CRANGĂ CLEOPATRA GEORGETA

PROBLEME DE INFORMATICĂ

Referent ştiinţific: **Prof.dr. Victor MITRANA**
Editor: **Dragon Mihai NEDELCU**
Machetare grafică: **Dragon Mihai NEDELCU**

Descrierea CIP a Bibliotecii Naţionale a României
CRANGĂ CLEOPATRA GEORGETA
Probleme de informatică/ Crangă Cleopatra Georgeta. - Drobeta Turnu-Severin : Editura Ştef, 2023
Conţine bibliogr.
ISBN 978-606-991-236-2

CRANGĂ CLEOPATRA GEORGETA

PROBLEME DE INFORMATICĂ

2023

OP 4 CP 401
Drobeta Tr.-Severin
+40 744 495 186

e-mail: office.editurastef@yahoo.com
office.editurastef@gmail.com

ISBN: 978-606-991-236-2

Ce şanse am să devin un bun programator ?

Această întrebare apare deseori în discuţiile sincere dintre profesori şi studenţii lor descurajaţi de întîrzierea apariţiei unor rezultate care să certifice buna lor calitate ca programatori. Vom încerca în rîndurile ce urmează să răspundem cît mai clar la această întrebare oferind, în plus, o perspectivă prospătată asupra acestui subiect, prin luarea în considerare a unei serii de factori mai puţin utilizaţi în procesul didactic contemporan.

Mai întîi să vedem ce s-ar putea înţelege prin sigtagma ***"bun programator",*** insisitînd în continuare doar pe aprofundarea adjectivului *bun,* fără a mai defini sau detalia ce se înţelege printr-un programator. Vom cita cuvintele recente ale lui Timoty Budd (profesor la Oregon State University) care dă următoarea definiţie: "*Un bun programator trebuie să fie înzestrat cu tehnică, experienţă, capacitate de abstractizare, logică, inteligenţă, creativitate şi talent*". Întru-totul de acord cu această definiţie vom trece în cele ce urmează la explicitarea fiecărei calităţi.

Înainte vom deduce următoarea consecinţă imediată - deosebit de importantă - ce rezultă din definiţia de mai sus: **cele şapte calităţi trebuie să fie prezente *toate*** pentru a se obţine calificativul de *bun programator.* Deci, prin lipsa sau prin prezenţa "atrofiată" a uneia , sau a mai multe din "ingredientele reţetei" de mai sus, acest calificativ nu mai poate fi atins.

1. Tehnica – este desigur o calitate ce poate fi, şi este, dobîndită doar prin aplicarea asiduă (conform proverbului: "exerciţiul îl face pe maestru") în activitatea concretă de programare a tehnicilor de programare învăţate şi asimilate de către programator în timpul formării sale profesionale. Nu este exclusă aici posibilitatea obţinerii tehnicii de programare înafara unui cadru specializat (într-o facultate de profil), ci chiar există

posibilitatea obţinerii ei prin studiu individual şi formaţie proprie (autodidact).

2. Experienţa – este perechea geamănă a calităţii de mai înainte, fără însă a se exclude una pe cealaltă. Nu vom mai repeta cum şi în ce condiţii poate fi ea obţinută ci vom deduce următoarea consecinţa imediată : *nici un programator începător nu poate fi numit* **bun programator** *întrucît el nu a avut cînd (adică timpul necesar) să dobîndească ambele calităţi*. Este binecunoscut faptul că o rubrică importantă ce se cere completată la angajare sau la schimbarea locului de muncă este *experienţa de programare în ani.* Se consideră în general că experienţa apare abia după minimum doi ani de programare. Acest fapt nu trebuie privit ca o descurajare pentru cei mai tineri programatori ci mai degrabă ca pe un motiv de ambiţionare şi ca o invitaţie la rapidă autoperfecţionare.

3. Abstractizarea – este o trăsătură a intelectului uman şi constituie un dat al oricărui om normal, dar din păcate(!) este o însuşire prea puţin dezvoltată şi prea puţin folosită de oamenii obişnuiţi. Ea constă din capacitatea de a extrage din context, de a vedea dincolo de suprafaţa imediată şi de a putea sesiza structura – scheletul ce susţine întreaga reţea de detalii ale unei probleme generale. Pentru a fi un bun programator acestă calitate trebuie să fie net amplificată faţă de "normal" întrucît stă la baza oricărui proces de analiză şi modelare a problemelor, cît şi la baza procesului de proiectare a soluţiilor generale. Absenţa sau mai exact atrofierea acestei capacităţi se constată practic la studenţi prin incapacitatea de a înţelege sau de a asimila explicaţii, demonstraţii sau modele abstracte (simplu spus, o acută şi permanentă "*lipsă de chef*" atunci cînd sînt atinse anumite subiecte ce nu mai au contact direct cu realitatea concretă, imediată – adică subiecte abstracte). Metoda pentru a recăpăta sau a amplifica această capacitate este de a face cît mai des uz de ea, adică de a o exersa mereu (conform zicalei "*funcţia creează organul*") într-un domeniu particular, susţinut de o motivaţie personală puternică. Altfel spus, capacitatea noastră de

abstractizare se va amplifica dacă vom încerca găsirea de soluţii la problemele dintr-unul din **domeniile noastre preferate**, pentru că rezolvarea acestora va fi automotivată, făcută "cu chef" şi va prezenta o doză sporită de atractivitate.

4. Logica – este o altă calitate intrinsecă a oricărui intelect sănătos. Ea este absolut necesară atît pentru a putea folosi mecanismele mentale de deducţie şi inducţie logică, cît şi pentru a putea înţelege uşor, dar în acelaşi timp corect, cursul – firul roşu al unei demonstraţii sau al unui raţionament întins pe mai multe pagini. Asemenea tuturor calităţilor intrinseci existente în stare potenţială, antrenarea şi amplificarea acesteia se face prin exerciţiu repetat, prin folosirea ei în mod curent.Din păcate, doar prin rezolvarea de integrame nu se ajunge la amplificarea logicii…

5. Inteligenţa – este una din cele mai de preţ calităţi intrinseci ale intelectului uman. În cîteva cuvinte, fără a avea pretenţia de a da prin acestea o definiţie, prin inteligenţă înţelegem capacitatea de a face (de a stabili) conexiuni sau legături noi şi folositoare (din latinescul *inter-legere*) între idei, cunoştinţe sau informaţii "aparent fără legătură". Faţă de logică, pe care o considerăm ca fiind o calitate bazală, inteligenţa este o calitate ce se "întinde pe verticala" intelectului şi are în plus trăsătura de a fi mult mai dinamică şi mai mobilă (chiar fulgerător de rapidă) în acţiune. Pentru cultivarea, amplificarea şi cizelarea acestei calităţi este nevoie de "punerea ei la lucru" cît mai des şi pe durate tot mai mari de timp. Insatisfacţia obţinerii unor rezultate rapide sau chiar imediate este un obstacol ce poate fi depăşit relativ uşor prin antrenarea inteligenţei pe un "teren" cunoscut şi accesibil, adică în **domeniul preferat de interes**. În acest fel există siguranţa de a fi susţinut de atracţia sporită pentru acel domeniu particular ceea ce va conduce prin efort perseverent (dar susţinut de această dată cu pasiune !) la apariţia rezultatelor aşteptate şi, implicit, a satisfacţiei.

