 4 | 5 | 6 |
| **4** | | | 1 | 2 | 3 | | |
| **5** | | | | 1 | 2 | 3 | |
| **6** | | | | | 1 | 2 | 3 |
| **7** | 3 | | | | | 1 | 2 |
| **8** | | 2 | | 3 | | | 1 |

Se folosește aceeași metodă pentru a completa și celelalte linii ale matricei. Pentru fiecare linie $i$ ($i \leq N-1$):

- Se caută pe linia $i$ apariția valorii $i$-1;
- Se caută primul spațiu liber (mergând circular spre dreapta). Primul spațiu liber înseamnă prima etapă în care se poate programa meciul dintre echipele $i$ și $i+1$ (trebuie ca ambele să fie libere în acea etapă). Se constată experimental că trebuie început de la a doua poziție liberă de după apariția valorii $i-1$;
- Se merge circular spre dreapta și, în căsuțele libere întâlnite se trec valorile $i+1, i+2, \ldots, N$. Concomitent, pe aceleași coloane ale liniilor $i+1, i+2, \ldots, N$ se trece valoarea $i$.

| Echipa | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | Etapa 5 | Etapa 6 | Etapa 7 |
|---|---|---|---|---|---|---|---|
| **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **2** | 1 | 8 | 3 | 4 | 5 | 6 | 7 |
| **3** | 7 | 1 | 2 | 8 | 4 | 5 | 6 |
| **4** | 6 | 7 | 1 | 2 | 3 | 8 | 5 |
| **5** | 8 | 6 | 7 | 1 | 2 | 3 | 4 |
| **6** | 4 | 5 | 8 | 7 | 1 | 2 | 3 |
| **7** | 3 | 4 | 5 | 6 | 8 | 1 | 2 |
| **8** | 5 | 2 | 6 | 3 | 7 | 4 | 1 |

Fiecare linie a matricei poate fi parcursă in acest fel de maxim 2 ori (o dată pentru găsirea coloanei de start și o dată pentru completarea liniei), deci numărul total de celule vizitate este cel mult $2 \times N^2$. Complexitatea pătratică este cea optimă, deoarece programul trebuie în orice caz să tipărească la ieșire circa $N^2$ numere.

```
#include <stdio.h>
#define NMax 200
unsigned char A[NMax+1][NMax+1];
int N,RealN;

void FindNextFree(int Line,int *K)
/* Cauta (circular) urmatorul spatiu liber pe linia Line */
{ do *K=*K%(N-1)+1;
  while (A[Line][*K]);
}

void MakeMatrix(void)
{ int i,j,k;

  for (i=1;i<=N;i++)
    for (j=1;j<=N;j++) A[i][j]=0;
  for (i=1;i<N;i++)  /* Pentru fiecare echipa */
    { if (i==1)      /* Alege coloana de start */
        j=1;
        else { j=0;
               do j++; while (A[i][j]!=i-1);
               FindNextFree(i,&j);
               FindNextFree(i,&j);
             }
      for (k=i+1;k<=N;k++) /* Completeaza circular linia */
        { A[i][j]=k;
```

```
27            A[k][j]=i;
28            if (k<N) FindNextFree(i,&j);
29          }
30      }
31  }
32
33  void WriteMatrix(void)
34  { int i,j;
35    FILE *F=fopen("output.txt","wt");
36
37    for (i=1;i<=RealN;i++)
38      { for (j=1;j<N;j++)
39          fprintf(F,"%4d",A[i][j]>RealN ? 0 : A[i][j]);
40        fprintf(F,"\n");
41      }
42    fclose(F);
43  }
44
45  void main(void)
46  {
47    printf("N=");scanf("%d",&RealN);
48    N=RealN+RealN%2; /* Se adauga 1 daca RealN e impar */
49    MakeMatrix();
50    WriteMatrix();
51  }
```

## 6.8 Problema 8

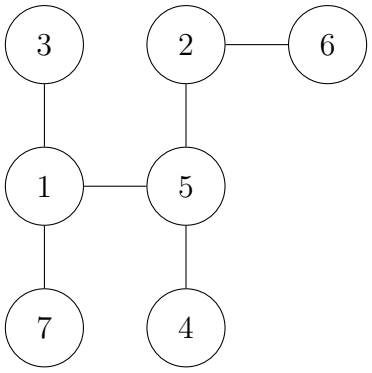
Problema **codului lui Prüfer** pentru arborii generali, mai exact cea a decodificării acestui cod, este genul de problemă care nu este excesiv de grea, dar care necesită un artificiu fără de care nu se poate ajunge la complexitatea optimă.

**ENUNȚ**: Un arbore general neorientat conex cu $N + 2$ vârfuri se poate codifica eficient printr-un vector cu $N$ numere, astfel:

- Numerotăm nodurile de la 1 la $N + 2$ într-o ordine oarecare;
- Eliminăm cea mai mică frunză (nod de grad 1) și adăugăm în vector numărul nodului de care ea aparținea;
- Reluăm procedeul pentru arborele rămas: tăiem cea mai mică frunză și adăugăm în vector numărul nodului de care ea aparținea;

- Repetăm procedeul până mai rămân doar două noduri.

Vectorul rezultat se numește codificare Prüfer a arborelui dat. Iată un exemplu de construcție a codului Prüfer atașat arborelui din figura de mai jos:




Codul lui Prüfer se obține astfel:

- Îl tăiem pe 3 și scriem 1;
- Îl tăiem pe 4 și scriem 5;
- Îl tăiem pe 6 și scriem 2;
- Îl tăiem pe 2 și scriem 5;
- Îl tăiem pe 5 și scriem 1;

Deci codificarea este (1, 5, 2, 5, 1) (și mai rămâne muchia 1-7, lucru care este evident, deoarece 1 și 7 sunt singurele noduri care nu au fost tăiate).

Așadar fiecărui arbore oarecare cu $N+2$ noduri i se poate atașa un vector cu $N$ componente numere naturale cuprinse între 1 și $N+2$. Se poate demonstra că funcția definită între cele două mulțimi (mulțimea arborilor și mulțimea vectorilor) este bijectivă. De aici rezultă două lucruri:

1. Există $(N+2)^N$ arbori generali cu $N+2$ noduri.
2. Codificarea Prüfer admite și decodificare (deoarece funcția de codificare este bijectivă). Tocmai aceasta este problema de rezolvat. Se dă un vector de $N$ numere întregi, fiecare cuprins între 1 și $N+2$. Se cere să se tipărească cele $N+1$ muchii ale arborelui decodificat.

**Intrarea** se face din fișierul text `INPUT.TXT` care conține două linii. Pe prima linie se dă $N$ ($1 \leq N \leq 10.000$), pe a doua cele $N$ numere separate prin spații.

**Ieșirea** se va face în fișierul text `OUTPUT.TXT`. Acesta va conține muchiile arborelui, câte una pe linie, o muchie fiind indicată prin vârfurile adiacente separate printr-un spațiu.

**Exemplu**:

| **INPUT.TXT** | **OUTPUT.TXT** |
|---|---|
| 5 | 5 2 |
| 1 5 2 5 1 | 1 3 |
| | 4 5 |
| | 7 1 |
| | 6 2 |
| | 1 5 |

**Timp de implementare**: 45 minute.

**Timp de rulare**: 2-3 secunde.

**Complexitate cerută**: $O(N)$.

**REZOLVARE**: Să pornim de la exemplul particular prezentat mai sus, urmând ca apoi să generalizăm algoritmul.

Primim la intrare $N = 5$ (deducem că arborele are 7 noduri) și codificarea (1, 5, 2, 5, 1). Primul element din vector este 1. Știm deci că, la primul pas, a fost eliminată o frunză al cărei părinte este nodul 1. Întrebarea este: ce număr purta respectiva frunză? În nici un caz 1, deoarece numărul 1 îl avea tatăl ei. Nici 2, nici 5 nu ar putea fi, deoarece aceste numere apar mai târziu în codificare, deci „mai este nevoie" de ele și nu pot fi încă eliminate. Rămân 3, 4, 6 și 7. Dintre acestea noi știm că a fost tăiată cea mai mică **frunză**. Este însă ușor de văzut că toate nodurile enumerate sunt frunze în arborele inițial (deoarece nu mai apar în vector, adică nici un alt nod nu mai este legat de ele), deci îl vom alege pe cel mai mic cu putință, adică pe 3. Rezultă că prima muchie tăiată a fost (1,3).

Pe poziția a doua în codificare apare numărul 5. Ce număr ar putea avea frunza tăiată? 1 și 2 mai apar ulterior în codificare deci nu pot fi eliminate încă, 5 este chiar tatăl frunzei necunoscute, iar 3 a fost deja eliminat. Dintre 4, 6 și 7, frunza cu numărul cel mai mic este 4, deci următoarea muchie tăiată este (4,5).

În continuare apare un 2. Cel mai mic nod care nu mai apare în vector și nici nu a fost deja eliminat este 6, deci muchia este (6,2). Se observă că mai departe nu mai apare nici un 2 în codificare, de unde deducem că după tăierea nodului 6, nodul 2 a devenit frunză și va putea fi la rândul său eliminat. Cu un raționament analog, următoarele muchii eliminate sunt (2,5) și (5,1), iar singurele noduri rămase sunt 1 și 7. Arborele a fost reconstituit corect.

Deci procedeul general este: pentru fiecare număr dintre cele $N$ din vector, nodul care aparținea de el poartă cel mai mic număr care nu intervine ulterior în codificare și nu a fost

tăiat deja. Astfel se generează $N$ muchii. A $N+1$-a muchie are drept capete ultimele noduri rămase netăiate.

Acesta este algoritmul. Trebuie să ne ocupăm acum de partea de implementare.

Pentru a testa la orice moment dacă un nod mai apare sau nu în vector, este bine să se creeze la citirea datelor un vector care să rețină numărul de apariții în codificare al fiecărui nod. Pentru exemplul dat, vectorul de apariții va fi $Apar = (2, 1, 0, 0, 2, 0, 0)$, semnificând că 1 și 5 apar de câte două ori, 2 apare o singură dată, iar 3, 4, 6 și 7 nu apar deloc. La fiecare pas, când se „restaurează" o muchie, se decrementează poziția corespunzătoare tatălui în vectorul de apariții. Un număr nu mai apare ulterior în codificare dacă pe poziția sa din vectorul de apariții se află un 0.

Astfel, o primă versiune de program ar putea fi:

**Intrare:** $N$, $V[1..N]$
1: construiește vectorul $Apar[1..N+2]$
2: **pentru** $i = 1$ la $N$ **execută**
3: caută primul $j$ pentru care $Apar[j] = 0$ și $j$ nu a fost tăiat
4: **tipărește** muchia $(j, V[i])$
5: $Apar[V[i]] \leftarrow Apar[V[i]] - 1$
6: marchează nodul $j$ ca fiind tăiat
7: **sfârșit pentru**

După cum se observă, căutarea celui mai mic nod $j$ se face în timp liniar, ceea ce înseamnă că algoritmul complet va necesita un timp pătratic. Pentru a-l reduce la o complexitate liniară, începem prin a observa că la fiecare pas alegem frunza cu numărul cel mai mic. Aceasta înseamnă că, în general, numărul frunzei alese spre a fi tăiată va crește la fiecare pas. Astfel, prima oară am tăiat nodul 4, apoi nodul 7. Singurul caz când va fi tăiată o frunză mai mică decât cea dinaintea ei este atunci când unui nod cu număr mic i se taie toate frunzele și devine el însuși o frunză, putând fi tăiat. În exemplul nostru, nodul 2 nu putea fi eliminat de la început, deși avea un număr mic, deoarece mai avea atașată frunza 6. După eliminarea muchiei (6,2), nodul 2 a devenit frunză și a fost eliminat imediat.

Putem deci păstra într-o variabilă (care în program se numește $Next$) următoarea frunză care care trebuie eliminată. Dacă prin decrementarea numărului de apariții la pasul curent nu s-a creat nici un zero în vectorul $Apar$, sau s-a creat un zero, dar pe o poziție $K > Next$, atunci totul este bine și la pasul următor se va elimina nodul $Next$. Dacă s-a creat un zero pe o poziție $K$ mai mică decât $Next$, rezultă că în arbore există acum două frunze, $K$ și $Next$, $K$ fiind mai mică, deci prioritară. Atunci la pasul următor se va elimina frunza de pe poziția $K$, urmând ca peste doi pași să se revină la frunza $Next$. Dacă prin eliminarea acestei frunze $K$ s-a creat un alt zero în vectorul de apariții, tot pe o poziție $K'$ mai mică decât Next, se va trece mai întâi

la acea poziție, urmând ca după aceea să se revină la poziția $Next$ etc.

La prima vedere, pare necesară menținerea unei stive în care să depunem numerele $Next, K, K'...$ și să le scoatem din stivă pe măsură ce nodurile respective sunt eliminate. Acest lucru ar presupune în continuare un algoritm pătratic. Totuși nu este așa deoarece, odată ce am tăiat un nod și l-am marcat ca atare, putem fi siguri că nu ne vom mai intâlni cu el până la sfârșitul decodificării, deci nu mai este necesară stocarea lui. Există o pseudo-stivă care are înălțimea 2: frunza asupra căreia se operează curent și nodul care urmează, $Next$.

În acest fel, numărul total de incrementări al variabilei $Next$ nu depășește $N$ (iar ea nu este niciodată decrementată), deci programul este rezolvat în timp liniar. Facem observația că algoritmul $O(N)$ este optim; nu poate exista unul mai bun deoarece există $O(N)$ muchii care trebuie tipărite.

```
#include <stdio.h>
#define NMax 10002
typedef int Vector[NMax];

Vector V,Apar;
int N;

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

  fscanf(InF,"%d",&N);
  for (i=1;i<=N+2;Apar[i++]=0);
  for (i=1;i<=N;i++)
    { fscanf(InF,"%d",&V[i]);
      Apar[V[i]]++;
    }
  fclose(InF);
}

void Decode(void)
{ int Current=0,Next,i;
  FILE *OutF=fopen("output.txt","wt");

  do; while (Apar[++Current]);     /* Se cauta prima frunza */
  Next=Current;
  for (i=1;i<=N;i++)
    { fprintf(OutF,"%d %d\n",Current,V[i]);
      if (Current==Next) do; while (Apar[++Next]);
      /* Daca am ajuns la ultimul 0, mai caut unul */
```

```
31        Apar[V[i]]--;
32        Current=(V[i]<Next) && (Apar[V[i]]==0) ? V[i] : Next;
33        /* Daca exista o frunza mai mica decat Next, */
34        /* ea este prioritara, altfel revin la Next */
35      }
36    fprintf(OutF,"%d %d\n",Current,Next);
37    fclose(OutF);
38  }
39
40  void main(void)
41  {
42    ReadData();
43    Decode();
44  }
```

## 6.9 Problema 9

Iată o problemă care necesită pentru reducerea complexității un artificiu asemănător celui din problema codului Prüfer. Ea a fost propusă în 1995 la concursul de selecție a echipelor României pentru IOI și CEOI. Deși cerința este puțin modificată pentru a nu ne izbi de dificultăți secundare, ideea generală de abordare este aceeași.

**ENUNȚ**: Președintele companiei X dorește să organizeze o petrecere cu angajații. Această companie are o structură ierarhică în formă de arbore. Fiecare angajat are asociat un număr întreg reprezentând măsura sociabilității sale. Pentru ca petrecerea să fie agreabilă pentru toți participanții, președintele dorește să facă invitațiile astfel încât:

- el însuși să participe la petrecere;
- pentru nici un participant la petrecere să nu fie invitat și șeful lui direct;
- suma măsurilor sociabilităților invitaților să fie maximă.