6. Creativitatea – este o calitate intrinsecă nu numai intelectului uman ci însăşi vieţii în general. Ea constă, în ultimă

instanţă, în capacitatea de a face (de a produce) ceva cu ***adevărat nou şi original***. De aceea am putea afirma că toate organismele vii, prin capacitatea lor de a se opune entropiei, creează mereu ordine din dezordine aducînd în acest fel ceva nou, neaşteptat. Ceea ce se aşteaptă însă de la un bun programator nu este doar acest tip de creativitate (gen: *adaptare inconştientă şi instinctivă*) ci o creativitate conştientă, responsabilă, reflectată în adaptarea soluţiilor existente sau chiar inventarea altora noi. În acest sens trebuie să menţionăm că există o legătură strînsă, dovedită şi verificată în practică (chiar dacă pare oarecum inexplicabil la prima vedere), între *creativitate – inteligenţă fluidă – curiozitate – sublimarea impulsurilor erotice - umor şi poftă de viaţă*. **Cultivarea şi amplificarea controlată** a oricărora dintre aceste patru trăsături va conduce în mod automat la amplificarea şi dinamizarea creativităţii intelectuale.

7. Talentul – este singura calitate ce nu poate fi cultivată şi amplificată. În accepţiunea sa obişnuită, prin talent se înţelege o sumă de înzestrări native sau o predispoziţie personală pentru un anumit domeniu. Existenţa talentului este percepută de cel în cauză ca ***uşurinţă – abilitate - dexteritate*** de a învăţa, asimila şi aplica toate cunoştinţele domeniului respectiv, abilitate ce este simţită de cel "talentat" ca un fel de *"ceva în plus"* în comparaţie cu capacităţile celor din jur. Din păcate, în accepţiunea comună se crede că talentul este calitatea suficientă care permite oricui atingerea cu siguranţă a calificativului *bun programator*, concepţie este infirmată de orice programator cu experienţă. Asta nu înseamnă că lipsa talentului în programare este permisă pentru atingerea acestui nivel, însă efortul, tenacitatea şi răbdarea existente în "cantităţi" mult sporite într-o asemenea situaţie de ne-înzestrare cu talent vor permite o apropiere sigură de acest calificativ. Din păcate, lipsa talentului va apărea la început sub forma unei insatisfacţii interioare şi ca o impresie acută că lipsesc rezultatele. Reamintim că însăşi cuvîntul *facultate* are la origine sensul de *capacitate, potenţialitate, înzestrare.* Deci, normal ar fi ca alegerea unui student pentru

frecventarea cursurilor unei Facultăţi să fi fost făcută ţinînd cont de aptitudinile şi abilităţile celui în cauză, descoperite în prelabil, adică să se dovedească talentat pentru domeniul ales. Acest lucru este cu atît mai important în cazul optării pentru învăţarea *programării,* cunoscută fiind ca o specializare complexă şi solicitantă.

Reluînd în sinteză ideile prezentate, putem spune că:

- Pentru a fi un *bun programator* trebuie să fie prezente următorele şapte calităţi într-o formă activă, dinamică: *tehnică, experienţă, capacitate de abstractizare, logică, inteligenţă, creativitate şi talent*.
- Dintre toate cele şapte calităţi necesare programării de înaltă calitate, numai una – *talentul* - nu este inerentă unui intelect sănătos. De altfel, prezenţa talentului nu este absolut necesară pentru a deveni programator, dar în timp ce absenţa lui îngreunează apropierea de calificativul *bun programator,* prezenţa lui şi amplificarea celorlalte calităţi este o garanţie a succesului, ce va fi cu siguranţă obţinut, însă nu fără efort, răbdare şi perseverenţă !
- Toate celelalte şase calităţi excluzînd talentul, *prezente fiind într-o formă potenţială*, trebuiesc doar *cultivate şi amplificate*. Am prezentat mai sus în detaliu modul de amplificare a fiecăreia.
- *"Cheia secretă"* ce conduce *cu siguranţă* la declanşarea procesului de dinamizare şi amplificare a fiecăreia din cele şase calităţi inerente este de *a avea mereu o motivaţie puternică* (de a învăţa "cu chef" sau "cu tragere de inimă" !). Acest fapt este posibil dacă se ţine cont de necesitatea adaptării efortului la domeniul preferat al celui în cauză. La modul concret, este necesar ca toate aplicaţiile, problemele, exerciţiile, întrebările, curiozităţile, inovaţiile, descoperirile, "săpăturile", etc., să fie făcute sau să fie alese, la început, din domeniul preferat (hobby-ul), chiar dacă acesta nu are la prima vedere legătură cu programarea. Scopul ce se atinge cu siguranţă în acest mod în

această primă fază este acela de a pune “la lucru” inteligenţa, creativitatea, logica, etc., ceea ce va conduce cu siguranţă la trezirea şi amplificarea rapidă a acestor calităţi. Acest fapt va permite apoi trecerea la o a doua fază în care, pe baza acumulărilor calitative obţinute, se poate trece la programarea propriu-zise “înarmat cu forţe proaspete”.

Încheiem răspunzînd într-o singură frază întrebării din titlu *Ce şanse am să devin un bun programator ?* :

dacă mă simt înzestrat cu *talent* pentru programare (adică nu mă simt inconfortabil la acest subiect) atunci, mobilizîndu-mi voinţa (motivaţia) şi amplificîndu-mi *capacitatea de abstractizare, logica, inteligenţa şi creativitatea (*ce există în mine într-o formă potenţială), prin practică de programare voi acumula în timp *tehnica* şi *experienţa* necesare pentru a deveni cu siguranţă un *bun programator* , însă nu fără efort, răbdare şi perseverenţă.

Legile succesului durabil

Cunoaşte-ţi Regulile de aur ale studentului şmecher ? Dacă nu, le puteţi fi afla “la o bere”, de la şmecher la şmecher. Noi le vom numi "Anti-legile succesului durabil" şi vi le prezentăm în continuare doar pentru a putea observa cum fiecare din aceste "legi" este o răsturnare (pervertire) a adevăratelor legi ale succesului.

1. Cel mai important este să termini facultatea şi să te vezi cu diploma în mînă. Ce contează cum ? Cine mai ştie dup-aia… ?

2. De ce să-nveţi …? Şi aşa majoritatea materiilor sînt tembele şi n-o să-ţi folosească niciodată în viaţă. …materiile tembele trebuie să fie predate numai pentru ca să cîştige şi profii’ o pîine.

3. Pune-te bine cu profesorii pînă treci examenul. Stai cu ei la o ţigară în pauză. Lasă-i pe ei să vorbească. Tu prefă-te că eşti interesat…

4. Ai trecut examenul ? Da ? Atunci… restul nu mai contează.

5. Nu contează dacă ai învăţat, ce ştii sau cît ştii. Important este să ai baftă la examen, să ai mînă bună sau să mergi "bine pregătit"… La puţini profi' nu se poate copia !

6. Sînt examene la care, se ştie bine, toată lumea copiază. Trebuie să fi nebun să-nveţi la ele !

7. Notele bune sînt numai pentru piloşi şi tocilari.

Acestor studenţi le sînt însă complet necunoscute **Legile succesului durabil.** Ele ar putea fi intuite doar de acei puţini care s-au format şi s-au educat în spiritul ideilor ce urmează să le explicăm în continuare. Aceste legi ne învaţă că bazele succesului durabil se pun încă din timpul şcolii şi mai ales din timpul facultăţii. Şi ne mai învaţă că succesul astfel "start-at" este destinat să dureze o viaţă întreagă.

1. ***Cel mai important în facultate este să-ţi faci o carte de vizită***, nu-i suficient să "vînezi" doar diploma. Dacă vei fi apreciat şi vei ajunge să fii considerat capabil sau chiar bun de cadrele didactice "cu greutate", vei ajunge să fi cunoscut şi bine cotat după absolvire şi-ţi vei găsi un loc bun de muncă. **Întotdeauna a fost şi va fi nevoie de oameni capabili** "pe piaţa muncii", nu de licenţiaţi "piloşi", "tolomaci" sau "papagali".