Se cere să se spună care este suma maximă a sociabilităților care se poate obține.

**Intrarea** se face din fișierul `INPUT.TXT` care conține trei linii de forma:

```
N
T(1) T(2) ... T(N)
S(1) S(2) ... S(N)
```

unde $N$ este numărul de angajați ai companiei, inclusiv președintele ($N \leq 1.000$), $T(k)$ este numărul de ordine al șefului direct al lui $k$ (dacă $T(k) = 0$, atunci $k$ este președintele), iar $S(k)$ este măsura sociabilității lui $k$. Valorile vectorului $S$ sunt de tipul întreg.

**Ieșirea**: Pe ecran se va tipări suma maximă a sociabilităților ce se poate obține.

**Exemplu**: Pentru intrarea:

```
7
2 5 2 5 0 4 2
2 5 3 13 8 4 3
```

pe ecran se va afișa numărul 20 (fiindcă la petrecere participă 1, 3, 5, 6 și 7).

**Timp de implementare**: 1h - 1h 15min.

**Timp de rulare**: 2-3 secunde.

**Complexitate cerută**: $O(N)$. De asemenea, menționăm că s-a specificat $N \leq 1.000$ numai pentru ca programul sursă de mai jos să nu se complice și să distragă atenția asupra unor lucruri neimportante. În mod normal, limita trebuia să fie $N \leq 10.000$.

**REZOLVARE**: Vom expune mai întâi principiul de rezolvare al problemei, apoi vom aborda detaliile de implementare.

Algoritmul are la bază programarea dinamică și necesită o parcurgere de jos în sus a arborelui, decizia pentru fiecare nod depinzând de valorile tuturor fiilor lui. Ne propunem să aflăm două caracteristici pentru fiecare nod $k$:

- $P(k)$ - suma maximă a sociabilităților care se poate obține în subarborele de rădăcină $k$ în cazul în care $k$ participă la petrecere;
- $Q(k)$ - suma maximă a sociabilităților care se poate obține în subarborele de rădăcină $k$ în cazul în care $k$ nu participă la petrecere;

Dacă reușim să determinăm aceste caracteristici, nu ne rămâne decât să-l tipărim pe $P(R)$ ($R$ fiind rădăcina arborelui). Într-adevăr, problema cere să se determine suma maximă a sociabilităților din întregul arbore, adică din subarborele de rădăcină $R$. În plus, se mai cere ca $R$ să participe la petrecere. Rămâne de aflat cum se stabilește relația între caracteristicile unui nod și cele ale fiilor săi.

- Dacă angajatul $k$ participă la petrecere, atunci automat nici unul din subordonații săi

direcți nu participă, și obținem relația:

$$P(k) = S(k) + \sum_{j} Q(j), \quad j \text{ fiu al lui } k \tag{6.26}$$

- Dacă angajatul $k$ nu participă la petrecere, atunci subordonații săi direcți pot să participe sau nu la petrecere, după cum este mai avantajos, și obținem relația:

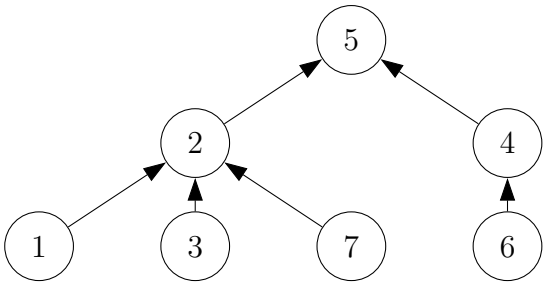
$$Q(k) = \sum_{j} \max(P(j), Q(j)), \quad j \text{ fiu al lui } k \tag{6.27}$$

Problema care se pune acum este cum să facem parcurgerea arborelui într-un mod cât mai avantajos. Pseudocodul (recursiv) sub forma sa cea mai generală este:

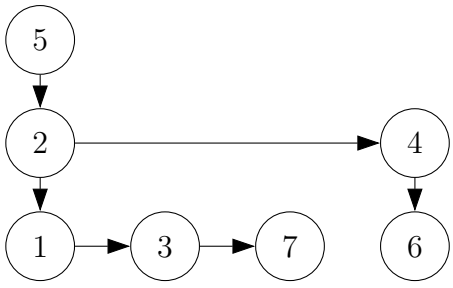
**Procedura 2** Calcul($k$, $P$, $Q$)

1: $P \leftarrow S(k)$
2: $Q \leftarrow 0$
3: **pentru toți** $j$ fiu al lui $k$ **execută**
4: Calcul($j$, $P_1$, $Q_1$)
5: $P \leftarrow P + Q_1$
6: $Q \leftarrow Q + \max(P_1, Q_1)$
7: **sfârșit pentru**
8: **returnează** $P$, $Q$

Soluția „de bun simț” este de a construi arborele alocat dinamic. Ar apărea însă o sumedenie de dificultăți. În primul rând, arborele este general, deci nu se cunoaște numărul maxim de fii pe care îi poate avea un nod (pe cazul cel mai defavorabil, rădăcina poate avea $N - 1$ fii). Aceasta înseamnă că legătura trebuie să fie de tip tata (fiecare nod pointează la tatăl său), ceea ce complică procedura de parcurgere: nu se poate apela procedura recursiv, dintr-un nod pentru toți fiii săi, deoarece nu se cunosc fiii, ci numai tatăl! O altă modalitate de alocare a arborelui ar fi cu doi pointeri pentru fiecare nod: unul către primul său fiu și unul către fratele său din dreapta (exemplul din enunț are reprezentate grafic mai jos cele două metode de construcție).




Legătură de tip tată




Legătură de tip fiu + frate

Probabil că veți fi de acord cu mine că e riscant să te aventurezi la o asemenea implementare în timp de concurs, deoarece lucrul cu pointeri presupune o atenție deosebită. Greșelile sunt mai greu de observat și de multe ori duc la blocarea calculatorului, care trebuie resetat mereu, pierzându-se astfel o mulțime de timp. Trebuie deci căutată o metodă de parcurgere a arborelui care să nu necesite o alocare dinamică a memoriei. Putem încerca astfel: inițial $P(i) = S(i)$ și $Q(i) = 0$ pentru orice nod. Urmează acum să tratăm pe rând fiecare nod. Cum? Știm că pentru fiecare nod $k$, numerele $P(k)$ și $Q(k)$ intervin în expresia lui $P(T(k))$ și $Q(T(k))$. Uitându-ne la formulele de mai sus, observăm că tot ce avem de făcut este să incrementăm $P(T(k))$ cu $Q(k)$ și $Q(T(k))$ cu $\max(P(k), Q(k))$.

Există o singură problemă: Pentru a putea folosi numerele $P(k)$ și $Q(k)$ trebuie să ne asigurăm că ele au fost deja calculate corect, în funcție de caracteristicile tuturor fiilor lui $k$. În cazul frunzelor, problema este rezolvată, deoarece ele nu au fii. Pentru nodurile interne, este necesar să știm câți fii au ele și câți din aceștia au fost tratați. În momentul în care toți fiii unui nod au fost tratați, poate fi tratat și nodul în sine. În program, vectorul $F$ reține numărul de fiii netratați ai fiecărui nod. La tratarea unui nod $k$ se face decrementarea lui $F(T(k))$. Un nod $k$ poate fi ales spre tratare dacă $F(k) = 0$. Se observă că formatul datelor de intrare permite cu ușurință construcția vectorului $F$ (numărul de fii al lui $k$ este egal cu numărul de apariții al lui $k$ în vectorul $T$).

Un algoritm mai ușor de implementat ar fi:

1: numără fiii fiecărui nod
2: **pentru** $i = 1$ la $N - 1$ **execută**
3: caută un nod $k$ cu $F(k) = 0$
4: $P(T(k)) \leftarrow P(T(k)) + Q(k)$
5: $Q(T(K)) \leftarrow Q(T(K)) + \max(P(k), Q(k))$
6: $F(T(K)) \leftarrow F(T(K)) - 1$
7: **sfârșit pentru**
8: **tipărește** $P(R)$

Partea delicată a acestui algoritm este căutarea unui nod $k$ cu $F(k) = 0$. Dacă ea se face secvențial, pornind de fiecare dată de la primul nod și cercetând fiecare element, programul care rezultă are complexitatea $O(N^2)$. Această căutare trebuie deci optimizată, și iată cum: În principiu, putem căuta nodurile cu $F(k) = 0$ mergând numai în sensul crescător al indicilor în vector. Vom folosi, ca și la problema codului lui Prüfer, două variabile: $K$ și $Next$. $K$ este nodul tratat în prezent, iar $Next$ este nodul care urmează a fi tratat la pasul următor, așadar $Next > K$. După tratarea nodului $K$ și decrementarea lui $F(T(k))$, pot surveni trei situații:

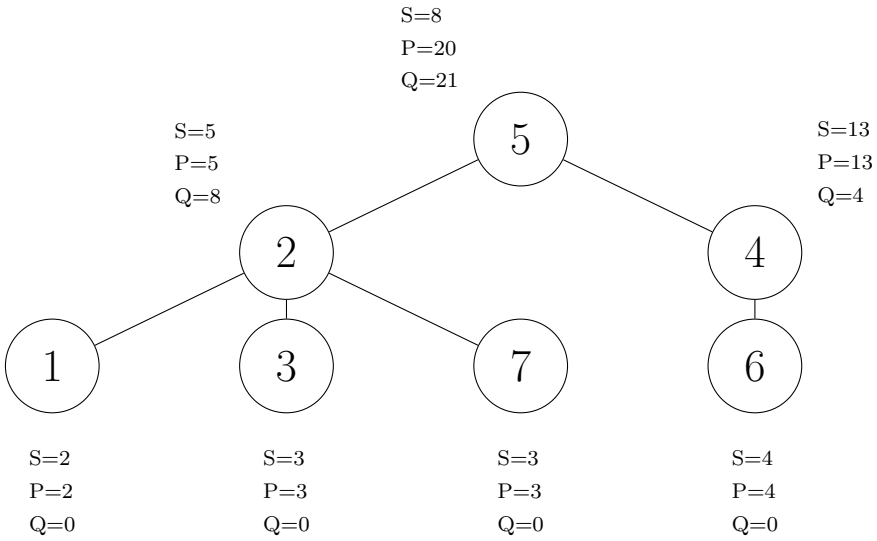
1. $F(T(k)) > 0$ și în vectorul $F$ nu a apărut nici un alt element zero, caz în care nimic nu se schimbă, algoritmul continuând cu tratarea nodului $Next$;

2. $F(T(k)) = 0$ și $T(k) > Next$, caz în care de asemenea se poate trata nodul $Next$ (deoarece selectarea nodurilor cu $F(k) = 0$ se face în ordine crescătoare a indicilor);

3. $F(T(k)) = 0$ și $T(k) < Next$, caz în care se va trata mai întâi nodul $T(k)$, urmând a se reveni apoi la nodul $Next$.

Pentru motivele explicate la codul lui Prüfer, acest algoritm nu necesită menținerea unei stive, iar timpul total de căutare este $O(N)$. Facem observația că nu se poate găsi un algoritm mai bun, deoarece trebuie parcurse toate cele $N$ noduri ale arborelui.

Iată în încheiere cum arată arborele cu valorile $P$ și $Q$ atașate fiecărui nod:




```c
#include <stdio.h>
#define NMax 1000
typedef long Vector[NMax+1];

Vector T,S,P,Q,F; /* T = Vectorul de tati,
                     S = sociabilitatile,
                     P,Q = caracteristicile,
                     F = numarul de fii */
int N,Root;

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

  fscanf(InF,"%d",&N);
  for (i=1;i<=N+1;F[i++]=0);
```

```
17    for (i=1;i<=N;i++)
18      { fscanf(InF,"%d",&T[i]);
19        if (T[i]) F[T[i]]++; else Root=i;
20      }
21    for (i=1;i<=N;i++) fscanf(InF,"%d",&S[i]);
22    fclose(InF);
23
24    for (i=1;i<=N;P[i++]=S[i]);
25    for (i=1;i<=N;Q[i++]=0);
26  }
27
28  void FindNext(int *K,int *Next)
29  { F[T[*K]]--;
30    F[*K]=-1;
31    if ((F[T[*K]]>0) || (T[*K]>*Next))
32      { *K=*Next;
33        while (F[*Next]) (*Next)++;
34      }
35      else *K=T[*K];
36  }
37
38  void TraverseTree(void)
39  { int K=0,Next,i;
40
41    do; while (F[++K]);
42    Next=K; do; while (F[++Next]);
43    for (i=1;i<N;i++)
44      { P[T[K]]+=Q[K];
45        Q[T[K]]+=(P[K]>Q[K]) ? P[K] : Q[K];
46        FindNext(&K,&Next);
47      }
48  }
49
50  void main(void)
51  {
52    InitData();
53    TraverseTree();
54    printf("%ld\n",P[Root]);
55  }
```

## 6.10 Problema 10

Această problemă a fost dată la unul din barajele pentru selecționarea lotului restrâns al României pentru Olimpiada Internațională din 1996.

**ENUNȚ**: Fie un număr prim $P$. Pe mulțimea $\{0, 1, \ldots, P-1\}$ se definesc operațiile binare $+$, $-$, $\times$, / modulo $P$, în felul următor:

1. $a + b \bmod P$ este restul împărțirii lui $a + b$ la $P$. Analog pentru „$\times$”.
2. Expresia $a - b$ este definită ca fiind soluția ecuației $b + x \equiv a \pmod{P}$. Analog pentru „/”.

Se știe că ecuația $b + x = a$ are întotdeauna soluție unică, iar $b \times x = a$ are soluție unică pentru orice $b \neq 0$. Pentru $b = 0$, operația $a/b$ nu e definită. De exemplu, dacă $P = 11$, atunci $6 + 7 = 2$, $6 - 7 = 10$, $6 \times 7 = 9$, $6/7 = 4$. Se știe că adunarea și înmulțirea modulo $P$ sunt comutative, asociative, posedă elemente neutre (pe 0, respectiv pe 1), iar adunarea este distributivă față de înmulțire. În plus, pentru orice $a$ există $b$ astfel încât $a + b = 0$; notând $b$ cu $(-a)$ avem $c - a = c + (-a) = c + b$ pentru orice $c$. De asemenea, pentru orice $a \neq 0$ există $b$ astfel încât $a \times b = 1$; notând $b$ cu $(1/a)$ avem că $c/a = c \times (1/a) = c \times b$.

Dându-se un număr prim $P$, un întreg $D$ între 0 și $P - 1$ și un șir de $N$ numere, cuprinse fiecare între 0 și $P - 1$, se cere să se introducă între elementele șirului operatorii $+$, $-$, $\times$, / și parantezele corespunzătoare, astfel încât să se obțină o expresie corectă a cărei valoare să fie $D$ (lucrând în aritmetica modulo $P$). În caz că acest lucru nu este posibil, se va afișa un mesaj corespunzător.

**Intrarea**: Fișierul `INPUT.TXT` conține două linii:

- pe prima linie se găsesc trei numere întregi: $P$, $N$ și $D$, separate prin spații ($2 \leq P \leq 23$, $P$ prim, $1 \leq N \leq 30$, $0 \leq D \leq P - 1$);
- pe următoarea linie se găsește șirul de numere ce formează expresia ($N$ numere întregi cuprinse între 0 și $P - 1$).

**Ieșirea**: Fișierul text `OUTPUT.TXT` va conține o singură linie pe care se va găsi expresia generată sau mesajul „Nu există soluție.”. Expresia va fi parantezată complet (fiecare operator va avea o pereche de paranteze atașate).