2. ***Cel mai important lucru în şcoală este că înveţi cum să înveţi.*** Cînd vrei să te recreezi rezolvînd integrame nu prea contează ce din ce domeniu ţi le-ai ales. Important pentru tine nu este cum, ci faptul că te destinzi. Tot astfel, în facultate important este nu neapărat ce înveţi, **ci că înveţi**! Multe cunoştinţe le vei uita în primii ani după absolvire, mai ales cele pe care ţi le-ai însuşit într-o stare de sforţare şi încrîncenare, fără plăcere. **Cel mai important este să înveţi de plăcere** căci numai aşa vei învăţa cum să înveţi. Iar aceasta nu se mai poate uita! Şi nu vei mai uita nicicînd că ai resursele şi puterea să treci prin forţe proprii examenele cele mai grele.

3. ***Succesul în viaţă se bazează pe relaţii umane echilibrate.*** (Acest fapt era cunoscut şi pe vremea regimului partidului comunist român P.C.R. însă datorită imoralităţii generalizate a societăţii el a fost aplicat pe invers: astfel, a apela de P.C.R. însemna atunci să apelezi la Pile, Cunoştinţe şi Relaţii.) Deci, **cel mai important lucru în şcoală este să înveţi arta de a stabilii relaţii umane prietenoase şi de încredere reciprocă**. Ceea ce va conta cel mai mult, peste ani, este că ai stabilit în timpul şcolii multe prietenii durabile şi de încredere care te vor "îmbogăţii" astfel pentru toată viaţa. În plus, nu uita: şi profesorii sînt oameni. Au şi ei nevoie de prieteni.

4. ***Colegii sînt martori şi devin cei mai exigenţi judecători ai trăsăturilor tale de caracter*.** Examenul, indiferent de materie sau disciplină, cu emoţiile şi peripeţiile lui este în sine o lecţie completă. Nu contează atît dacă l-ai luat sau dacă l-ai picat, **ci contează cum**! Contează ce fel de om eşti în astfel de situaţii, cînd tocmai îţi construieşti "cartea de vizită sau blazonul". Nu uita că nu te afli doar în faţa profesorilor ci eşti tot timpul înconjurat de colegii care te judecă, chiar dacă ţi-e nu-ţi spun. Pentru că **aşa cum te comporţi acum în examen, aşa te vei comporta toată viaţa**.

5. ***Examenele grele sînt cele care îţi pot forma un caracter puternic.*** Ceea ce este important în examen, ca şi în situaţiile de viaţă, este încrederea în reuşită şi stăpînirea de sine chiar dacă n-ai învăţat toată materia. **Dacă ai învăţat destul ca să te simţi stăpîn pe tine atunci ai trecut examenul !** Chiar acesta a fost rostul lui, ce dacă ţi-a dat notă mică! Crezi că, după ce vei trece examenul, peste zece ani îţi vei mai aminti cu ce notă?

6. ***Cei cu un caracter slab şi vicios se vor da la un moment dat în vileag*.** Cei care copiază nu-şi dau seama că ei îşi "infectează" caracterul. Şi nici cît de grave sînt consecinţele infectării cu "microbul" cîştigului imediat obţinut prin furt. Oare se vor mai putea debarasa vreodată de acest viciu tentant ? Dar de cunoscutele "efecte secundare": sentimentul de nesiguranţă

fără o fiţuică în buzunar, atracţia irezistibilă pentru "aruncarea privirii" împrejur, părerea de rău că "Ce prost sînt, puteam să copiez tot !", etc. cînd vor mai scăpa ? **Cei care se obişnuiesc să copieze, atît cît vor trăi, vor fi jumătate om-jumătate fiţuică**. Ca în vechile bancuri cu miliţieni…

7. ***Oricine este acum apt să înveţe şi să-şi însuşească pentru întreaga sa viaţă Legea efortului.*** Pe profesori îi impresionează cel mai tare efortul depus şi-l vor aprecia cu note maxime. Ei supra-notează pe cei "care vor, sînt bine intenţionaţi, dar încă nu pot". Profesorii cunosc adevărul exprimat în *Legea omului de geniu (legea lui Einstein)*: "**Geniul este compus 99% din transpiraţie şi 1% din inspiraţie**". Profesorii adevăraţi se străduiesc să noteze mai ales calitatea umană şi profesională a studentului. Reţineţi: dacă studentul a fost prietenos, activ şi deschis în timpul anului şcolar şi a depus un efort constant pentru a se perfecţiona, fapt ce nu a scăpat ochiului atent al profesorului, examenul devine în final pentru el o formalitate…

Multe vorbe şi păreri pot fi auzite pe această temă în familie, în pauze la şcoală sau la barul preferat. Cît sînt ele de adevărate ? S-ar putea da oare o definiţie precisă pentru *succesul în viaţă* ?

Noi nu cunoaştem o astfel de definiţie, ştim doar că există o multitudine de păreri şi opinii, unele profund contradictorii. Este însă de bun simţ să credem că se poate numi "de succes" acea viaţă care este plină de satisfacţii, bucurii şi visuri împlinite. Acea viaţă care să-şi merite din plin exclamaţia: "Asta da, viaţă !" ?

Regula de aur a succesului durabil este: **Învaţă să-ţi construieşti singur viaţa.** Şi apoi, dacă ai învăţat, apucă-te fără întîrziere să-ţi "faci" viaţa fericită.

Studenţia, prin entuziasmul, optimismul şi idealismul ei, este o perioadă optimă pentru a învăţa cum **să-ţi faci o viaţă de succes** ! Atenţie, mulţi şi-au dat seama prea tîrziu că studenţia a fost pentru ei în multe privinţe ultimul tren…

Probleme de judecată

Oferim în cele ce urmează o selecţie de probleme ce nu necesită cunoştinţe de matematică avansate (doar nivelul gimnazial) dar care pun la încercare capacitatea de judecată, inspiraţia şi creativitatea gîndirii. Rezolvarea acestor probleme constituie un bun antrenament pentru creşterea capacităţii de gîndire creativă precum şi a fluidităţii gîndirii. Credem că nu degeaba aceste două trăsături sînt considerate cele mai importante semne ale tinereţii minţii.

Problemele, selectate din multiple surse, nu au putut fi grupate în ordinea dificultăţii mai ales datorită diversităţii şi varietăţii lor. Ele au fost doar separate în cîteva categorii a căror nume vrea să sugereze un anumit mod de gîndire pe care l-am folosit şi noi în rezolvarea lor. Cele cu un grad mai mare de dificultate au fost marcate cu un semn (sau mai multe semne) de exclamare.

Criteriul principal pe baza căruia s-a făcut această selecţie a fost următorul: **fiecare problemă cere în rezolvarea ei un minimum de inventivitate şi creativitate**. Majoritatea problemelor te pun "faţă în faţă cu imposibilul", aşa că rezolvarea fiecărei probleme necesită depăşirea unor "limitări ale gîndirii" plus un minimum de originalitate în gîndire. Tocmai de aceea, pentru rezolvarea lor este nevoie de efort, putere de concentrare şi perseverenţă. Zis într-un singur cuvînt: este necesar şi un strop de **pasiune**.

Considerăm că eforturile consecvente ale celor care vor rezolva aceste probleme vor fi din plin răsplătite prin plăcerea "minţii biruitoare" şi prin amplificarea calităţilor următoare: capacitate sporită de efort intelectual, putere de concentrare mărită şi prospeţime în gîndire.

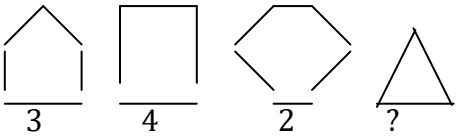
Vă dorim mult succes !

Probleme de perspicacitate

1. Ştiind că o sticlă cu dop costă 1500 lei şi că o sticlă fără dop costă 1000 lei, cît costă un dop ?

2. Ştiind că un ou costă 1000 lei plus o jumătate de ou, cît costă un ou ?

3. Ce număr lipseşte alături de ultima figură:



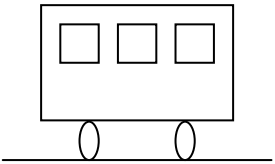
4. Lui Popescu nici prin gînd nu-i trecea să folosească toate mijloacele pe care le avea la îndemînă ca să lupte împotriva adversarilor tendinţei contra neintroducerii mişcării anti-fumat. Care este poziţia lui Popescu: este *pentru* sau *contra* fumatului ?

5. **Împărţirea "imposibilă".** Să se împartă numărul 12 în două părţi astfel încît fiecare parte să fie 7.

6. **9 puncte.** Să se secţioneze toate cele 9 mici discuri cu o linie frîntă neîntreruptă (fără a ridica creionul de pe hîrtie) compusă din 4 segmente. (!) Dar din trei segmente, este posibil ?

7. **Trei cutii.** În trei cutii identice sînt închise trei perechi de fructe: fie o pereche de mere, fie o pereche de pere, fie o pereche formată dintr-un măr şi o pară. Pe cele trei cutii sînt lipite trei etichete: "două mere", "două pere" şi, respectiv, "un măr şi o pară". Ştiind că nici una din etichete nu corespunde cu conţinutul cuitei închise pe care se află, să se afle care este *numărul minim* de extrageri a cîte un fruct pentru a se stabili conţinutul fiecărei cutii.

8. În ce direcţie merge autobuzul din desenul alăturat ?



9. **(!) Întrerupătoarele.** Pe peretele alăturat uşei încuiate de la intrarea unei încăperi, se află trei întrerupătoare ce corespund cu cele trei becuri de pe plafonul încăperii în care nu putem intra. Acţionînd oricare din întrerupătoare, dunga de lumină care apare pe sub uşă ne asigură că niciunul din cele trei becuri nu este ars. Cum putem afla, fără a pătrunde în încăpere, care întrerupător corespunde cu care bec ?

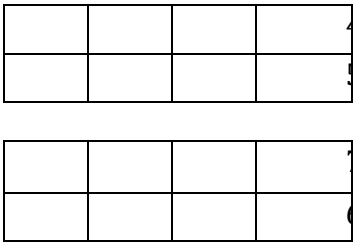
10. (!!) Cine mută ultimul cîştigă. Doi jucători dispun de o masă de joc de formă circulară sau pătrată şi de un număr mare de monezi identice. Ei mută plasînd pe masa de joc în spaţiul neocupat, fără suprapunere, cîte o monedă alternativ pînă cînd unul dintre jucători, care pierde în acest caz, nu mai poate plasa nicăieri o monedă. Să se arate că primul jucător are o strategie sigură de cîştig.

11. **(!!!) Iepurele şi robotul-vînător.** Într-o incintă închisă (un gen de arenă) se află un iepuraş şi un robot-vînător înzestrat cu cleşti, mijloc de deplasare, calculator de proces şi "ochi" electronici. Ştiind că viteza de deplasare a robotului-vînător este constantă şi de zeci de ori mai mare decît a iepuraşului, ce şanse mai are iepuraşul de a scăpa ?

12. **Cîntarul defect.** Avînd la dispoziţie un cîntar gradat defect care greşeşte constant cu aceeaşi valoare (cantitate necunoscută de grame), putem să cîntărim ceva determinîndu-i corect greutatea ?

13. Jocul dubleţilor (inventat de Carroll Lewis). Ştiind că trecerea de la un cuvînt cu sens la altul cu sens este permisă doar prin modificarea unei singure litere odată (de exemplu: UNU → UNI → ANI → ARI → GRI → GOI → DOI) se cere: *Dovediţi că IARBA este VERDE şi că MAIMUŢA a condus la OMENIRE, faceţi din UNU DOI, schimbaţi ROZ-ul în ALB, puneţi ROUGE pe OBRAZ şi faceţi să fie VARA FRIG.*

14. Împăturirea celor 8 pătrate. Împăturiţi iniţial în opt o foaie dreptunghiulară după care desfaceţi-o şi însemnaţi fiecare din cele opt zone dreptunghiulare obţinute (marcate de pliurile de îndoire) cu o cifră de la 1 la 8. Puteţi împături foaia astfel obţinută reducînd-o de opt ori (la un singur dreptunghi sau pătrat) astfel încît trecînd cu un ac prin cele opt pliuri suprapuse acesta să le perforeze exact în ordinea 1, 2, 3, ..., 8 ? Încercaţi aceste două configuraţii:



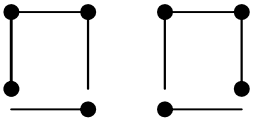
15. Problemă pentru cei puternici. Încercaţi să împăturiţi de 8 ori, pur şi simplu, o coală de hîrtie (de fiecare dată linia de îndoire este "în cruce" peste cea dinainte). Este posibil ? (!)Determinaţi ce dimensiuni ar trebui să aibă foaia la început pentru a putea fi împăturită de 8 ori.

16. Este posibil ca un cal să treacă prin toate cele 64 de pătrăţele ale unei table de şah, începînd dintr-un colţ şi terminînd în colţul diagonal opus ?

17. Într-un atelier există 10 lădiţe ce conţin fiecare piese cu greutatea de 100 grame, cu excepţia uneia din lădiţe ce conţine piese avînd grutatea de 90 grame. Puteţi preciza care este lădiţa cu pricina, folosind un cîntar doar pentru o singură dată ?

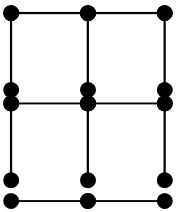
Probleme cu chibrituri

1. (!) Eliminînd un singur băţ de chibrit ceea ce rămîne în faţa ochilor este un elipsoid!

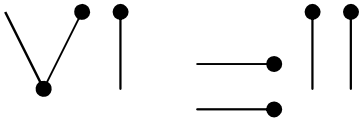


2. **(!) 9 beţe.** Să se aşeze 9 beţe de chibrit astfel încît ele să se întîlnească la vîrf tot cîte trei în şase vîrfuri distincte.

3. De la 4 la 3. În figura ce conţine 4 pătrate, mutînd 4 beţe să se obţină o figură ce conţine doar 3 pătrate.



4. **6 = 2 ?** Mutînd doar un singur băţ de chibrit să se restabilească egalitatea:



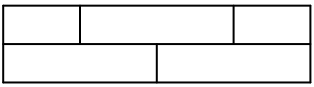
5. **Problema ariilor întregi.** Puteţi aşeza 12 chibrituri astfel încît ele să formeze contururile unor poligoane ce au aria întreagă egală cu 5, (!!) 4, 3, 2, (!!!) 1 ? Se subînţelege că un chibrit poate fi asimilat cu un segment de lungime 1 şi că nu există nici o dificultate de a forma "din ochi" unghiuri drepte.

Probleme de logică şi judecată

1. Substituirea literelor. Subtituiţi literele cu cifre astfel încît următoarele adunări să fie corecte: GERALD + DONALD = ROBERT ; FORTY + TEN + TEN = SIXTY ; BALON + OVAL = RUGBY.