**Exemple**:

| INPUT.TXT | OUTPUT.TXT |
|---|---|
| 11 3 6<br>4 7 9 | (4+(7/9)) |
| 11 3 7<br>1 1 1 | Nu exista solutie. |

**Timp de execuție** pentru un test: 1 minut.

Modificările și completările propuse de autor sunt:

- **Timp de execuție**: 10 secunde
- **Timp de implementare**: 1h 30 minute, maxim 1h 45 minute.
- **Complexitate cerută**: $O(N^3 \times P^2)$.

**REZOLVARE**: Problema în sine nu este foarte complicată. Ea face apel la programarea dinamică, dar este foarte asemănătoare cu una din problemele bine cunoscute de elevi - înmulțirea optimă a unui șir de matrice. Ceea ce o face mai „provocatoare" este timpul alocat implementării. De fapt, la respectiva probă au fost propuse trei probleme, cam de același nivel de dificultate, iar timpul total permis a fost de 4 ore. De asemenea, structurile de date impuse și modul de utilizare a lor sunt mai rar întâlnite.

Ideea de la care se pornește în rezolvarea acestei probleme este următoarea: Parantezarea și completarea cu operatori a unui șir de $N$ numere, $V = (V(1), V(2), \ldots, V(N))$, astfel încât să se obțină rezultatul $D$ este posibilă dacă și numai dacă există două valori $D_1$ și $D_2$, un număr întreg $K$ cuprins între 1 și $N-1$ și un operator $\oplus \in \{+, -, \times, /\}$ astfel încât următoarele condiții să fie îndeplinite simultan:

1. Este posibilă parantezarea și completarea cu operatori a șirului $V' = (V(1), V(2), \ldots, V(K))$ astfel încât să se obțină rezultatul $D_1$;
2. Este posibilă parantezarea și completarea cu operatori a șirului $V'' = (V(K+1), V(K+2), \ldots, V(N))$ astfel încât să se obțină rezultatul $D_2$;
3. $D_1 \oplus D_2 = D$

Cu alte cuvinte, trebuie să găsim un loc în care să „spargem" expresia noastră în așa fel încât, luând două valori care se pot obține pentru partea din stânga, respectiv din dreapta, și inserând între ele operatorul potrivit, să obținem valoarea $D$.

Pentru aflarea operatorului, a locului de împărțire a expresiei în două și a celor două valori necesare, ar fi de ajuns patru instrucțiuni repetitive `for`. Totuși, rămâne o singură întrebare: cum se poate verifica dacă se poate sau nu obține valoarea $D_1$ pentru subexpresia din stânga,

respectiv valoarea $D_2$ pentru subexpresia din dreapta? Aceasta este o subproblemă similară cu problema în sine, dar redusă la dimensiuni mai mici. O abordare directă ar fi comodă: se scrie o procedură care efectuează cele patru instrucțiuni for și, pentru fiecare combinație posibilă de operatori și operanzi, se reapelează recursiv ca să afle dacă valorile operanzilor se pot obține. Totuși, este intuitiv că această variantă va avea o complexitate uriașă, care o face inutilizabilă.

Motivul principal al nerentabilității acestei implementări este că ea reface de nenumărate ori exact aceleași calcule. Spre exemplu, dacă $N = 4$, programul va încerca să spargă vectorul $(V(1), V(2), V(3), V(4))$ în două părți. Există trei moduri posibile:

(a) $(V(1))$ și $(V(2), V(3), V(4))$;

(b) $(V(1), V(2))$ și $(V(3), V(4))$;

(c) $(V(1), V(2), V(3))$ și $(V(4))$;

Pentru a studia cazul (**c**), expresia stângă va trebui la rândul ei ruptă în două bucăți, lucru care poate fi făcut în două moduri:

(d) $(V(1))$ și $(V(2), V(3))$;

(e) $(V(1), V(2))$ și $(V(3))$;

Se observă că deja secvența $(V(1), V(2))$ a fost studiată de două ori (în cazurile (**b**) și (**e**)), iar secvența $(V(1))$ tot de două ori (în cazurile (**a**) și (**d**)). Exemplele pot continua. Dacă însă am reușit să nu mai evaluăm de două ori aceeași secvență, complexitatea programului s-ar reduce foarte mult. Pentru aceasta, trebuie să pornim cu secvențe foarte scurte (întâi cele de un singur număr, apoi cele de două numere), și să trecem la secvențe mai lungi, bazându-ne pe faptul că secvențele mai lungi se descompun în secvențe mai scurte care au fost deja analizate. Facem mențiunea că o secvență cu un singur număr poate fi parantezată într-un singur fel (practic nu este nevoie de paranteze și operatori) și poate produce un singur rezultat, egal cu valoarea numărului. O secvență de două numere poate produce maximum patru rezultate distincte, prin folosirea pe rând a celor patru operatori disponibili.

Pentru a stoca rezultatele obținute, vom folosi o matrice tridimensională $A$ de dimensiuni $N \times N \times P$. $A[i, j, r]$ indică dacă există vreo parantezare corespunzătoare a secvenței $(V(i), V(i+1), \ldots, V(j))$ astfel încât să se obțină rezultatul $r$. Dacă o asemenea parantezare există, $A[i, j, r]$ va indica punctul în care trebuie spartă în două expresia (printr-un număr între $i$ și $j-1$). Dacă nu există o asemenea parantezare, $A[i, j, r]$ va lua o valoare specială (0 de exemplu).

De aici decurge modul de inițializare al matricei:

$$\begin{cases} A[i,i,V(i)] = i, & \forall 1 \le i \le N \\ A[i,j,r] = 0, & \text{pentru orice alte valori ale lui } i,j \text{ și } r \end{cases} \tag{6.28}$$

Matricea se va completa pe diagonală, începând de la diagonala principală și terminând în colțul de NE. Pentru a afla toate valorile care se pot obține prin parantezarea secvenței $(V(i), V(i+1), \ldots, V(j))$, se va împărți această expresie în două părți disjuncte, în toate modurile posibile: $(V(i))$ și $(V(i+1), \ldots, V(j))$, apoi $(V(i), V(i+1))$ și $(V(i+2), \ldots, V(j))$ și așa mai departe până la $(V(i), V(i+1), \ldots, V(j-1))$ și $(V(j))$. Fiecăreia din părțile obținute îi va corespunde în matrice un element de forma $A[i,k]$ sau $A[k+1,j]$, cu $i \le k < j$. În orice caz, toate elementele de această formă se vor afla în matrice dedesubtul diagonalei din care face parte elementul $A[i,j]$, deci pentru secvențele respective se cunosc deja toate valorile pe care le pot lua prin parantezare și introducerea operatorilor. Tot ce avem de făcut este să combinăm în toate modurile aceste valori prin inserarea fiecăruia din cei patru operatori pentru a obține toate valorile posibile ale expresiei $(V(i), V(i+1), \ldots, V(j))$.

Dacă în final $A[1,N,D] \neq 0$, atunci problema are soluție. Vom vedea imediat și cum se reconstituie ea. Iată modul de compunere a matricei pentru exemplul din enunț (cu deosebirea că, în figurile de mai jos, $A[i,j]$ nu mai este un vector cu $P$ elemente, ci o mulțime de valori între 0 și $P-1$, această reprezentare fiind mai comodă):

$$A[1,1] = \{4\} \quad A[2,2] = \{7\} \quad A[3,3] = \{9\} \tag{6.29}$$

$$A = \begin{pmatrix} \{4\} & ? & ? \\ ? & \{7\} & ? \\ ? & ? & \{9\} \end{pmatrix} \tag{6.30}$$

Între numerele 4 și 7 plasăm cei patru operatori și obținem:

$$\begin{aligned} 4 + 7 &\equiv 0 \pmod{11} \\ 4 - 7 \equiv 4 + 4 &\equiv 8 \pmod{11} \\ 4 \times 7 &\equiv 6 \pmod{11} \\ 4 \;/\; 7 \equiv 4 \times 8 &\equiv 10 \pmod{11} \\ A[1,2] &= \{0,6,8,10\} \end{aligned}$$

Analog se procedează pentru numerele 7 și 9:

$$
\begin{aligned}
7 + 9 &\equiv 5 \pmod{11} \\
7 - 9 \equiv 7 + 2 &\equiv 9 \pmod{11} \\
7 \times 9 &\equiv 8 \pmod{11} \\
7 \,/\, 9 \equiv 7 \times 5 &\equiv 2 \pmod{11} \\
A[2,3] &= \{2,5,8,9\}
\end{aligned}
$$

$$
A = \begin{pmatrix} \{4\} & \{0,6,8,10\} & ? \\ ? & \{7\} & \{2,5,8,9\} \\ ? & ? & \{9\} \end{pmatrix} \tag{6.31}
$$

Pentru a calcula $A[1,3]$, putem grupa termenii în două moduri: Fie primul separat și ultimii doi separat, fie primii doi separat și ultimul separat. În primul caz, ultimii doi termeni - după cum s-a văzut - pot produce patru rezultate distincte (2, 5, 8 și 9). Combinând oricare din aceste rezultate cu primul termen (4) și adăugând orice operator, vor rezulta 16 valori posibile, din care evident unele vor coincide. Analog se procedează și pentru celălalt caz:

| Cazul I | Cazul II |
|---|---|
| $4 + 2 \equiv 6 \pmod{11}$ | $0 + 9 \equiv 9 \pmod{11}$ |
| $4 - 2 \equiv 2 \pmod{11}$ | $0 - 9 \equiv 2 \pmod{11}$ |
| $4 \times 2 \equiv 8 \pmod{11}$ | $0 \times 9 \equiv 0 \pmod{11}$ |
| $4 \,/\, 2 \equiv 2 \pmod{11}$ | $0 \,/\, 9 \equiv 0 \pmod{11}$ |
| $4 + 5 \equiv 9 \pmod{11}$ | $6 + 9 \equiv 4 \pmod{11}$ |
| $4 - 5 \equiv 10 \pmod{11}$ | $6 - 9 \equiv 8 \pmod{11}$ |
| $4 \times 5 \equiv 9 \pmod{11}$ | $6 \times 9 \equiv 10 \pmod{11}$ |
| $4 \,/\, 5 \equiv 3 \pmod{11}$ | $6 \,/\, 9 \equiv 8 \pmod{11}$ |
| $4 + 8 \equiv 1 \pmod{11}$ | $8 + 9 \equiv 6 \pmod{11}$ |
| $4 - 8 \equiv 7 \pmod{11}$ | $8 - 9 \equiv 10 \pmod{11}$ |
| $4 \times 8 \equiv 10 \pmod{11}$ | $8 \times 9 \equiv 6 \pmod{11}$ |
| $4 \,/\, 8 \equiv 6 \pmod{11}$ | $8 \,/\, 9 \equiv 7 \pmod{11}$ |
| $4 + 9 \equiv 2 \pmod{11}$ | $10 + 9 \equiv 8 \pmod{11}$ |
| $4 - 9 \equiv 6 \pmod{11}$ | $10 - 9 \equiv 1 \pmod{11}$ |
| $4 \times 9 \equiv 3 \pmod{11}$ | $10 \times 9 \equiv 2 \pmod{11}$ |
| $4 \,/\, 9 \equiv 9 \pmod{11}$ | $10 \,/\, 9 \equiv 6 \pmod{11}$ |

$$A[1,3] = \{0,1,2,3,4,6,7,8,9,10\}$$

$$A = \begin{pmatrix} \{4\} & \{0,6,8,10\} & \{0\ldots4,6\ldots10\} \\ ? & \{7\} & \{2,5,8,9\} \\ ? & ? & \{9\} \end{pmatrix} \tag{6.32}$$

Se observă că singura valoare care nu se poate obține prin parantezarea șirului (4, 7, 9) este 5. Să vedem acum și care este metoda de reconstituire a soluției. Fie $A[1,N,D] = X$. Dacă $X = 0$, atunci nu există soluție. Dacă $X \neq 0$, atunci $X$ indică poziția din vector după care trebuie inserat semnul. Nu se indică însă ce semn trebuie inserat, nici care sunt valorile care trebuie obținute pentru partea stângă, respectiv dreaptă. De aceea, vom căuta o combinație oarecare de valori $D_1$ și $D_2$ care se pot obține și un operator oarecare astfel încât $D_1 \oplus D_2 = D$. Odată ce le găsim, vom căuta, prin aceeași metodă, o modalitate de a obține valoarea $D_1$ în partea stângă a vectorului (știm sigur că această modalitate există) și o modalitate de a obține valoarea $D_2$ în partea dreaptă a vectorului.

Iată în continuare câteva detalii de implementare. Pentru efectuarea operațiilor matematice modulo $P$ s-a scris o funcție separată, $Expr$, care primește două numere $X$ și $Y$ între 0 și $P-1$ și un număr $Op$ între 1 și 4 reprezentând codificarea operatorului și întoarce un număr între 0 și $P$, reprezentând valoarea operației $(X\,Op\,Y)$. De fapt, ar trebui ca această funcție să întoarcă un număr între 0 și $P-1$ dacă operația este posibilă și să nu întoarcă nimic dacă operația este imposibilă, respectiv dacă se încearcă o împărțire la 0. Cum acest lucru nu este posibil în Pascal, am asimilat valoarea $P$ cu un cod de eroare, iar funcția va întoarce această valoare (care nu poate fi atinsă prin operații obișnuite) în cazul unei împărțiri prin 0. Mai departe, pentru a nu face un test separat dacă rezultatul funcției reprezintă o adresă valabilă în cea de-a treia dimensiune a tabloului, am preferat să supradimensionăm tabloul cu o unitate și să ignorăm tot ceea ce se scrie în coloana $P$.

Să ne ocupăm acum de operațiile aritmetice. Adunarea, scăderea și înmulțirea se fac în timp constant. O problemă apare în cazul împărțirii, deoarece ea nu mai seamănă deloc cu cea învățată pe mulțimea $\mathbb{R}$. Pentru a calcula $X/Y$, trebuie găsit acel număr $Z$ care, înmulțit cu $Y$, să dea $X$. Acest lucru se poate face într-o primă fază prin căutare secvențială (se încearcă valoarea 0, apoi 1, apoi 2 și așa mai departe; trebuie să existe un cât deoarece $P$ este prim). Tehnica are însă influențe neplăcute asupra complexității, supărătoare asupra timpului de rulare și dezastruoase asupra punctajului obținut. De aceea, este bine ca, în măsura în care timpul o permite, să se construiască o tabelă predefinită de calculare a inverșilor. Atunci, în loc să se efectueze împărțirea $X/Y$, se efectuează înmulțirea $X \times Y^{-1}$. Inversul unui element depinde și de modulul ales. Programul care urmează construiește un tabel care a fost importat în programul sursă ca o constantă, matricea `Invers` (`Invers[A,B]` este inversul lui $B$ modulo $A$). Liniile

corespunzătoare unor numere neprime au fost totuși inserate, pentru ușurința implementării.

```
program Invert;
const NMax=30;
      PMax=23;
var i,P:Integer;

function Invers(P,K:Integer):Integer;
var i:Integer;
begin
  i:=0;
  repeat Inc(i) until (K*i) mod P=1;
  Invers:=i;
end;

begin
  Assign(Output,'invers.txt');Rewrite(Output);
  for P:=2 to PMax do
    begin
      Write('{',P:2,'} (');
      for i:=1 to NMax do
        begin
          if (P in [2,3,5,7,11,13,17,19,23]) and (i<P)
            then Write(Invers(P,i):2)
            else Write(99);
          if i<>NMax then Write(',');
          if i=NMax div 2 then Write(#13#10'      ');
        end;
      WriteLn('),');
    end;
  Close(Output);
end.
```

Să analizăm în sfârșit complexitatea: Trebuie completată o matrice, deci $O(N^2)$ elemente. Pentru fiecare element din matrice, secvența corespunzătoare din vector trebuie spartă în două în toate modurile posibile, deci încă $O(N)$. Pentru fiecare descompunere, trebuie combinate în toate felurile toate valorile disponibile pentru partea stângă, respectiv dreaptă. Cum numărul de valori este $O(P)$, rezultă o complexitate de $O(P^2)$. Înmulțind toți acești factori rezultă o complexitate totală de $O(N^3 \times P^2)$. Dacă nu am fi făcut împărțirea a două numere modulo P în timp constant, ci în $O(P)$, atunci complexitatea totală ar fi fost $O(N^3 \times P^3)$, deci timpul de rulare putea fi și de 20 de ori mai mare.

Programul de mai jos pare îngrozitor de lung, dar, dacă avem în vedere faptul că o bună

bucată o reprezintă constanta `Invers`, care este generată, putem avea speranțe să-l scriem în timpul alocat.

```
program ParaNT;
{$B-,I-,R-,S-}
const NMax=30;
      PMax=23;
      NoWay=0;
      OpNames:String[4]='+-*/';
      Invers:array[2..PMax,1..NMax] of Integer=
{ 2}(( 1,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{ 3} ( 1, 2,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{ 4} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{ 5} ( 1, 3, 2, 4,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{ 6} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{ 7} ( 1, 4, 5, 2, 3, 6,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{ 8} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{ 9} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{10} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{11} ( 1, 6, 4, 3, 9, 2, 8, 7, 5,10,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{12} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{13} ( 1, 7, 9,10, 8,11, 2, 5, 3, 4, 6,12,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{14} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{15} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{16} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{17} ( 1, 9, 6,13, 7, 3, 5,15, 2,12,14,10, 4,11, 8,
      16,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{18} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99,99),
```

```
{19} ( 1,10,13, 5, 4,16,11,12,17, 2, 7, 8, 3,15,14,
       6, 9,18,99,99,99,99,99,99,99,99,99,99,99),
{20} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{21} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{22} (99,99,99,99,99,99,99,99,99,99,99,99,99,99,99,
      99,99,99,99,99,99,99,99,99,99,99,99,99,99),
{23} ( 1,12, 8, 6,14, 4,10, 3,18, 7,21, 2,16, 5,20,
      13,19, 9,17,15,11,22,99,99,99,99,99,99,99,99));

type Matrix=array[1..NMax, 1..NMax, 0..PMax] of Integer;
       { S-a inclus si valoarea PMax, care nu poate fi
         atinsa, pentru a se depozita "deseurile" }
     Vector=array[1..NMax] of Integer;
var A:Matrix;        { Matricea de calcul }
    V:Vector;        { Numerele }
    N,P,D:Integer;

procedure ReadData;
var i:Integer;
begin
  Assign(Input,'input.txt');Reset(Input);
  ReadLn(P,N,D);
  for i:=1 to N do Read(V[i]);
  Close(Input);
end;

function Expr(X,Y,Op:Integer):Integer;
{ Calculeaza expresia (X Op Y) unde Op=1 ('+'),
  Op=2 ('-'), Op=3 ('*'), Op=4 ('/'). Daca Op=4
  si Y=0 se returneaza valoarea P (care nu poate
  fi atinsa prin alte operatii corecte). }
begin
  case Op of
    1:Expr:=(X+Y) mod P;
    2:Expr:=(X+P-Y) mod P;
    3:Expr:=(X*Y) mod P;
    4:if Y=0 then Expr:=P  { = imposibil }
              else Expr:=(X*Invers[P,Y]) mod P
      { S-a creat o tabela predefinita de inversi,
        deoarece altfel impartirea se efectua numai
        in O(P) }
  end; {case}
end;
```

```
87
88  procedure Combine(i,j,k:Integer);
89  { Urmeaza a se combina toate valorile posibile
90    pentru A[i,k] si A[k+1,j] pentru a se afla
91    toate valorile posibile pentru A[i,j] }
92  var p1,p2,Op:Integer;
93  begin
94    for p1:=0 to P-1 do
95      if A[i,k,p1]<>NoWay
96        then for p2:=0 to P-1 do
97               if A[k+1,j,p2]<>NoWay
98                 then { Am gasit doua valori posibile
99                        si aplicam cei patru operatori }
100                     for Op:=1 to 4 do
101                       A[i,j,Expr(p1,p2,Op)]:=k;
102 end;
103
104 procedure ComposeMatrix;
105 var i,j,k,l:Integer;
106 begin
107   { Initializarea matricei }
108   for i:=1 to N do
109     for j:=1 to N do
110       for k:=0 to P-1 do A[i,j,P]:=NoWay;
111
112   for i:=1 to N do
113     A[i,i,V[i]]:=1; { sau orice <> NoWay }
114
115   for l:=2 to N do { Lungimea intervalelor }
116     for i:=1 to N-l+1 do
117       begin
118         j:=i+l-1; { S-au fixat [i,j] capetele intervalului }
119         for k:=i to j-1 do { Se alege locul de impartire }
120           Combine(i,j,k);
121       end;
122 end;
123
124 procedure SeekValues(Lo,Hi,Mid,Value:Integer;
125                      var v1,v2:Integer; var Op:Char);
126 { Se stie unde e "sparta" expresia in doua; se cauta
127   valorile care trebuie obtinute pentru partea stanga,
128   respectiv dreapta, si pentru operator }
129 var i,j,k:Integer;
130 begin
131   for i:=0 to P-1 do
```

```
132     if A[Lo,Mid,i]<>NoWay
133       then for j:=0 to P-1 do
134              if A[Mid+1,Hi,j]<>NoWay
135                then for k:=1 to 4 do
136                       if Expr(i,j,k)=Value
137                         then begin
138                                v1:=i;
139                                v2:=j;
140                                Op:=OpNames[k];
141                                Exit;
142                              end;
143 end;
144
145 procedure WriteExpression(Lo,Hi,Value:Integer);
146 var v1,v2,Place:Integer;
147     Op:Char;
148 begin
149   if Lo=Hi
150     then Write(V[Lo])
151     else begin
152            Place:=A[Lo,Hi,Value];
153            SeekValues(Lo,Hi,Place,Value,v1,v2,Op);
154            Write('(');
155            WriteExpression(Lo,Place,v1);
156            Write(Op);
157            WriteExpression(Place+1,Hi,v2);
158            Write(')');
159          end;
160 end;
161
162 procedure WriteSolution;
163 begin
164   Assign(Output,'output.txt');Rewrite(Output);
165   if A[1,N,D]=NoWay
166     then Write('Nu exista solutie')
167     else WriteExpression(1,N,D);
168   WriteLn;
169   Close(Output);
170 end;
171
172 begin
173   ReadData;
174   ComposeMatrix;
175   WriteSolution;
176 end.
```

O posibilă îmbunătățire a programului de sus ar fi să reținem în $A[i, j, r]$ nu numai locul unde se face secționarea expresiei, ci și operatorul introdus și eventual și valorile care trebuie obținute pe partea stângă, respectiv dreaptă. Totuși, volumul de date ar fi crescut corespunzător și ar fi devenit greu de manipulat. În schimb, în versiunea prezentă, programul merge puțin mai lent, dar nesesizabil. Să vedem de ce. Complexitatea reconstituirii expresiei în sine se află astfel: avem de reconstituit $O(N)$ operatori. Pentru fiecare din ei, trebuie să căutăm valorile stângă și dreaptă și operatorul în sine, deci $O(4 \times P^2)$. Complexitatea totală a reconstituirii datelor este $O(N \times P^2)$, adică oricum mult mai mică față de cea a compunerii matricei. Este preferabil să nu complicăm structurile de date și codul scris, mai ales că diferența ca timp de rulare este infimă.

Propunem ca temă cititorului o versiune a acestei probleme, în care nu se va mai lucra în inelul $\mathbb{Z}_P$, ci într-un grup cu elementele $a, b, c, d, \ldots$, pentru care se cunoaște tabela de compoziție. În acest caz avem un singur operator $\oplus$, iar cerința este să se parantezeze expresia $x_1 \oplus x_2 \oplus \cdots \oplus x_k$ astfel încât rezultatul să fie $y$.

## 6.11 Problema 11

Problema celui mai lung prefix a fost dată la a VIII-a Olimpiadă Internațională de Informatică, Veszprem 1996. Iată enunțul nemodificat al problemei:

**ENUNȚ**: Structura unor compuși biologici este reprezentată prin succesiunea constituenților lor. Acești constituenți sunt notați cu litere mari. Biologii sunt interesați să descompună o secvență lungă în altele mai scurte, numite primitive. Spunem că o secvență $S$ poate fi compusă dintr-un set de primitive $P$ dacă există $N$ primitive $p_1, \ldots, p_N$ în $P$ astfel încât concatenarea $p_1 p_2 \ldots p_N$ a primitivelor să fie egală cu $S$. Aceeași primitivă poate interveni de mai multe ori în concatenare și nu trebuie neapărat ca toate primitivele să fie prezente.

Primele $M$ caractere din $S$ se numesc prefixul lui $S$ de lungime $M$. Scrieți un program care primește la intrare un set de primitive $P$ și o secvență de constituenți $T$. Programul trebuie sa afle lungimea celui mai lung prefix al lui $T$ care se poate compune din primitive din $P$.

**Datele de intrare** apar în două fișiere. Fișierul `INPUT.TXT` descrie setul de primitive $P$, iar fișierul `DATA.TXT` conține secvența de examinat. Pe prima linie din `INPUT.TXT` se află $N$, numărul de primitive din $P$ ($1 \leq N \leq 100$). Fiecare primitivă se dă pe două linii consecutive: pe prima lungimea $L$ a primitivei ($1 \leq L \leq 20$), iar pe a doua un șir de litere mari de lungime $L$. Toate cele $N$ primitive sunt distincte.

Fiecare linie din fișierul `DATA.TXT` conține o literă mare pe prima poziție. El se termină cu

o linie conținând un punct („.”). Lungimea secvenței este cuprinsă între 1 si 500.000.

**Ieșirea**: Pe ecran se va tipări lungimea celui mai lung prefix din $T$ care poate fi compus din primitive din $P$.

**Exemplu**:

| **INPUT.TXT** | **OUTPUT.TXT** |
|---|---|
| 5 | A |
| 1 | B |
| A | A |
| 2 | B |
| AB | A |
| 3 | C |
| BBC | A |
| 2 | B |
| CA | A |
| 2 | A |
| BA | B |
| | C |
| | B |
| | . |

Pe ecran se va tipări numărul 11.

**Timp de rulare**: 30 secunde pentru un test.

**Timp de implementare**: 1h.

**Complexitate cerută**: $O(S \times L \times N)$, unde $S$ este lungimea secvenței.

**REZOLVARE**: Menționăm de la început că datele problemei sunt supradimensionate. Nici unul din cele zece teste cu care au fost verificate programele nu a depășit în realitate 12 primitive. În schimb, fișierul `DATA.TXT` a fost unic pentru toate testele și a conținut o secvență de lungime 500.000.

Problema se rezolvă și în acest caz prin reducerea ei la una similară, dar cu date de intrare mai mici. Respectiv, un prefix $S$ al lui $T$ se poate descompune în primitive dacă există o primitivă $p_i$ astfel încât $S = S' + p_i$ și $S'$ se poate descompune în primitive. Am redus împărțirea în primitive a lui $S$ la despărțirea în primitive a lui $S'$, care are o lungime mai mică decât $S$. O primă modalitate, pur teoretică, de a rezolva problema, este să reținem toate prefixele lui $T$ care se pot descompune în primitive; în felul acesta, putem studia prefixe din ce în ce mai lungi, bazându-ne pe prefixe mai scurte deja studiate. Totuși, este imposibil să ținem minte toate prefixele lui $T$, deoarece lungimea medie a unui prefix poate atinge 250.000 caractere.

O îmbunătățire care poate fi adusă acestui algoritm este următoarea: deoarece toate prefixele aparțin aceleiași secvențe de constituenți, $T$, este suficient să reținem în întregime secvența $T$ și, pentru fiecare prefix ce se poate descompune în primitive, păstrăm doar lungimea sa. Făcând abstracție de limitările de memorie, putem crea un vector $V$ cu $S$ variabile booleene (unde $S \leq 500.000$), iar $V[i]$ va indica dacă subșirul de lungime $i$ din $T$ se poate descompune în primitive. $V[i]$ va primi valoarea **True** dacă și numai dacă există o primitivă $p$ în $P$ de lungime $L$ astfel încât să fie îndeplinite simultan condițiile:

- $V[i-L] =$ **True**;
- secvența de caractere `T[i-L+1]T[i-L+2]...T[i]` este egală cu $p$.

Iată o primă variantă (în pseudocod) a algoritmului:

**Intrare:** $T$, primitivele
1: **pentru** $i = 1$ la $S$ **execută**
2: **dacă** există o primitivă $p$ astfel încât $V[i-L]$ și $T[i-L+1]T[i-L+2]\ldots T[i] = p$ **atunci**
3: $V[i] \leftarrow$ **True**
4: **altfel**
5: $V[i] \leftarrow$ **False**
6: **sfârșit dacă**
7: **sfârșit pentru**
8: caută cel mai mare $i$ pentru care $V[i] =$ **True**
9: **tipărește** $i$

Chiar și în acest caz, apare o problemă, deoarece avem nevoie de doi vectori, unul de caractere și altul de variabile booleene, ambii de lungime maxim 500.000. Necesarul de memorie este deci cam de 1MB. Sigur, pentru calculatoarele de astăzi această sumă este ușor de alocat, dar problema admite oricum o soluție la fel de rapidă și mult mai economică. Iată care este principiul:

Pentru a vedea dacă un prefix $S$ de lungime $L$ se poate descompune în primitive, noi avem nevoie să cunoaștem dacă prefixele de lungime mai mică se pot descompune. Dar avem oare nevoie de toate prefixele? Nu, deoarece noi vom concatena unul din prefixele de lungime mai mică cu o primitivă pentru a obține noul prefix $S$. Însă primitivele au lungime de maxim 20 caractere. Așadar, noi nu trebuie să cunoaștem decât dacă prefixele de lungime $L-1, L-2, \ldots, L-20$ se pot descompune; restul nu ne interesează. În felul acesta am eliminat vectorul $V$ și l-am redus la un vector de numai 20 de elemente (care în program se numește `CanGet`). La fiecare moment, când se prelucrează un nou caracter din secvența de constituenți $T$, primul element din `CanGet` se pierde (deoarece informația pe care el o stochează este învechită), iar

următoarele 19 elemente se deplasează spre stânga cu câte o poziție. Al 20-lea element, care acum a rămas disponibil, va fi calculat la pasul curent.

O altă modificare pornește de la observația că, datorită aceleiași limitări a lungimii primitivelor la 20 de caractere, nu avem nevoie nici măcar să reținem întregul vector $T$, ci numai ultimele 20 de litere ale lui. La fiecare pas, litera cea mai „veche" din $T$ (adică de indice minim) se va pierde, iar la celelalte 19 litere se va adăuga litera nou citită. Avem așadar nevoie doar de un string de 20 de caractere, pe care în program l-am numit `Last`. Pentru a deplasa spre stânga vectorii `CanGet` și `Last`, se pot folosi fie atribuirile succesive, fie rutinele de acces direct la memorie.

Deoarece prefixele care se pot descompune sunt identificate în ordinea crescătoare a lungimii, ultimul asemenea prefix găsit este tocmai cel de lungime maximă. Dar pentru că, în momentul în care găsim un prefix, nu putem ști dinainte că el este ultimul, trebuie să reținem într-o variabilă lungimea celui mai lung prefix găsit până la momentul respectiv, variabilă pe care o actualizăm de fiecare dată când găsim un nou prefix.