2. Test de angajare la Microsoft. Patru excursionişti ajung pe malul unui rîu pe care doresc să-l traverseze. Întrucît s-a înoptat şi ei dispun doar de o singură lanternă, ei pot să treacă rîul cel mult cîte doi laolaltă. Ştiind că, datorită diferenţelor de vîrstă şi datorită oboselii, ei ar avea individual nevoie pentru a traversa rîul de 1, 2, 8 şi 10 minute, se cere să se decidă dacă este posibilă traversarea rîului în aceste conditţii în doar 17 minute ?

3. (!) Imposibilă. Să se taie toate cele 16 segmente ale figurii următoare cu o singură linie curbă continuă şi care nu se intersectează cu ea însăşi.



4. (!) Problema "ochilor albaştri". Sîntem martorii următorului dialog între două persoane X şi Y. << **X:** Eu am trei copii. Produsul vîrstei lor este 36 iar suma vîrstei lor este egală cu numărul de etaje al blocului din vecini de mine. Îl ştii, nu-i aşa ? **Y:** Desigur. Dar numai din cît mi-ai spsus nu pot să deduc care

este vîrsta copiilor tăi. **X:** Bine, atunci află că cel mare are ochi albaştrii.>> Puteţi afla care este vîrsta celor trei copii ?

5. Problema călugărului budhist. Într-o dimineaţă, exact la răsăritul soarelui, un călugăr budhist porneşte de la templul de la baza muntelui pentru a ajunge la templul din vîrful muntelui exact la apusul soarelui, unde el se roagă toată noaptea. A doua zi el porneşte din vîrf pe aceeşi cărare, tot la răsăritul soarelui, pentru a ajunge la templul de la baza muntelui exact la apusul soarelui. Să se arate că a existat un loc pe traseu în care călugărul s-a aflat în ambele zile exact la aceaşi oră.

6. Vinul în apă şi apa în vin. Dintr-o sticlă ce conţine un litru de apă este luat un pahar (un decilitru) ce este turnat pest un litru de vin. Vinul cu apa se amestecă bine după care se ia cu acelaşi pahar o cantitate egală de "vin cu apă" ce se toarnă înapoi peste apa din sticlă. Avem acum mai multă apă în vin decît vin în apă, sau invers ?

7. (!!!!) Cuiele în echilibru. Avem la dispoziţie 7 cuie normale, cu capul obişnuit. Înfigem unul vertical în podea (sau într-o placă de lemn). Se cere să se aşeze cele 6 cuie rămase în **echilibru stabil** pe capul cuiului vertical, fără ca niciunul din cele şase cuie să atingă podeaua.

8. (!!) Ţigările tangente. Este posibil să aşezăm pe masă şase ţigări astfel încît fiecare să se atingă cu fiecare (oricare două să fie tangente) ? **(!!!)** Dar şapte ţigări ?

9. (!) Problema celor 12 înţelepţi (în variantă modernă). Managerul unei mari companii doreşte să pună la încercare inteligenţa şi puterea de judecată a celor 12 membrii ai consiliului său de conducere. Luînd 12 cărţi de joc, unele de pică şi altele de caro, el le aşează cîte una pe fruntea fiecărui consilier astfel încît fiecare să poată vedea cărţile de pe frunţile celorlalţi

dar nu şi pe a sa. Managerul le cere celor care consideră că au pe frunte o carte de caro (diamond) să facă un pas în faţă, altfel ei nu vor mai putea face parte din consiliu. După ce îşi repetă cererea de şapte ori, timp în care niciunul din cei 12 consilieri nu face nici o mişcare (ci doar se privesc unii pe alţii), toţi consilierii care au într-adevăr pe frunte o carte de caro ies deodată în faţă. Puteţi deduce cîţi au ieşit şi cum şi-au dat ei seama ce carte este aşezată pe fruntea lor ?

10. Păianjenul şi musca. Pe peretele lateral al unei hale cu dimensiunile de 40 x 12 x12 metri, pe linia mediană a peretelui lateral şi exact la 1 metru de tavan, se află un păianjen. Pe peretele lateral opus, tot pe linia mediană şi exact la 1 metru de podea, se află o muscă amorţită. Care este distanţa cea mai scurtă pe care păianjenul o are de parcurs de-a lungul pereţilor pentru a se înfrupta din muscă ?

11. Rifi şi Ruf. Cei doi iubiţi Rifi şi Ruf, din nordica ţară Ufu-Rufu, locuiesc în sate diferite aflate la distanţa de 20 km unul de altul. În fiecare dimineaţă ei pornesc exact deodată (la răsărit) unul spre celălalt spre a se întîlni şi a se săruta confrom obiceiului nordic: nas în nas. Într-o dimineaţă o muscă rătăcită porneşte exact la răsăritul soarelui de pe nasul lui Rifi direct spre nasul lui Ruf, care o alungă trimiţînd-o din nou spre nasul lui Rifi, ş.a.m.d. ..., pînă cînd ea sfîrşeşte tragic în momentul "sărutului" celor doi. Ştiind că Rifi se deplasează cu 4 km/oră, Ruf cu 6 km/oră iar musca zboară cu 10 km/oră, se cere să se afle ce distanţă a parcurs musca în zbor de la răsărit şi pînă în momentul tragicului ei sfîrşit.

12. ***O anti-problemă de şah.*** *În următoarea configuraţie a pieselor pe o tablă de şah se cere să nu daţi mat dintr-o mutare ! (Albul atacă de jos în sus. Legenda: P-pion, N-nebun, R-rege, T-turn, C-cal. Alăturat fiecărei piese este scrisă culoarea sa, alb-a sau negru-n.)*

NN a						RR a	TT a
	TT n						NN a
						TT a	
		NN n		PP n		PP n	
		PP a		RR n		PP a	
	PP n			PP a			PP n
	PP a			PP a			PP a
			CC a		CC a		

13. ***Bronx contra Brooklyn.*** *Un tînăr, ce locuieşte în Manhattan în imediata apropiere a unei staţii de metrou, are două prietene, una în Brooklyn şi cealaltă în Bronx. Pentru a o vizita pe cea din Brooklyn el ia metroul ce merge spre partea de jos a oraşului, în timp ce, pentru a o vizita pe cea din Bronx, el ia din acelaşi loc metroul care merge în direcţie opusă. Metrourile spre Brooklyn şi spre Bronx intră în staţie cu aceeşi frecvenţă: din 10 în 10 minute fiecare. Dar, deşi el coboară în staţia de metrou în fiecare sîmbătă la întîmplare şi ia primul metrou care vine (nedorind să "favorizeze" pe nici una din prietenele sale), el a constatat că, în medie, el merge în Brooklyn de 9 ori din 10. Puteţi găsi o explicaţie logică a fenomenul ?*

14. ***(!!) Problema celor 12 bile.*** *În faţa noastră se află 12 bile identice ca formă, vopsite la fel, dar una este cu siguranţă falsă, ea fiind fie mai grea, fie mai uşoară, fiind făcută dintr-un alt*

material. Avem la dispoziţie o balanţă şi se cere să determinăm doar prin 3 cîntăriri care din cele 12 bile este falsă precizînd şi cum este ea: mai grea sau mai uşoară. ***(!!!) Mai mult****, puteţi determina care este numărul maxim de bile din care prin 4 cîntăriri cu balanţa se poate afla exact bila falsă şi cum este ea ?*

15. (!) Problema celor 2 perechi de mănuşi. *Aflat într-o situaţie ce implică intervenţia de urgenţă, un medic chirurg constată că are la dispoziţie doar 2 perechi de mănuşi sterile deşi el trebuie să intervină rapid şi să opereze succesiv 3 bolnavi. Este posibil ca cele trei operaţii de urgenţă să se desfăşoare în condiţii de protecţie normale cu numai cele 2 perechi de mănuşi ? (Sîngele fiecăruia din cei 3 pacienţi, precum şi mîna doctorului nu trebuie să conducă la un contact infecţios.)*

16. (!!) Problema frînghiei prea scurte. *O persoană ce are asupra ei doar un briceag şi o frînghie lungă de 30 metri se află pe marginea unei stînci, privind în jos la peretele vertical de 40 metri aflat sub ea. Frînghia poate fi legată doar în vîrf sau la jumătatea peretelui (la o înălţime de 20 metri de sol) unde se află o mică platformă de sprijin. Cum este posibil ca persoana aflată în această situaţie să ajungă teafără jos coborînd numai pe frînghie, fără a fi nevoită să sară deloc punîndu-se astfel în pericol ?*

17. Problema lumînărilor neomogene. *Avem la dispoziţie chibrite şi două lumînări care pot arde exact 60 minute fiecare însă, ele fiind neomogene, nu vor arde cu o viteză constantă. Cum putem măsura precis o durată de 45 minute ?*

18. (!!) O jumătate de litru. *Avem în faţa noastră un vas cilindric cu capacitatea de 1 litru, plin ochi cu apă. Se cere să măsurăm cu ajutorul lui ½ litru de apă, fără a ne ajuta de nimic altceva decît de mîinile noastre.*

Probleme de logică şi judecată cu "tentă informatică"

1. **(!!!) Decriptarea scrierii încifrate.** Se dau următoarele numere împreună cu denumirile lor cifrate:

5 nabivogedu
6 nagevogedu
10 nabivobinaduvogedu
15 nabivonagevogedunaduvogedu
20 nabivogenagevogenaduvogedu
25 nabivonabivobinagevogedunagevogenaduvogedu
30 nabivodunanabivobiduvogedu
50 nabivonabivonabivogedunagevogenaduvogedunanabivobiduvogedu
60 nabivonagevogedunagevogenanabivobiduvogedu
90 nabivonaduvogedunagevodunanabivobiduvogedu
100 nabivonabivobinagevogenaduvogedunagevodunanabivobiduvogedu

Care este regula de încifrare? Ce numere reprezintă următoarele coduri cifrate:

nagevonagevogedunanabivobiduvogedu;

nagevonaduvogedunanabivobiduvogedu;

naduvogenanabivobiduvogedu;

nanabivogeduvogedu;

nabivonabivonaduvogedunagevonagevogedunanabivobiduvogedu;

nanagevobiduvogedu?

Încifraţi numerele 256 şi 1024 prin acestă metodă.

2. (!!!) Altfel de codificare binară a numerelor. *Descoperiţi metoda de codificare binară a numerelor folosită în continuare:*

1	1	20	101010
2	10	25	1000101
3	11	30	1010001
5	110	40	10001001
10	1110	50	10100100
15	10010	60	100001000

Puteţi spune ce numere sînt codificate prin 100, 101, 1000, 1111, 10000 şi 11111 ? Puteţi codifica numerele 70, 80, 90, 100, 120, 150 şi 1000 ?

3. **(!!!) Problema dialogului perplex.** Există două numere m şi n din intervalul [2..99] şi două persoane P şi S astfel încît persoana P ştie produsul lor, iar S ştie suma lor. Ştiind că între P şi S a avut loc următorul dialog:

"Nu ştiu numerele" spune P.

"Ştiam ca nu ştii" răspunde S, "nici eu nu ştiu."

"Acuma ştiu !" zice P strălucind de bucurie.

"Acum ştiu şi eu..." şopteşte satisfăcut S.

să se determine toate perechile de numere m şi n ce "satisfac" acest dialog (sînt soluţii ale problemei).

4. **(!!!!) Împăturirea celor 8 pătrate.** Împăturiţi iniţial în opt o foaie dreptunghiulară după care desfaceţi-o şi însemnaţi fiecare pătrăţel obţinut cu o cifră de la 1 la 8. Proiectaţi un algoritm şi realizaţi un program care, primind configuraţia (numerotarea) celor 8 pătrăţele, să poată decide dacă se poate împături foaia astfel obţinută reducînd-o de opt ori (la un singur pătrat) astfel încît trecînd cu un ac prin cele opt foi suprapuse acesta să le perforeze exact în ordinea 1, 2, 3, ..., 8.

5. **(!!!!) Problema fetelor de la pension.** Problema a apărut pe vremea cînd fetele învăţau la pension fără ca prin prezenţa lor băieţii să le tulbure educaţia. Pedagoaga fetelor unui pension de 15 fete a hotarît ca în fiecare dupa-amiază, la ora de plimbare, fetele să se plimbe în cinci grupuri de cîte trei. Se cere să se stabilească o programare a plimbărilor pe durata unei săptămîni (şapte zile) astfel încît fiecare fată să ajungă să se plimbe numai o singură dată cu oricare din celelalte paisprezece (oricare două fete să nu se plimbe de două ori împreună în decursul unei săptămîni).

Noţiuni fundamentale de programare

Programarea este disciplina informatică ce are ca scop realizarea de programe care să constituie soluţiile oferite cu ajutorul calculatorului unor probleme concrete. Programatorii sînt acele persoane capabile să implementeze într-un limbaj de programare metoda sau algoritmul propus ca soluţie respectivei probleme, ce se pretează a fi soluţionată cu ajutorul calculatorului. După nivelul de implicare în efortul de rezolvare a problemelor specialiştii în programare pot fi împărţiţi în diverse categorii: analişti, analişti-programatori, ingineri-programatori, simpli programatori, etc. Cu toţii au însă în comun faptul că fiecare trebuie să cunoască cît mai bine programare şi să fie capabil, nu doar să citească, ci chiar să scrie "codul sursă", adică programul propriu-zis. Din acest punct de vedere cunoştinţele de programare sînt considerate "ABC-ul" informaticii şi sînt indispensabile oricărui profesionist în domeniu.

1.Cele trei etape ale rezolvării unei probleme cu ajutorul calculatorului

În rezolvarea sa cu ajutorul calculatorului orice problemă trece prin trei etape obligatorii: *Analiza problemei, Proiectarea algoritmului de soluţionare* şi *Implementarea algoritmului într-un program pe calculator.* În ultima etapă, sub acelaşi nume, au fost incluse în plus două subetape cunoscute sub numele de *Testarea* şi *Întreţinerea programului.* Aceste subetape nu lipsesc din "ciclul de viaţă" a oricărui produs-program ce "se respectă" dar , pentru simplificare, în continuare ne vom referi doar la primele trei mari etape.

Dacă etapa *implementării* algoritmului într-un program executabil este o etapă exclusiv practică, realizată "în faţa calculatorului", celelalte două etape au un pronunţat caracter teoretic. În consecinţă, primele două etape sînt caracterizate de

un anumit grad de abstractizare. Din punct de vedere practic însă, şi în ultimă instanţă, criteriul decisiv ce conferă succesul rezolvării problemei este dat de calitatea implementării propriuzise. Mai exact, succesul soluţionării este dat de performanţele programului: utilitate, viteză de execuţie, fiabilitate, posibilităţi de dezvoltare ulterioare, lizibilitate, etc. Cu toate acestea este imatură şi neprofesională "strategia" programatorilor începători care, neglijînd primele două etape, sar direct la a treia fugind de analiză şi de componenta abstractă a efortului de soluţionare. Ei se justifică cu toţii prin expresii puerile de genul: "Eu nu vreau să mai pierd vremea cu "teoria", am să fac programul cum ştiu eu. Cîtă vreme nu va face altcineva altul mai bun decît al meu, nu am de ce să-mi mai bat capul !".

2.Cum se stabileşte corectitudinea şi eficienţa soluţionării ?