În felul acesta, am reușit să reducem memoria folosită aproape la strictul necesar, adică numai la dicționarul de primitive și la doi vectori de câte douăzeci de caractere. Se recomandă totuși să se aloce un buffer cât mai mare pentru citirea datelor din fișierul `DATA.TXT` pentru mărirea vitezei de citire. Repartizarea buffer-ului se face cu procedura Pascal `SetTextBuf`.

O optimizare care nu a fost inclusă în program, fiind lăsată ca temă cititorului, este următoarea: dacă la un moment dat, în timpul examinării secvenței de constituenți, este întâlnit un șir de cel puțin 20 de prefixe consecutive, din care nici unul nu se poate descompune în primitive, atunci nici mai departe nu vom mai întâlni vreun prefix care să se poată descompune. Explicați de ce. Această optimizare poate să nu aducă uneori nimic nou în evoluția programului, dar alteori poate să reducă la zero timpul de rulare.

Prezentăm mai jos codul sursă al programului. A fost preferat limbajul Pascal, deoarece pune la dispoziție rutine mai comode de manevrare a șirurilor de caractere.

```
1  program LongestPrefix;
2  {$B-,D-,I-,R-,S-}
3  const NMax=100;
4        LMax=20;
5  type Str20=String[LMax+1];
6       LexType=array[1..NMax] of Str20;
7       BooleanVector=array[1..LMax+1] of Boolean;
8       BufferType=array[1..62000] of Byte;
9  var Lex:LexType;      { Dictionarul de primitive }
10     N:Integer;        { Numarul de primitive }
11     Biggest:LongInt;  { Lungimea maxima }
```

```
12      Buf:BufferType;  { Buffer de intrare }
13
14  procedure ReadPrimitives;
15  var i,Len:Integer;
16  begin
17    Assign(Input,'input.txt');
18    Reset(Input);
19    ReadLn(N);
20    for i:=1 to N do
21      begin
22        ReadLn(Len);
23        ReadLn(Lex[i]);
24      end;
25    Close(Input);
26  end;
27
28  procedure Decompose;
29  var Last:Str20;  { Ultimele 20 de caractere citite }
30      CanGet:BooleanVector;
31       { CanGet[i] indica daca Last[i] poate fi
32         ultimul caracter al unei descompuneri }
33      i,L:Integer;
34      Current:LongInt; { Lungimea curenta }
35  begin
36    Assign(Input,'data.txt');
37    SetTextBuf(Input,Buf,SizeOf(Buf));
38    Reset(Input);  { Deschide fisierul cu un buffer atasat }
39
40    Biggest:=0;
41    Last[0]:=Chr(LMax+1);
42    FillChar(Last,LMax,'@');
43    FillChar(CanGet,LMax,True);
44    Current:=0;  { Deocamdata nu s-a citit nimic }
45
46    ReadLn(Last[LMax+1]); { Citeste primul caracter }
47    repeat
48      Inc(Current);
49      i:=0;
50      { Cauta o primitiva potrivita }
51      repeat
52        Inc(i);
53        L:=Length(Lex[i]);
54        CanGet[LMax+1]:=CanGet[LMax+1-L]
55          and (Copy(Last,LMax+2-L,L)=Lex[i]);
56      until CanGet[LMax+1] or (i=N);
```

```
57      { Am gasit o primitiva? }
58      if CanGet[LMax+1]
59        then Biggest:=Current;
60      { Deplaseaza spre stanga CanGet si Last }
61      Move(CanGet[2],CanGet[1],LMax);
62      Move(Last[2],Last[1],LMax);
63      { Avanseaza la urmatorul caracter }
64      ReadLn(Last[LMax+1]);
65    until Last[LMax+1]='.';
66
67    Close(Input);
68  end;
69
70  procedure WriteSolution;
71  begin
72    Assign(Output,'output.txt');
73    Rewrite(Output);
74    WriteLn(Biggest);
75    Close(Output);
76  end;
77
78  begin
79    ReadPrimitives;
80    Decompose;
81    WriteSolution;
82  end.
```

## 6.12 Problema 12

Această problemă a fost propusă la a IV-a Balcaniadă de Informatică, Nicosia 1996.

**ENUNȚ**: Grupul de rock U2 va da un concert în Nicosia. Un grup de $N \leq 200$ fani U2 așteaptă la coadă în scopul de a cumpăra bilete de la singura caserie deschisă. Fiecare persoană vrea să cumpere numai un bilet, iar casierul poate vinde unei persoane cel mult două bilete.

Casierul folosește $T[i]$ unități de timp pentru a servi al $i$-lea fan ($1 \leq i \leq N$). Este posibil totuși ca doi fani așezați la coadă unul după altul (de exemplu, al $j$-lea și al $j+1$-lea) să convină ca numai unul din ei să rămână la coadă, iar celălalt să plece, dacă timpul $R[j]$ ($1 \leq j \leq N-1$) în care casierul servește al $j$-lea și al $j+1$-lea fan este mai mic decât $T[j]+T[j+1]$. Deci, pentru a minimiza timpul de lucru al casierului, fiecare persoană din coadă încearcă să negocieze cu predecesorul și cu succesorul său, ceea ce va duce în final la o servire mai rapidă.

Fiind date numerele întregi pozitive $N$, $T[i]$ ($1 \leq i \leq N$) și $R[j]$ ($1 \leq j \leq N-1$), se

cere să se minimizeze timpul total al casierului. Acest lucru va fi realizat grupând într-un mod optim perechi de persoane consecutive. Atenție! Nu este necesar ca un anumit fan să se cupleze neapărat cu predecesorul sau cu succesorul său.

**Intrarea**: În fișierul `INPUT.TXT`, datele de intrare sunt date pe trei linii:

- prima linie conține numărul întreg $N$;
- a doua linie conține $N$ întregi: valorile $T[i]$, separate prin câte un spațiu;
- a treia linie conține $N$-1 întregi: valorile $R[j]$, separate de asemenea prin câte un spațiu;

Ieșirea se va face în fișierul `OUTPUT.TXT`, astfel:

- prima linie conține un întreg care reprezintă timpul total (minim) al casierului;
- pe fiecare din următoarele linii se află un singur număr sau două numere separate prin caracterul '+'. Mai exact, fiecare linie conține numărul $i$ dacă al $i$-lea fan este servit singur, sau $i + (i + 1)$ dacă cei doi fani sunt serviți ca o pereche.

**Exemplu**:

| **INPUT.TXT** | **OUTPUT.TXT** |
|---|---|
| 7 | 14 |
| 5 4 3 2 1 4 4 | 1 |
| 7 3 4 2 2 4 | 2+3 |
| | 4+5 |
| | 6+7 |

**Timp de rulare**: 15 secunde pentru un test.

Acesta este enunțul în forma lui de la Nicosia. Iată acum și completările „din studio":

- Numărul de fani este $N \leq 5.000$;
- **Complexitatea cerută** este $O(N)$;
- **Timpul de rulare** este de o secundă.

**REZOLVARE**: O primă metodă de rezolvare o vom lăsa în seama cititorului, întrucât ea este foarte asemănătoare cu rezolvarea problemei înmulțirii optime a unui șir de matrice. Ideea de pornire este de a defini o matrice $D$ cu $N$ linii și $N$ coloane, în care $D[X, Y]$ reprezintă timpul minim în care pot fi serviți fanii $X, X + 1, \ldots, Y - 1, Y$. Scopul este de a afla valoarea $D[1, N]$.

După cum se știe, însă, înmulțirea optimă a unui șir de matrice se poate determina în timp $O(N^3)$. Pentru condițiile inițiale ale problemei ($N \leq 200$), metoda se încadrează în timp. Ea putea fi deci folosită la concurs, pe câtă vreme dacă se impun condițiile suplimentare, rezolvarea în timp cubic nu mai dă rezultate. Iată ideea de rezolvare a problemei în timp liniar.

Vom denumi o **cuplare de ordin** $K$ modul în care primii $K$ fani sunt serviți câte unul sau câte doi. Fiecare cuplare are atașat un cost, respectiv timpul consumat de casier pentru a servi toți fanii. Dintre toate cuplările de ordin $K$, cele pentru care costul este minim (în cazul general pot fi mai multe) se vor numi **cuplări optime de ordinul** $K$. Cerința problemei exprimată cu noua terminologie este: să se găsească o cuplare optimă de ordinul $N$.

Să presupunem acum că am găsit cumva această cuplare optimă de ordinul $N$, pe care o vom nota cu $C_N$. Dacă în această cuplare al $K$-lea fan este servit de unul singur, atunci modul de servire al primilor $K-1$ fani reprezintă o cuplare optimă de ordinul $K-1$, pe care o vom nota cu $C_{K-1}$. Demonstrația nu este grea: dacă fanii $1, 2, \ldots, K-1$ nu ar fi cuplați în mod optim în cadrul lui $C_N$, atunci ar exista o cuplare a lor $C'_{K-1}$ de cost mai mic decât $C_{K-1}$. Dar această cuplare mai bună ar putea fi folosită pentru a obține o cuplare mai bună a tuturor celor $N$ fani (servind primii $K-1$ fani conform cuplării $C'_{K-1}$, iar pe ceilalți conform cuplării $C_N$). S-ar obține astfel o cuplare $C'_N$ de cost mai mic decât $C_N$, ceea ce este absurd, deoarece am presupus $C_N$ ca fiind optimă.

În mod absolut identic se poate demonstra că dacă $C_N$ este o cuplare optimă de ordinul $N$ în care fanii $K$ și $K+1$ sunt serviți împreună, atunci modul de servire al primilor $K-1$ fani reprezintă o cuplare optimă de ordinul $K-1$. Recunoaștem în aceste afirmații principiul programării dinamice: optimul global presupune optime locale. De aici deducem că, pentru a realiza o cuplare optimă a primilor $K$ fani avem nevoie de câte o cuplare optimă pentru primii $K-1$ fani, respectiv pentru primii $K-2$ fani, urmând ca apoi să aflăm care este costul minim al unei cuplări de ordin $K$, atât în ipoteza că fanul al $K$-lea este servit singur, cât și în ipoteza că el este servit împreună cu al $K-1$-lea fan.

Vom crea așadar doi vectori $T_1$ și $T_2$, ambii cu câte $N$ elemente, în care:

- $T_1[K]$ este timpul minim în care pot fi serviți fanii $1, 2, \ldots, K$, astfel încât fanul al $K$-lea să rămână singur;
- $T_2[K]$ este timpul minim în care pot fi serviți fanii $1, 2, \ldots, K$, astfel încât fanii $K$ și $K-1$ să fie cuplați.

Datele inițiale pe care le putem trece în cei doi vectori sunt:

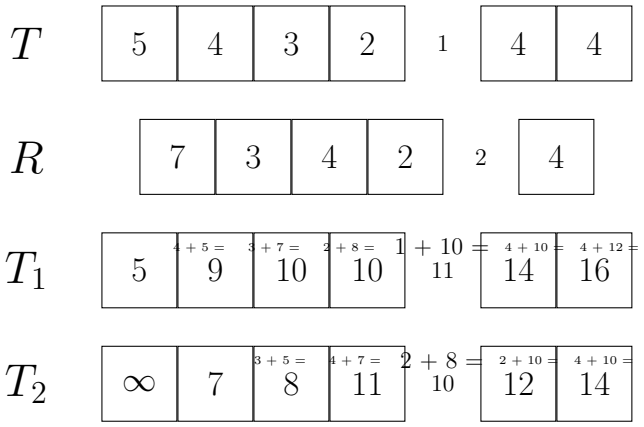
- $T_1[1] = T[1]$;
- $T_2[1] = \infty$ (un singur fan nu poate fi cuplat cu nimeni);

- $T_1[2] = T[1] + T[2]$ (dacă al doilea fan rămâne singur, atunci și primul rămâne singur);
- $T_2[2] = R[1]$.

Relațiile de recurență se deduc fără prea multă bătaie de cap:

- $T_1[K] = T[K] + \min(T_1[K-1], T_2[K-1])$ (dacă fanul $K$ rămâne singur, atunci el este servit în timpul $T[K]$, iar fanul $K-1$ se va cupla cu $K-2$ sau va rămâne singur, după cum este mai convenabil);
- $T_2[K] = R[K-1] + \min(T_1[K-2], T_2[K-2])$ (dacă fanul $K$ se cuplează cu $K-1$, atunci ei sunt serviți în timpul $R[K-1]$, iar fanul $K-2$ se va cupla cu $K-3$ sau va rămâne singur, după cum este mai convenabil);

Putem deci să completăm vectorii de la stânga la dreapta. Odată ce am făcut aceasta, costul minim al cuplării de ordinul $N$ este $\min(T_1[N], T_2[N])$. Pentru a reconstitui și așezarea fanilor, vom proceda recurent, astfel: dacă $T_2[N] < T_1[N]$, înseamnă că este mai avantajos ca fanii $N$ și $N-1$ să fie cuplați și reluăm reconstituirea pentru fanii $1, 2, \ldots, N-2$. În caz contrar, înseamnă că este mai avantajos ca fanul $N$ să rămână singur și reluăm reconstituirea pentru fanii $1, 2, \ldots, N-1$. De exemplu, iată modul în care se completează vectorii $T_1$ și $T_2$ pentru exemplul din enunț:




- $T_2[7] < T_1[7] \implies$ Fanii 6 și 7 sunt cuplați. Reconstituim așezarea primilor 5 fani.
- $T_2[5] < T_1[5] \implies$ Fanii 4 și 5 sunt cuplați. Reconstituim așezarea primilor 3 fani.
- $T_2[3] < T_1[3] \implies$ Fanii 2 și 3 sunt cuplați, iar primul fan este singur.

```
#include <stdio.h>
#define NMax 5001
typedef unsigned IntVector[NMax];
```

```
4  typedef long LongVector[NMax];
5
6  IntVector T, R;
7  LongVector T1, T2;
8  int N;
9  FILE *OutF;
10
11 void ReadData(void)
12 { FILE *F=fopen("input.txt", "rt");
13   int i;
14
15   fscanf(F,"%d\n", &N);
16   for (i=1; i<=N;)
17     fscanf(F,"%d", &T[i++]);
18   for (i=1; i<N;)
19     fscanf(F,"%d", &R[i++]);
20 }
21
22 long Min(long X, long Y)
23 {
24   return X<Y? X : Y;
25 }
26 void Match(void)
27 { int i;
28   T1[1]=T[1]; T2[1]=0xFFFF; /* =Infinit */
29   T1[2]=T[1]+T[2]; T2[2]=R[1];
30   for (i=3; i<=N; i++)
31     {
32       T1[i]=T[i]+Min(T1[i-1],T2[i-1]);
33       T2[i]=R[i-1]+Min(T1[i-2],T2[i-2]);
34     }
35 }
36
37 void GoBack(int K)
38 {
39   if (K)
40     if (T1[K]<T2[K])
41       { GoBack(K-1);
42         fprintf(OutF,"%d\n",K);
43       }
44       else { GoBack(K-2);
45              fprintf(OutF,"%d+%d\n",K-1,K);
46            }
47 }
48
```

```
49 void WriteSolution(void)
50 {
51   OutF=fopen("output.txt", "wt");
52   fprintf(OutF,"%d\n",Min(T1[N],T2[N]));
53   GoBack(N);
54   fclose(OutF);
55 }
56
57 void main(void)
58 {
59   ReadData();
60   Match();
61   WriteSolution();
62 }
```

## 6.13 Problema 13

Probabil că orice elev care are cât de cât experiență în programare a auzit despre **problema celui mai lung subșir crescător**. Când este prezentată la concurs, problema e „învelită" sub diverse forme. Aici o vom formaliza din punct de vedere matematic.

**ENUNȚ**: Se dă un vector cu $N$ elemente numere întregi. Se cere să se determine cel mai lung subșir crescător.

**Intrarea**: Fișierul de tip text `INPUT.TXT` conține pe prima linie numărul $N$ de elemente din vector ($N \leq 10.000$), iar pe fiecare din următoarele $N$ linii se află câte un element al vectorului.

**Ieșirea**: În fișierul de tip text `OUTPUT.TXT` se vor lista pe prima linie lungimea $L$ a celui mai lung subșir crescător, iar pe următoarele $L$ linii subșirul în sine. Dacă există mai multe soluții, se va tipări una singură.

**Exemplu**:

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

**Timp de implementare**: 45 minute.

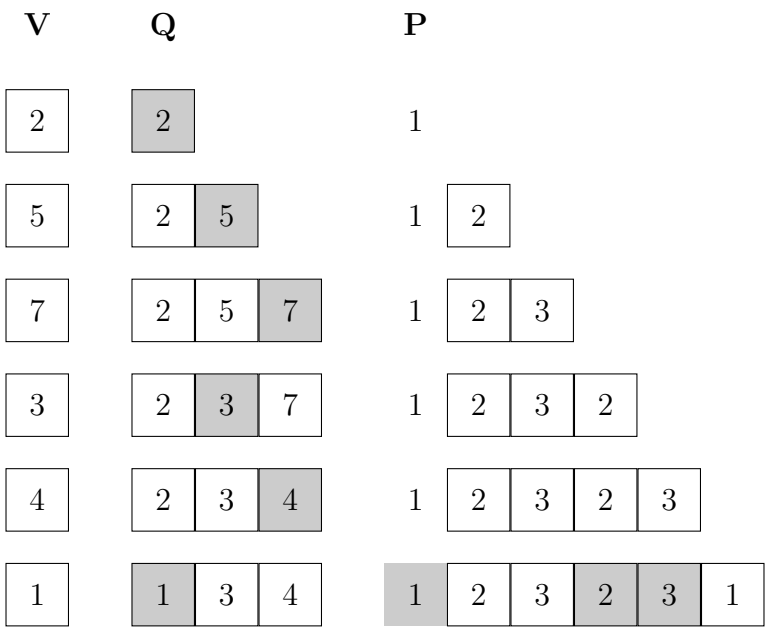
**Timp de rulare**: 5 secunde.

**Complexitate cerută**: $O(N \log N)$.

**REZOLVARE**: Începem prin a lămuri diferența dintre noțiunile de „subșir" și „subsecvență". Fie $V[1], V[2], \ldots, V[N]$ vectorul citit. Prin **subșir de lungime** $L$ al vectorului $V$ se înțelege o succesiune nu neapărat continuă de elemente $V[K_1], V[K_2], \ldots, V[K_L]$, unde $K_1 < K_2 < \cdots < K_L$. Prin **subsecvență de lungime** $L$ a vectorului, începând de la poziția $K$, se înțelege succesiunea continuă de elemente $V[K], V[K+1], \ldots, V[K+L-1]$.

Rezolvarea prin metoda programării dinamice este în general cunoscută și nu vom insista asupra ei. Probabil că aflarea celui mai lung subșir crescător este punctul de plecare al oricărui elev în învățarea programării dinamice. Totuși, această metodă de rezolvare are complexitatea $O(N^2)$. În continuare prezentăm o metodă mai puțin cunoscută și ceva mai dificil de implementat, dar mult mai eficientă. Ea are complexitatea $O(N \log N)$. Vom enunța principiul de rezolvare, urmat de o schiță a demonstrației de corectitudine.

Fie $V$ vectorul citit. Se parcurge vectorul de la stânga la dreapta și se construiesc în paralel doi vectori $P$ și $Q$, astfel: inițial vectorul $Q$ este vid. Se ia fiecare element din $V$ și se suprascrie peste cel mai mic element din $Q$ care este strict mai mare ca el. Dacă nu există un asemenea element în $Q$, cu alte cuvinte dacă elementul analizat din $V$ este mai mare ca toate elementele din $Q$, atunci el este adăugat la sfârșitul vectorului $Q$. Concomitent, se notează în vectorul $P$ poziția pe care a fost adăugat în vectorul $Q$ elementul din $V$. Iată cum se construiesc vectorii $P$ și $Q$ pentru exemplul din enunț:



Lungimea $L$ la care ajunge vectorul $Q$ la sfârșitul acestei prelucrări este tocmai lungimea celui mai lung subșir crescător al vectorului $V$. Pentru a afla exact și care sunt elementele subșirului crescător se procedează astfel: se caută ultima apariție în vectorul $P$ a valorii $L$. Să spunem că ea este găsită pe poziția $K_L$. Se caută apoi ultima apariție în vectorul $P$ a valorii $L-1$, anterior poziției $K_L$. Ea va fi pe poziția $K_{L-1} < K_L$. Analog se caută în vectorul $P$ valorile $L-2, L-3, \ldots, 2, 1$. Subșirul crescător este $S = (V[K_1], V[K2], \ldots, V[K_L])$.

În figura de mai sus au fost hașurate elementele respective găsite în vectorul $P$. Vectorul $Q$ are la sfârșit lungimea $L = 3$, deci cel mai lung subșir crescător are trei elemente. Se caută în vectorul $P$ cifrele 3, 2 și 1 și se găsesc pe pozițiile $K_1 = 1$, $K_2 = 4$ și $K_3 = 5$, ceea ce înseamnă că cel mai lung subșir crescător este (2, 3, 4).

Demonstrația (care, fără a pierde din corectitudine, face apel la intuiție...) folosește aceleași notații de mai sus și are următoarele etape:

**Propoziția 1.** *Vectorul $Q$ este în permanență sortat crescător.*

*Demonstraţie.* Folosim inducția matematică. După primul pas (inserarea elementului $V[1]$), vectorul $Q$ are un singur element și este bineînțeles ordonat. Trebuie acum arătat că dacă vectorul Q este sortat după inserarea elementului $V[i-1]$, el rămâne sortat și după inserarea lui $V[i]$. Într-adevăr, pentru elementul $V[i]$ există două variante:

a) $V[i]$ este mai mare decât toate elementele lui $Q$, caz în care este adăugat la sfârșitul lui $Q$, care rămâne sortat;

b) Există $t > 1$ astfel încât $Q[t-1] \leq V[i] < Q[t]$, caz în care $V[i]$ se suprascrie peste $Q[t]$. Dar $Q[t] \leq Q[t+1] \implies Q[t-1] \leq V[i] < Q[t+1]$ și vectorul $Q$ rămâne sortat.

□

Această deducție ne va fi utilă în calculul complexității algoritmului.

**Propoziția 2.** *Odată ce au fost scrise pentru prima oară, elementele vectorului $Q$ nu mai pot decât să scadă sau să rămână constante. Ele nu pot crește niciodată.*

*Demonstraţie.* Afirmația este evidentă, decurgând din modul de construcție a lui $Q$. □

**Propoziția 3.** *Elementele din vectorul $V$ de la poziția $K_{i-1}+1$ la poziția $K_i - 1$ sunt fie mai mici decât $V[K_{i-1}]$, fie mai mari decât $V[K_i]$.*

*Demonstraţie.* Acest lucru este natural, deoarece dacă ar exista $K_{i-1} < X < K_i$ astfel încât $V[K_{i-1}] \leq V[X] \leq V[K_i]$, atunci șirul $S$ ar putea fi extins cu elementul $V[X]$, deci nu ar fi subșir maximal, contradicție. □

**Propoziția 4.** *Toate elementele care urmează în $V$ după $V[K_L]$ sunt mai mici decât $V[K_L]$.*

*Demonstrație.* Dacă ar exista $X > K_L$ astfel încât $V[X] \geq V[K_L]$, atunci $S$ ar putea fi extins la dreapta cu $V[X]$, contradicție. □

**Propoziția 5.** *Elementul $V[K_1]$ este suprascris peste poziția $Q[1]$.*

*Demonstrație.* Dacă elementul $V[K_1]$ este inserat în $Q$ pe o poziție $t > 1$, rezultă că, în momentul tratării lui $V[K_1]$, pe poziția $t-1$ în vectorul $Q$ exista deja un număr $X < V[K_1]$. Aceasta înseamnă că $S$ poate fi prelungit la stânga cu $X$, deci $S$ nu este un subșir crescător maxim, contradicție. □

**Propoziția 6.** *Orice element $V[K_i]$ va fi suprascris în $Q$ pe poziția următoare celei pe care a fost scris $V[K_{i-1}]$*

*Demonstrație.* Să presupunem că $V[K_{i-1}]$ a fost scris peste $Q[t]$. Înainte de a ajunge să tratăm elementul $V[K_i]$, a trebuit să tratăm elementele $V[K_{i-1}+1], V[K_{i-1}+2], \ldots, V[K_i-1]$, care, după cum s-a stabilit la punctul (3), pot fi:

a) mai mici decât $V[K_{i-1}]$, caz în care ele vor fi scrise în vector pe poziții mai mici sau egale cu $t$, iar $Q[t]$ va scădea, deci $Q[t] \leq V[K_{i-1}]$ (conform punctului 2);

b) mai mari decât $V[K_i]$, caz în care vor fi scrise în $Q$ pe poziții mai mari decât $t$, așadar $Q[t+1] > V[K_i]$.

Se obține lanțul de inegalități $Q[t] \leq V[K_{i-1}] \leq V[K_i] \leq Q[t+1]$, de unde rezultă conform modului de construcție că $V[K_i]$ va fi scris peste $Q[t+1]$. Cu aceasta, folosind și punctul (5), am demonstrat că $V[K_i]$ este scris pe poziția $Q[i], \forall i = 1, 2, \ldots, L$.

□

După tratarea primelor $K_L$ elemente din $V$, vectorul $Q$ are lungimea $L$. Mai rămâne de văzut ce se întâmplă cu elementele $V[K_L+1], \ldots, V[N]$.

**Propoziția 7.** *Elementele care îi urmează lui $V[K_L]$ se scriu în $Q$ pe poziții mai mici sau egale cu $L$.*

*Demonstrație.* Acest fapt este intuitiv dacă se ține cont de punctul (4). Deducem că la final vectorul $Q$ are lungime $L$, adică aceeași cu a lui $S$. □

Modul de reconstituire a lui $S$ din vectorul $P$ este corect. Trebuie să avem grijă ca, atunci când căutăm în vectorul $P$ o apariție a valorii $X$, să o alegem pe ultima disponibilă, altfel

pot apărea erori. Spre exemplu, pentru vectorii dați în exemplu, $V = (2, 5, 7, 3, 4, 1)$ și $P = (1, 2, 3, 2, 3, 1)$, dacă se alege prima apariție a lui 2 (pe poziția a doua), avem subșirul $S = (2, 5, 4)$ care nu este crescător. Dacă însă alegem ultima apariție a lui 2 (pe poziția a patra), avem subșirul crescător maximal $S = (2, 3, 4)$.

Calculul complexității este ușor de făcut:

- Pentru construcția vectorilor $P$ și $Q$ sunt necesare $N$ inserții în vectorul $Q$, care este sortat; o inserție binară cere $O(\log N)$, așadar complexitatea primei părți este $O(N \log N)$.
- Pentru reconstituirea lui $S$ se face o singură parcurgere a vectorului $P$, deci $O(N)$.

Complexitatea totală a algoritmului este $O(N \log N)$.

```
#include <stdio.h>
#include <stdlib.h>
#define Infinity 30000
typedef int Vector[10001];

Vector V,P,Q,*S;
int N,Len; /* Len = Lungimea vectorului Q */

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

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

int Insert(int K,int Lo,int Hi)
{ int Mid=(Lo+Hi)/2;

  if (Lo==Hi)
    { if (Hi>Len) Q[++Len+1]=Infinity;
      Q[Lo]=K;
      return Lo;
    }
    else if (K<Q[Mid]) return Insert(K,Lo,Mid);
                       else return Insert(K,Mid+1,Hi);
}

```

```
30 void BuildPQ(void)
31 { int i,Place;
32
33   Len=0;Q[1]=Infinity;
34   for (i=1;i<=N;i++)
35     P[i]=Insert(V[i],1,Len+1);
36 }
37
38 void BuildS(void)
39 { int i,K=N;
40
41   S=malloc(sizeof(*S));
42   for (i=Len;i;i--)
43     { while (P[K]!=i) K--;
44       (*S)[i]=V[K];
45     }
46 }
47
48 void WriteSolution(void)
49 { FILE *F=fopen("output.txt","wt");
50   int i;
51
52   fprintf(F,"%d\n",Len);
53   for(i=1;i<=Len;i++) fprintf(F,"%d\n",(*S)[i]);
54
55   fclose(F);
56 }
57
58 void main(void)
59 {
60   ReadData();
61   BuildPQ();
62   BuildS();
63   WriteSolution();
64 }
```

Menționăm că în sursa de mai sus se putea construi vectorul $S$ peste vectorul $Q$, deoarece pentru construirea lui $S$ nu avem nevoie decât de elementele vectorului $P$. În acest fel, programul ar fi avut nevoie numai de trei vectori, iar volumul total de date nu ar fi depășit un segment. Am preferat totuși varianta în care vectorul $S$ este alocat dinamic pentru a evita confuziile.

## 6.14 Problema 14

Continuăm cu două probleme foarte asemănătoare. Atât de asemănătoare, încât diferența dintre ele pare - la o privire superficială - neglijabilă. Totuși, algoritmul de rezolvare se schimbă fundamental.

**ENUNȚ**: $N$ grămezi de mere trebuie împărțite la $N$ copii. Deoarece copiii sunt buni prieteni, trebuie ca împărțirea să se facă în mod echitabil, fiecare primind același număr de mere. Spiritul de dreptate al copiilor este atât de puternic, încât ei preferă ca unele mere să nu fie date nici unui copil, decât ca unii să primească mai multe mere ca alții. O condiție suplimentară este ca fie toate merele dintr-o grămadă să fie împărțite copiilor, fie grămada să nu mai fie împărțită deloc. Desigur, interesul este ca fiecare copil să primească un număr cât mai mare de mere.

Să se selecteze un număr de grămezi din cele $N$ astfel încât numărul total de mere să se dividă cu $N$, iar suma selectată să fie maximă. Dacă există mai multe soluții, se cere una singură.

**Intrarea**: Fișierul de intrare `INPUT.TXT` conține pe prima linie numărul $N$ de grămezi ($1 \leq N \leq 100$). Pe a doua linie se dau cantitățile de mere din cele $N$ grămezi ($N$ numere naturale pozitive, toate mai mici ca 200, separate prin spații).

**Ieșirea**: În fișierul `OUTPUT.TXT` se va scrie pe prima linie numărul maxim de mere găsit. Pe a doua linie se vor tipări indicii grămezilor selectate, în ordine crescătoare.

**Exemplu**:

| INPUT.TXT | OUTPUT.TXT |
|---|---|
| 4<br>3 2 5 7 | 12<br>1 2 4 |

**Timp de implementare**: 45 minute - maxim o oră.

**Timp de rulare**: 1 secundă.

**Complexitate cerută**: $O(N^2)$.

**REZOLVARE**: O primă modalitate, de altfel foarte comodă, este să verificăm toate posibilitățile de a selecta grămezi. Aceasta presupune să generăm toate submulțimile mulțimii de grămezi, iar pentru fiecare submulțime să calculăm suma merelor. În felul acesta putem afla submulțimea pentru care suma merelor se divide cu $N$ și este maximă. Din nefericire, această soluție, banal de implementat, are o complexitate exponențială, mai precis $O(N \times 2^N)$, deoarece există $2^N$ submulțimi ale mulțimii grămezilor și pentru fiecare submulțime putem calcula suma merelor în timp liniar. Prin urmare, suntem departe de complexitatea cerută în enunț.