Este adevărat că ultima etapă în rezolvarea unei probleme – implementarea – este decisivă şi doveditoare, dar primele două etape au o importanţă capitală. Ele sînt singurele ce pot oferi răspunsuri corecte la următoarele întrebări dificile: *Avem certitudinea că soluţia găsită este corectă ? Avem certitudinea că problema este complet rezolvată ? Cît de eficientă este soluţia găsită ? Cît de departe este soluţia aleasă de o soluţie optimă ?*

Să menţionăm în plus că literatura informatică de specialitate conţine un număr impresionant de probleme "capcană" pentru începători, şi nu numai pentru ei. Ele provin majoritatea din realitatea imediată dar pentru fiecare dintre ele nu se cunosc soluţii eficiente. De exemplu, este dovedit teoretic că problema, "aparent banală" pentru un calculator, a proiectării *Orarului optim* într-o instituţie de învăţămînt (şcoală, liceu, facultate) este o problemă intratabilă la ora actuală (toate programele care s-au realizat pînă acum nu oferă decît soluţii aproximative fără a putea spune cît de aproape sau de departe este soluţia optimă de orar).

Cîţi dintre programatorii începători n-ar fi surprinşi să afle că problema "atît de simplă" (ca enunţ), a cărei soluţionare tocmai au abandonat-o, este de fapt o problemă dovedită teoretic ca fiind intratabilă sau chiar insolvabilă algoritmic ? Partea proastă a lucrurilor este că, aşa cum ciupercile otrăvite nu pot fi cu uşurinţă deosebite de cele comestibile, tot astfel problemele netratabile pot fi cu uşurinţă confundate cu nişte probleme uşoare la o privire rapidă şi lipsită de experienţă.

Dacă ar fi să sintetizăm în cîte un cuvînt efortul asupra căruia se concentrează fiecare din cele trei etape – *analiza, proiectarea* şi *implementarea*– cele trei cuvinte ar fi: *corectitudine, eficienţă* şi *impecabilitate*. Etapa de analiză este singura care permite *dovedirea cu argumente riguroase* a corectitudinii soluţiei, iar etapa de proiectare este singura care poate *oferi argumente precise* în favoarea eficienţei soluţiei propuse.

În general problemele concrete din informatică au în forma lor iniţială sau în enunţ o caracteristică pragmatică, fiind foarte ancorate în realitatea imediată. Totuşi ele conţin în formularea lor iniţială un grad mare de *eterogenitate*, *diversitate* şi *lipsă de rigoare.* Fiecare dintre aceste "defecte" este un obstacol major pentru demonstrarea corectitudinii soluţiei. Rolul esenţial al etapei de analiză este acela de a transfera problema "de pe nisipurile mişcătoare" ale realităţii imediate de unde ea provine într-un plan abstract, adică de *a o modela*. Acest "univers paralel abstract" este dotat cu mai multă rigoare şi disciplină internă, avînd legi precise, şi poate oferi instrumentele logice şi formale necesare pentru demonstrarea riguroasă a corectitudinii soluţiei problemei. Planul abstract în care sînt "transportate" toate problemele de informatică este planul sau *universul obiectelor matematice* iar corespondentul problemei în acest plan va fi *modelul matematic abstract* asociat problemei. Demonstrarea corectitudinii proprietăţilor ce leagă obiectele universului matematic a fost şi este sarcina matematicienilor. Celui ce analizează problema din punct de vedere informatic îi

revine sarcina (nu tocmai uşoară) de a dovedi printr-o **demonstraţie constructivă** că există o *corespondenţă precisă (o bijecţie !)* între părţile componente ale problemei reale, "dezasamblată" în timpul analizei, şi părţile componente ale modelului abstract asociat. Odată descoperită, formulată precis şi dovedită, această "perfectă oglindire" a problemei reale în planul obiectelor matematice oferă certitudinea că toate proprietăţile şi legăturile ce există între subansamblele modelului abstract se vor regăsii precis (prin reflectare) între părţile interne ale problemei reale, şi invers. Atunci, soluţiei abstracte descoperite cu ajutorul modelului matematic abstract îi va corespunde o soluţie reală concretizată printr-un algoritm ce poate fi implementat într-un program executabil.

Aceasta este calea generală de rezolvare a problemelor şi oricine poate verifica acest fapt. De exemplu, ca şi exerciţiu, încercaţi să demonstraţi corectitudinea (adică să se aducă argumente precise, clare şi convingătoare în favoarea *corectitudinii*) metodei de extragere a radicalului învăţată încă din şcoala primară (cu grupare cifrelor numărului în grupuri cîte două, etc…) sau a algoritmului lui Euclid de determinare a celui mai mare divizor comun a două numere prin împărţiri întregi repetate. Desigur nu pot fi acceptate argumente copilăreşti de forma: "Algoritmul este corect pentru că aşa l-am învăţat!" sau "Este corect pentru că aşa face toată lumea !" din moment ce nu se oferă o argumentaţie matematică riguroasă.

Ideea centrală a etapei a doua – proiectarea unui algoritm de soluţionare eficient poate fi formulată astfel: din studiul *proprietăţilor* şi *limitelor* modelului matematic abstract asociat problemei se deduc *limitele inferioare ale complexităţii minimale ("efortului minimal obligatoriu")* inerente **oricărui** algoritm ce va soluţiona problema în cauză. *Complexitatea internă a modelului abstract* şi *complexitatea soluţiei abstracte* se va reflecta imediat asupra *complexităţii reale a algoritmului*, adică asupra *eficienţei* de soluţionare a problemei. Acest fapt permite *prognosticarea* încă din această fază – faza de proiectare

a algoritmului de soluţionare – a eficienţei practice, măsurabilă ca *durată de execuţie*, a programului.

3. Noţiunile fundamentale ale programării: algoritm, limbaje de descriere a algoritmilor, program, limbaje de programare

3.1. Algoritmul

Se ştie că la baza oricărui program stă un algoritm (care, uneori, este numit *metodă de rezolvare*). Noţiunea de *algoritm* este o noţiune fundamentală în informatică şi înţelegerea ei, alături de înţelegerea modului de funcţionare a unui calculator, permite înţelegerea noţiunii de *program* executabil. Vom oferi în continuare o definiţie unanim acceptată pentru noţiunea de *algoritm*:

Definiţie. Prin algoritm se înţelege o *mulţime finită* de operaţii (instrucţiuni) elementare care executate într-o *ordine bine stabilită (determinată)*, pornind de la un set de date de intrare dintr-un domeniu de valori posibile (valide), produce în *timp finit* un set de date de ieşire (rezultate).

Cele trei caracteristici esenţiale ale unui algoritm sînt:

Determinismul – dat de faptul că ordinea de execuţie a instrucţiunilor algoritmului este bine precizată (strict determinată).

Acest fapt dă una din calităţile de bază a calculatorului: "el" va face întotdeauna ceea ce i s-a cerut (prin program) să facă, "el" nu va avea iniţiative sau opţiuni proprii, "el" nu-şi permite să greşească nici măcar odată, "el" nu se va plictisi ci va duce programul la acelaşi sfîrşit indiferent de cîte ori i se va cere să repete acest lucru. Nu aceeaşi situaţie se întîmplă cu fiinţele umane (*Errare humanum est).* Oamenii pot avea în situaţii determinate un comportament non-deterministic (surprinzător). Acesta este motivul pentru care numeroşi utilizatori de calculatoare (de exemplu contabilii), datorită fenomenului de personificare a calculatorului (confundarea acţiunilor şi

dialogului "simulat" de programul ce rulează pe calculator cu reacţiile unei personalităţi vii), nu recunosc perfectul determinism ce stă la baza executării oricărui program pe calculator. Exprimîndu-se prin propoziţii de felul: "*De trei ori i-am dat să facă calculele şi de fiecare dată mi-a scos aceleaşi valori aiurea!*" ei îşi trădează propria viziune personificatoare asupra unui fenomen determinist.