Punctul de plecare pentru rezolvarea corectă a problemei este din nou principiul de opti-

malitate al programării dinamice. Să notăm cu $M[1], M[2], \ldots, M[N]$ cantitățile de mere din fiecare grămadă. Să considerăm că submulțimea optimă conține în total $S$ mere, iar grămada cu numărul $N$ face parte din ea. Fie $K$ restul împărțirii lui $M[N]$ la $N$, deci

$$M[N] \equiv K \pmod{N} \tag{6.33}$$

Atunci suma $S - M[N]$ dă restul $(N - K) \bmod N$ la împărțirea prin $N$, de unde deducem că

$$S - M[N] \equiv N - K \pmod{N} \tag{6.34}$$

De asemenea, putem afirma că $S - M[N]$ este cea mai mare dintre toate sumele care se pot obține folosind grămezile $1, 2, \ldots, N-1$ și care dau același rest la împărțirea prin $N$. Demonstrația nu este grea: dacă ar exista o sumă mai mare decât $S - M[N]$ congruentă cu $N - K$ modulo $N$, am putea folosi această sumă și grămada $M[N]$ pentru a obține o sumă $S' > S$ astfel încât

$$S' \equiv 0 \pmod{N} \tag{6.35}$$

Această observație ne sugerează și metoda de rezolvare a problemei. Pentru a afla care este submulțimea de sumă maximă, avem două variante:

- Grămada $N$ face parte din această submulțime, caz în care trebuie să descoperim cea mai mare sumă care se poate obține adunând grămezi dintre primele $N-1$ și care dă restul $N - K$ la împărțirea prin $N$;
- Grămada $N$ nu face parte din această submulțime, caz în care trebuie să descoperim cea mai mare sumă care se poate obține adunând grămezi dintre primele $N-1$ și care se împarte exact la $N$.

Pentru că nu putem ști de la început dacă grămada a $N$-a face sau nu parte din submulțimea maximală, trebuie să avem răspunsul pregătit pentru ambele situații. Mai mult, pentru a putea afla care sunt sumele maxime formate cu primele $N-1$ grămezi care dau diferite resturi (în cazul nostru 0 sau $N - K$) la împărțirea prin $N$, trebuie să reluăm exact aceeași problemă: grămada cu numărul $N-1$ poate face sau nu parte din submulțime. În concluzie, putem formaliza problema astfel: avem nevoie să putem răspunde la toate întrebările de forma „Care este suma maximă care se poate forma cu grămezi din primele $P$ astfel încât restul la împărțirea prin $N$ să fie $Q$?". Vom nota răspunsul la această întrebare cu $R[P, Q]$ (unde $1 \leq P \leq N$ și $0 \leq Q < N$). Răspunsurile tuturor întrebărilor se pot deci dispune într-o matrice $R$, iar scopul

nostru este să-l aflăm pe $R[N,0]$. Am observat că pentru a-l putea afla pe $R[P,Q]$ avem nevoie de două valori din linia $P$-1 a matricei (după cum grămada $P$ face sau nu parte din submulțimea optimă). Pentru a afla toate elementele liniei $P$ a matricei, este deci foarte probabil să avem nevoie de întreaga linie $P-1$.

Astfel, problema se reduce la a compune o linie a matricei din cea precedentă. Dacă presupunem că grămada $P$ nu intră în componența submulțimii optime, atunci linia $P$ este identică cu linia $P-1$:

$$R[P,Q] = R[P-1,Q], \quad \forall 0 \leq Q < N \tag{6.36}$$

Dacă presupunem că grămada $P$ face parte din submulțime, atunci avem egalitatea:

$$R[P,Q] = R[P-1,(Q-M[P]) \bmod N] + M[P], \quad \forall 0 \leq Q < N \tag{6.37}$$

Alegând dintre aceste variante pe cea care ne convine mai mult, obținem:

$$R[P,Q] = \max \left\{ \begin{array}{c} R[P-1,Q] \\ R[P-1,(Q-M[P]) \bmod N] + M[P] \end{array} \right\} \quad \forall 0 \leq Q < N \tag{6.38}$$

În felul acesta se completează fără nici un fel de probleme matricea $R$. După cum am spus, $R[N,0]$ indică suma maximă divizibilă cu $N$. Mai rămâne de văzut cum se face reconstituirea soluției. De fapt, nu avem decât să parcurgem aceleași etape ale raționamentului, dar în sens invers. Respectiv: dacă $R[N,0] = R[N-1,0]$, atunci deducem că grămada a $N$-a nu a fost folosită pentru a se obține suma maximă și avem nevoie să obținem suma maximă divizibilă cu $N$ din primele $N-1$ grămezi (adică $R[N-1,0]$). Dacă $R[N,0] \neq R[N-1,0]$, atunci grămada a $N$-a a fost folosită și avem nevoie să obținem suma maximă de rest $N-M[N]$ modulo $P$ folosind primele $N-1$ grămezi. Pe cazul general, pentru a obține suma maximă de rest $Q$ folosind primele $P$ grămezi, avem două posibilități:

- Dacă $R[P,Q] = R[P-1,Q]$, atunci grămada $P$ nu este folosită și trebuie să obținem același rest $Q$ folosind doar primele $P-1$ grămezi;
- Dacă $R[P,Q] \neq R[P-1,Q]$, atunci grămada $P$ este folosită și trebuie să obținem restul $Q-M[P]$ modulo $N$ folosind primele $P-1$ grămezi.

Menționăm că programul este puțin diferit de ceea ce s-a explicat până acum, în sensul că prima linie din matricea $R$ corespunde ultimei grămezi de mere, a doua linie corespunde penultimei grămezi de mere ș.a.m.d. Cu alte cuvinte, grămezile de mere sunt procesate în ordine inversă. Am făcut acest lucru pentru a ușura procedura de aflare a soluției; se observă că

prima oară se decide dacă ultima grămadă face parte din submulțime, apoi penultima etc. Deci și la găsirea soluției se generează grămezile în ordine inversă. Prin această dublă inversiune, indicii grămezilor de mere selectate vor fi listați în ordine crescătoare. Dacă am fi prelucrat grămezile de mere în ordinea lor din fișier, s-ar fi impus scrierea unei proceduri recursive pentru afișarea soluției.

Să facem și calculul complexității acestui program. Citirea datelor se face în $O(N)$, la fel și reconstituirea soluției. Pentru a completa o linie din matrice, avem nevoie să parcurgem linia precedentă, adică $O(N)$. Pentru compunerea întregii matrice, timpul necesar este pătratic.

Mai trebuie făcută o singură observație referitoare la modul de inițializare a matricei. Vom adăuga în matricea $R$ linia cu numărul 0, care va conține sumele maxime ce se pot obține fără a folosi nici o grămadă. Desigur, se poate obține numai suma 0, care dă restul 0 la împărțirea prin $N$, iar alte sume nu se pot obține. Vom pune deci $R[0,0] = 0$ și $R[0,i] =$ **Impossible** pentru orice $1 \leq i < N$, unde **Impossible** este o constantă specială (preferabil negativă, pentru a nu se confunda cu valorile obișnuite din matrice).

```
#include <stdio.h>
#define NMax 101
#define Impossible -30000

int M[NMax], R[NMax][NMax], N;
void ReadData(void)
{ FILE *F = fopen("input.txt", "rt");
  int i;

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

void Share(void)
{ int i,j;

  R[0][0]=0;
  for (i=1; i<N; R[0][i++]=Impossible);

  for (i=1; i<=N; i++)
    {
      for (j=0; j<N; j++)
        R[i][j] = R[i-1][j];
      for (j=0; j<N; j++)
        if (R[i-1][j] != Impossible
```

```
            && R[i-1][j] + M[N+1-i] >
               R[i][ (R[i-1][j]+M[N+1-i]) % N ])
          R[i][ (R[i-1][j]+M[N+1-i]) % N ] =
          R[i-1][j] + M[N+1-i];
    }
}

void WriteSolution(void)
{ FILE *F = fopen("output.txt", "wt");
  int i,j;

  fprintf(F, "%d\n", R[N][0]);
  j=0;
  for (i=N; i; i--)
    if (R[i][j] != R[i-1][j])
      {
        fprintf(F, "%d ", N+1-i);
        j = (j + N - M[N+1-i]%N) % N;
      }
  fprintf(F, "\n");
  fclose(F);
}

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

## 6.15 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<br>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

$$S(k) = \sum_{i=1}^{k} X(i) \tag{6.39}$$

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

$$S(k_2) - S(k_1) = \sum_{i=k_1+1}^{k_2} X(i) \tag{6.40}$$

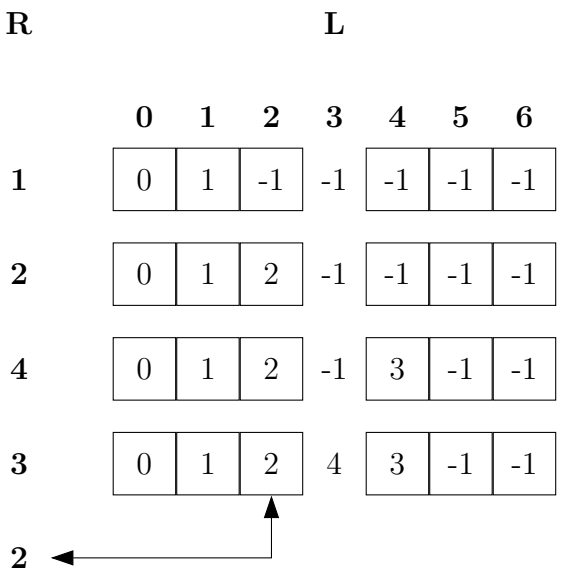
Dacă găsim în vectorul $S$ două valori $S(k_1)$ și $S(k_2)$ care dau același rest la împărțirea prin $P$, înseamnă că diferența lor se divide cu $P$, deci șirul de obiecte $k_1 + 1, k_1 + 2, \ldots, k_2$ poate constitui o soluție.

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  }
```

## 6.16 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<br>1 2 3 2 1 1 | 1 4<br>2 5<br>3 6<br>4 6<br>5 6 |
| 3<br>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,

$$\sum_{i=1}^{N} G[i] = 2 \cdot (N-1) \tag{6.41}$$

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 complexitate $N \log N$. 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:

$$N = 2 \cdot (N-1) \implies N = 2 \tag{6.42}$$

- 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 $N$.

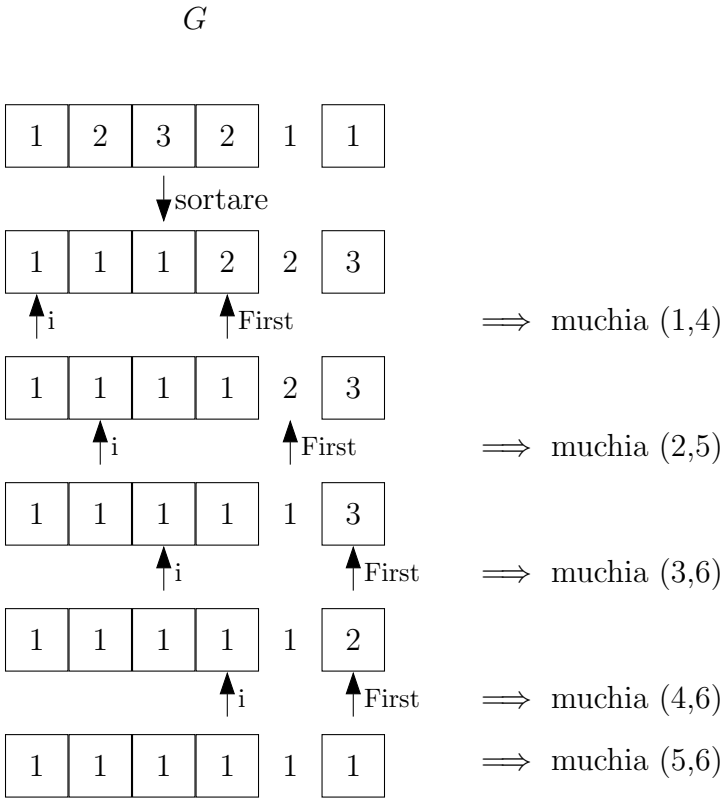
- Vectorul $G$ este în continuare sortat, deoarece $G[K-1] = 1 < G[K] \leq G[K+1]$ înainte de decrementarea lui $G[K]$, deci după decrementare vom avea $G[K-1] = 1 \leq G[K] < G[K+1]$, adică dubla relație de ordonare se păstrează.

- 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);
```

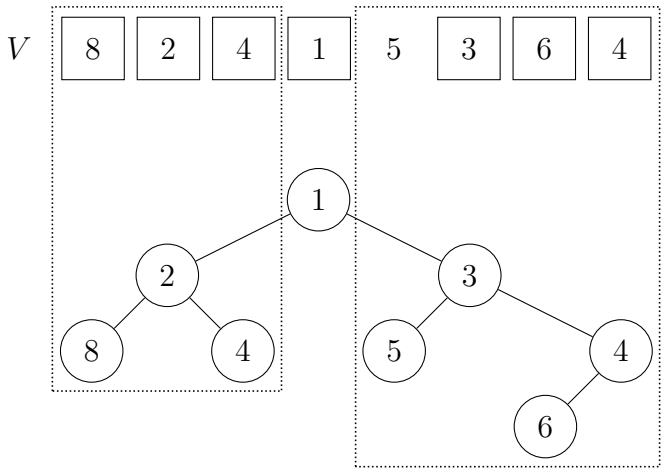
```
  for (i=1, Sum=0; i<=N; i++)
    { fscanf(F,"%d",&G[i]);
      Sum+=G[i];
    }
  fclose(F);
}

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);
    }
}
```

```
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 }
```

## 6.17 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:

$$V = (N \quad N-1 \quad N-2 \quad \dots \quad 2 \quad 1) \tag{6.43}$$

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

$$N + (N-1) + (N-2) + \cdots + 1 = \frac{N(N+1)}{2} \tag{6.44}$$

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

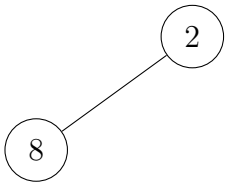
valoare mai mică sau egală cu $X$. Toate elementele care se aflau înainte în stivă pe poziții mai mari sau egale cu poziția pe care a fost inserat $X$ vor fi eliminate din stivă, iar elementul care se afla exact pe poziția lui $X$ va deveni fiul stâng al lui $X$. $X$ însuși va deveni fiul drept al predecesorului 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.

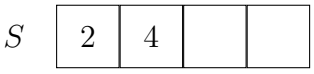
La început stiva este vidă. Primul element din V are valoarea 8, drept care îl vom pune în stivă, iar arborele cartezian va avea un singur nod:

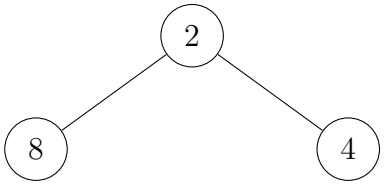


Următorul element sosit este 2. Acesta este mai mic decât 8, deci trebuie introdus înaintea lui în stivă. El va fi deci primul element din stivă și rădăcina arborelui cartezian la acest moment. Concomitent, 8 va fi eliminat din stivă și va deveni fiul stâng al lui 2:

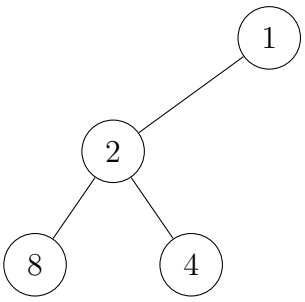




Se observă că arborele obținut este tocmai arborele cartezian atașat secvenței $(V[1], V[2])$. Următorul element este 4, care este mai mare decât 2, deci trebuie adăugat în vârful stivei. Nici un element nu este eliminat din stivă, iar 4 devine fiul drept al lui 2:

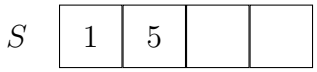


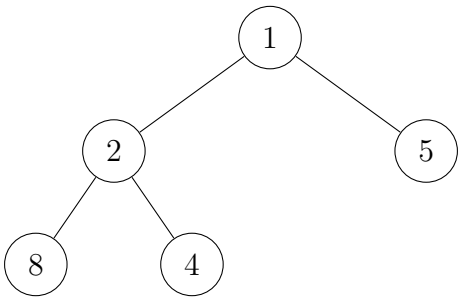



Următorul element sosit este 1, care este mai mic decât toate numerele din stivă. Stiva se va goli, iar numărul 2 (cel peste care se va scrie 1) va deveni fiul stâng al lui 1:

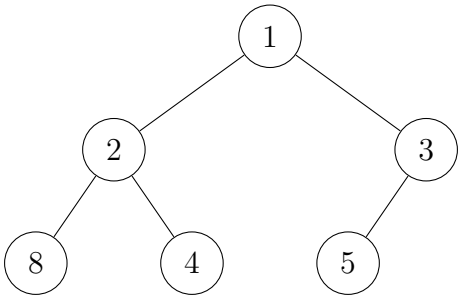




Deja arborele începe să semene cu forma sa finală. Urmează elementul 5, care va fi adăugat în stivă și „atârnat” în dreapta lui 1:

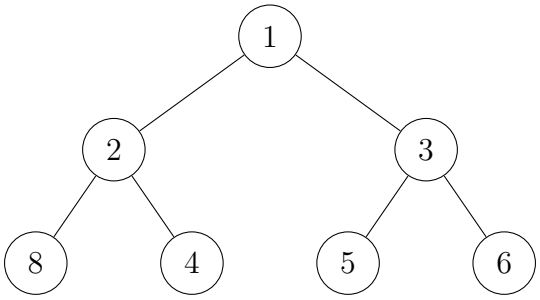


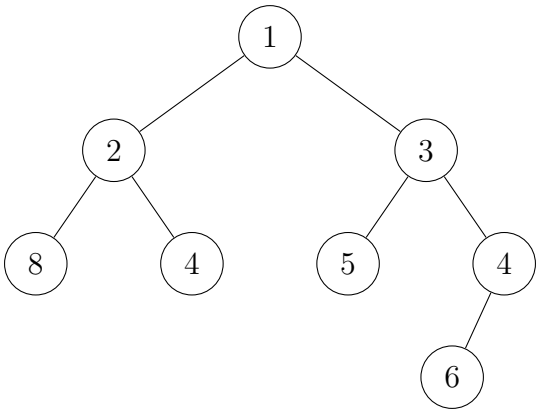

Elementul 3 este mai mare ca 1, căruia îi va deveni fiu drept, dar mai mic ca 5, pe care îl va elimina din stivă:

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:

$$\begin{aligned} O(\log 1 + \log 2 + \cdots + \log N) = O(\sum_{k=1}^{N} \log k) = \\ = O(\log \prod_{k=1}^{N} k) = O(\log N!) = O(N \cdot \log N) \end{aligned} \tag{6.45}$$

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 $T$).

```c
#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 */
```

```
27        T[i]=S[k-1];
28        if (k<=LenS) T[S[k]]=i;
29        /* i este ultimul element din stiva, deci... */
30        S[LenS=k]=i;
31      }
32  }
33
34  void WriteSolution(void)
35  { FILE *F=fopen("output.txt","wt");
36    int i;
37    for (i=1; i<=N;)
38      fprintf(F,"%d ",T[i++]);
39    fprintf(F,"\n");
40  }
41
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.

## 6.18 Problema 18

**ENUNȚ**: Se dă un vector nesortat cu elemente numere reale oarecare. Considerând că vectorul ar fi sortat, se cere să se găsească distanța maximă între două elemente consecutive ale sale,

fără însă a sorta efectiv vectorul.

**Intrarea**: Fișierul `INPUT.TXT` conține pe prima linie numărul $N$ de elemente din vector ($N \leq 5.000$). Pe următoare linie se dau numerele separate prin spații.

**Ieșirea**: Pe ecran se va tipări un mesaj de forma:

Distanța maximă este $D$

**Exemplu**: Pentru fișierul de intrare cu conținutul

```
4
5 3.2 2 3.7
```

răspunsul trebuie să fie

```
Distanta maxima este 1.3
```

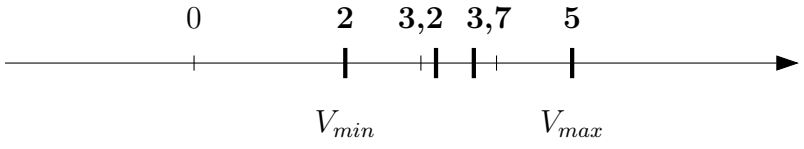
**Timp de implementare**: 30 minute.

**Timp de rulare**: 2-3 secunde.

**Complexitate cerută**: $O(N)$.

**REZOLVARE**: Desigur că primul lucru la care ne gândim este să sortăm vectorul și să îl parcurgem apoi de la stânga la dreapta, căutând distanța maximă între două elemente consecutive. Complexitatea unui asemenea algoritm este $O(N \log N)$. Nici această soluție nu este rea, iar la un concurs, comisiei de corectare i-ar veni destul de greu să găsească teste care să departajeze un algoritm în $O(N \log N)$ de unul liniar, chiar și pentru $N = 5.000$. Totuși, vom arăta care este algoritmul liniar; în primul rând de dragul „artei", iar în al doilea rând pentru că nu este cu mult mai greu de implementat decât o sortare.

Primul lucru care trebuie făcut este găsirea maximului și a minimului din vector; să notăm aceste valori cu $V_{max}$ și $V_{min}$. Aceste operații se fac în timp liniar, eventual chiar la citirea datelor din fișier. Apoi se împarte intervalul $[V_{min}, V_{max}]$ de pe axa reală în $N - 1$ intervale egale. Iată cazul exemplului din enunț, unde $V_{min} = 2$ și $V_{max} = 5$:




Lungimea fiecărui interval va fi deci de

$$D = \frac{V_{max} - V_{min}}{N - 1} \qquad \text{(în cazul nostru } D = 1\text{)} \tag{6.46}$$

De ce s-a făcut această împărțire? Dacă notăm cu $D_{max}$ distanța maximă pe axă între două numere vecine, adică tocmai valoarea pe care o căutăm, se poate demonstra că $D_{max} \geq D$. Într-adevăr, între cele $N$ numere de pe axă se formează $N-1$ intervale. Dacă presupunem că $D_{max} < D$, rezultă că distanța între oricare două numere consecutive de pe axă este mai mică decât $D$. De aici deducem că distanța dintre primul și ultimul număr, adică $V_{max} - V_{min}$, este mai mică decât $(N-1) \times D$. Dar aceasta duce la relația:

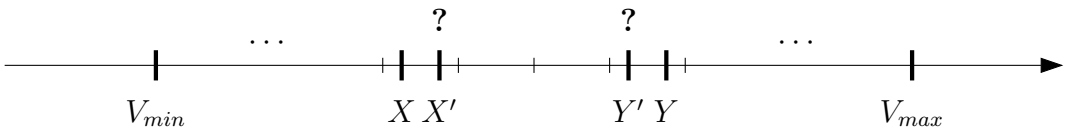
$$V_{max} - V_{min} < (N-1) \cdot \frac{V_{max} - V_{min}}{N-1} \implies V_{max} - V_{min} < V_{max} - V_{min} \tag{6.47}$$

relație care este absurdă; demonstrația afirmației $D_{max} \geq D$ este completă.

Următorul pas pe care îl avem de făcut este să parcurgem încă o dată vectorul de numere și să aflăm pentru fiecare element căruia dintre intervalele de lungime îi aparține. Și această operație se poate face în $O(N)$. Convenim ca dacă un număr $X$ se află exact la limita dintre două intervale, adică

$$X = V_{min} + k \cdot D, \quad 0 \leq k < N \tag{6.48}$$

el să fie considerat ca aparținând intervalului din dreapta. Aceasta înseamnă că elementul de valoare $V_{max}$ nu aparține nici unuia din cele $N-1$ intervale, ci celui de-al $N$-lea interval, $[V_{max}, V_{max} + D]$. Să vedem la ce ne ajută acest lucru. Din moment ce $D_{max} \geq D$, rezultă că este imposibil ca distanța maximă să se producă între două numere din același interval, deoarece distanța în cadrul aceluiași interval nu poate atinge valoarea $D$. Este deci obligatoriu ca distanța maximă să apară între două elemente din intervale distincte. Să urmărim în figura următoare ce alte proprietăți mai au aceste numere:




Dacă $X$ și $Y$ sunt valorile între care diferența este maximă, este de la sine înțeles că între ele nu mai există nici un număr, deoarece se prespune că $X$ și $Y$ sunt consecutive în vectorul sortat. Aceasta înseamnă însă că $X$ este cel mai mare număr din intervalul său, iar numărul $X'$ nu poate exista acolo unde a fost el figurat. Analog, $Y$ este cel mai mic număr din intervalul său, iar numărul $Y'$ nu poate exista. De fapt, în nici unul din intervalele dintre cele care le cuprind pe $X$ și $Y$ nu poate exista nici un număr.

Prin urmare, diferența maximă se poate produce numai între maximul unui interval și minimul imediat următor. Următorul pas în găsirea soluției presupune aflarea pentru fiecare din cele $N-1$ intervale (sau $N$ dacă îl considerăm și pe ultimul, cel care nu îl conține decât pe

$V_{max}$) a minimului și a maximului. Și acest pas se execută în $O(N)$, deoarece procesarea fiecărui element din vector se reduce la numai două comparări, cu minimul și cu maximul intervalului în care se încadrează el. Vor rezulta doi vectori care în program se vor numi $Lo$ și $Hi$. Iată care sunt valorile lor pentru exemplul nostru:

| **Intervalul**: | [2, 3) | [3, 4) | [4, 5) | [5, 6) |
|---|---|---|---|---|
| $Lo$: | 2 | 3,2 | – | 5 |
| $Hi$: | 2 | 3,7 | – | 5 |

Deoarece în intervalul [4,5) nu se află elemente, rezultă că elementele corespunzătoare din vectorii $Lo$ și $Hi$ trebuie să aibă o valoare specială care să informeze programul asupra acestui lucru. De exemplu, sursa oferită mai jos folosește următorul artificiu: inițializează vectorul $Lo$ cu valori foarte mari ($V_{max}+1$), astfel încât orice număr „repartizat" într-un interval să modifice această valoare. Similar, vectorul $Hi$ este inițializat cu $V_{min}-1$. Dacă pentru un interval aceste valori se păstrează până la sfârșit, putem trage concluzia că în respectivul interval nu se află nici un număr.

În continuare, elementele vectorilor $Lo$ și $Hi$ se amestecă formând un nou vector $W$ care de data aceasta este sortat. Sortarea este foarte ușoară, pentru că nu avem decât să așezăm numerele în ordinea $Lo[1]$, $Hi[1]$, $Lo[2]$, $Hi[2]$, ..., $Lo[N]$, $Hi[N]$. Deși la prima vedere pare că noul vector rezultat are $2N$ elemente, de fapt el are numai $N$ elemente, pentru că:

- Dacă într-un interval $K$ există un singur număr, (cazul intervalelor [2,3) și [5,6)) sau există numai numere egale, atunci $Lo[K] = Hi[K]$ și este suficient să copiem în $W$ una singură dintre aceste două valori;
- Dacă într-un interval nu există nici un număr, putem să nu copiem nici o valoare în vectorul $W$.

Astfel, construcția vectorului $W$ se poate face în timp liniar, mai exact în $O(2N)$. Se observă că la această construcție se poate întâmpla ca unele numere să „dispară", adică să nu fie trecute în vectorul $W$. De exemplu, dacă între numerele 3,2 și 3,7 ar mai fi existat un număr, 3,5, el nu ar fi fost nici minim, nici maxim pentru intervalul său, deci nu ar fi fost copiat. Totuși, trierea în acest fel a elementelor nu afectează în nici un fel soluția. În cazul nostru, nu se întâmplă să dispară nici un element, deci $W = (2, 3, 2, 3, 7, 5)$.

După ce am construit vectorul $W$, nu mai avem decât să-l parcurgem de la stânga la dreapta și să tipărim diferența maximă întâlnită între două numere consecutive (repetăm, vectorul $W$ este sortat), această ultimă etapă necesitând și ea un timp liniar. Nu se poate obține o complexitate inferioară celei liniare, întrucât citirea datelor presupune ea însăși $N$ operații.

Ca un detaliu de implementare, odată ce au fost construiți vectorii $Lo$ și $Hi$, vectorul $V$ nu mai este necesar, deci putem construi chiar în el vectorul $W$, pentru a economisi memorie.

```
#include <stdio.h>
#define NMax 5000
float V[NMax+1], Lo[NMax+1], Hi[NMax+1];
float Delta, Max, Min;
int N;

void ReadData(void)
/* Citeste vectorul si afla maximul si minimul */
{ FILE *F=fopen("input.txt", "rt");
  int i;

  fscanf(F, "%d\n", &N);
  fscanf(F, "%f", &V[1]);
  Max=Min=V[1];

  for (i=2; i<=N; i++)
    {
      fscanf(F, "%f", &V[i]);
      if (V[i]>Max) Max=V[i];
        else if (V[i]<Min) Min=V[i];
    }
  fclose(F);
}

void Split(void)
{ int i, K;

  Delta = (Max-Min)/(N-1);
  /* Se initializeaza vectorii Lo si Hi */
  for (i=0; i<=N; Lo[i]=Max+1, Hi[i++]=Min-1);

  /* Se construiesc intervalele */
  for (i=1; i<=N; i++)
    {
      K = (V[i]-Min)/Delta;
      if (V[i]<Lo[K]) Lo[K]=V[i];
      if (V[i]>Hi[K]) Hi[K]=V[i];
    }
}

void Rebuild(void)
```

```
42 /* Rescrie vectorul V, pentru a economisi memorie,
43    pastrand numai capetele intervalelor */
44 { int i, M=0;
45
46   for (i=0;i<N; i++)
47     {
48       if (Lo[i] != Max+1) V[++M]=Lo[i];
49       if (Hi[i] != Min-1 && Hi[i] != Lo[i]) V[++M]=Hi[i];
50     }
51   N=M;
52 }
53
54 void FindGap(void)
55 /* Acum cautarea distantei maxime se face
56    secvential, vectorul fiind sortat */
57 { int i;
58   float Gap=0;
59
60   for (i=2; i<=N; i++)
61     if (V[i]-V[i-1] > Gap)
62       Gap = V[i]-V[i-1];
63   printf("Distanta maxima este %0.3f\n", Gap);
64 }
65
66 void main(void)
67 {
68   ReadData();
69   if (Max==Min)
70     puts("0");
71   else
72     {
73       Split();
74       Rebuild();
75       FindGap();
76     }
77 }
```

# Bibliografie

[1] Donald E. Knuth, *The Art of Computer Programming*, Addison Wesley, Massachusetts, 1973. Tradusă în limba română de către Editura Tehnică cu denumirea *Tratat de programarea calculatoarelor*.

[2] T.H. Cormen, C.E. Leiserson., R.L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1992.

[3] Răzvan Andonie, Ilie Gârbacea, *Algoritmi fundamentali - o perspectivă C++*, Editura Libris, Cluj-Napoca, 1995.

# Cuprins