Universalitatea – dată de faptul că, privind algoritmul ca pe o metodă automată (mecanică) de rezolvare, această metodă are un caracter general-universal. Algoritmul nu oferă o soluţie punctuală, pentru un singur set de date de intrare, ci oferă soluţie pentru o mulţime foarte largă (de cele mai multe ori infinită) de date de intrare valide. Aceasta este trăsătura de bază care explică deosebita utilitate a calculatoarelor şi datorită acestei trăsături sîntem siguri că investiţia financiară făcută prin cumpărarea unui calculator şi a produsului-soft necesar va putea fi cu siguranţă amortizată. Cheltuiala se face o singură dată în timp ce programul pe calculator va putea fi executat rapid şi economicos de un număr oricît de mare de ori, pe date diferite !

De exemplu, metoda (algoritmul) de rezolvare învăţată la liceu a ecuaţiilor de gradul doi: $ax^2+bx+c=0$, se aplică cu succes pentru o mulţime infinită de date de intrare: $(a,b,c)\in\Re\setminus\{0\}x\Re x\Re$.

Finitudinea – pentru fiecare intrare validă orice algoritm trebuie să conducă în timp finit (după un număr finit de paşi) la un rezultat. Această caracteristică este analogă proprietăţii de convergenţă a unor metode din matematică: trebuie să avem garanţia, dinainte de a aplica metoda (algoritmul), că metoda se termină cu succes (ea converge către soluţie).

Să observăm şi diferenţa: în timp ce metoda matematică este corectă chiar dacă ea converge către soluţie doar *la infinit* (!), un algoritm trebuie să întoarcă rezultatul după un număr finit de paşi. Să observăm deasemenea că, acolo unde matematica nu oferă dovada, algoritmul nu va fi capabil să o ofere nici el. De exemplu, nu este greu de scris un algoritm care să verifice corectitudinea *Conjecturii lui Goldbach: "Orice număr par se scrie*

ca sumă de două numere prime", dar, deşi programul rezultat poate fi lăsat să ruleze pînă la valori extrem de mari, fără să apară nici un contra-exemplu, totuşi conjectura nu poate fi astfel infirmată (dar nici afirmată!).

4. Secretul învăţării rapide a programării

Există posibilitatea învăţării rapide a programării ?

Desigur. Experienţa predării şi învăţării programării ne-a dovedit că există metode diferite de învăţare a programării, mai rapide sau mai lente, mai temeinice sau mai superficiale. Din moment ce se doreşte învăţarea rapidă a programării înseamnă că, pentru cel ce doreşte aceasta, problemele ce îşi aşteaptă rezolvarea cu ajutorul calculatorului sînt importante sau stringente. Am putea chiar presupune că soluţionarea lor rapidă este un deziderat mai important decît învăţarea programării. Tocmai de aceea, fiind conştienţi de acest fapt, vom prezenta în continuare una din cele mai rapide metode de învăţare a programării.

Să observăm mai întîi că pentru învăţarea unei limbi străine este necesară comunicarea şi vorbirea intensă a acelei limbi. Cu toţii am putut constata că dacă există o motivaţie sau nevoie puternică de a comunica în acea limbă, cel puţin pentru o perioadă de timp, procesul de învăţare a ei este foarte rapid. De exemplu, dacă ne aflăm într-o ţară străină sau dacă dorim apropierea de o persoană străină (mai ales dacă este atrăgătoare şi de sex opus…) categoric vom constata că am învăţat mult mai iute limba respectivă. Şi aceasta datorită faptului că efortul de învăţare a fost mascat în spatele efortului (intens motivat!) de a comunica şi de a ne face cunoscute intenţiile şi gîndurile.

La fel, pentru învăţarea rapidă şi cu uşurinţă a programării efortul trebuie îndreptat, nu spre "silabisirea" limbajului de programare, ci spre rezolvarea de probleme şi spre scrierea directă a programelor de soluţionare a acestora. Concentrîndu-ne asupra problemelor ce le soluţionăm nici nu

vom observa cînd şi în ce fel am învăţat să scriem programe. La urma urmei, programarea este doar un instrument, doar o unealtă "de scris", şi nu un scop în sine. Dacă vrei iute să înveţi să scrii, contează cum sau în ce mînă ţii stiloul ?...

Nu trebuie deloc neglijat şi un al doilea "factor secret". Aşa cum "*meseria nu se învaţă, ci se fură*", tot astfel programarea se poate învăţa mult mai uşor apelînd la ajutorul unui profesor sau a unui specialist. Acesta, prin experienţa şi cunoştinţele sale de specialitate ne poate ajuta să păşim alături de el "pe cărări bătătorite" şi într-un ritm susţinut.

În concluzie, într-o descriere plastică şi metaforică, metoda secretă cea mai rapidă de "ascensiune" în programare este metoda *"privirii concentrate spre vîrf, cu ghidul alături şi pe cărări bătătorite"*.

Noţiuni primare de programare în Pascal şi C

În spiritul celor spuse mai sus, vom introduce acum "într-un ritm alert", prin exemple concrete, noţiunile elementare de programare în limbajele Pascal şi C (în paralel). Vom pleca de la prezentarea structurii generale a unui program iar apoi vom trece la prezentarea celor patru structuri-instrucţiuni elementare conţinute în psedo-limbajul de descriere a algoritmilor. Vom avea în plus grijă de a precede descrierea fiecărei structuri elementare de liniile de declarare a tipului variabilelor implicate. Peste tot vor apare linii de comentariu (ignorate de compilator). În limbajul Pascal comentariile sînt cuprinse între acolade {*comentariu*}, pe cînd în C ele sînt cuprinde între construcţia de tipul /* *comentariu**/ sau apar la sfîrşitul liniei precedate de două slash-uri //*comentariu*.

Structura unui program	
Program Nume_de_Program; {această linie poate să lipsească}	// linii de incluziuni de fişiere header

{ Zona de declaraţii constante, variabile, proceduri şi funcţii }
BEGIN
{ Corpul programului format din instrucţiuni terminate cu punct-vigulă ; Corpul programului poate fi privit ca o instrucţiune compusă }
END.
(Orice se va scrie după punct va fi ignorat de către compilator)
	// declaraţii de variabile şi funcţii externe (globale)
	
	void main(void){
	// declaraţii de variabile locale
	
	// corpul programului format din instrucţiuni terminate cu punct-vigulă ;
	}

Exemplu :
Program Un_Simplu_Test;
Const e=2.68;
Var x:real;
BEGIN
x:=1./2+e*(1+e);
Writeln('Rezultatul este:',x);
END.
	Exemplu :
	#include <stdio.h>
	
	int e=2.68;
	float x;
	void main(void){
	x=1./2+e*(1+e);
	printf("Rezultatul este %f:",x);
	}

Atribuirea : var:=expresie;	

Var i,j:integer;perimetrul:real;
……..
j:=2000 div 15; { împărţire întreagă obligatorie }
i:=i+(j-1)*Sqr(2*j+1); { Sqr (Square) – funcţia de ridicare la pătrat }
perimetrul:=2*PI*i; { PI – constantă reală implicită }
	#include <math.h> // declară constanta M_PI
	int i,j; float perimetrul;
	……..
	j=2000 / 15; // împărţire întreagă implicită !!
	i+=(j-1)* (2*j+1)*(2*j+1); // în C avem operatorul
	// de adunare + înainte de egal = ; funcţia putere în
	// C este pow(x,y)
	perimetrul=2*M_PI*i;

Intrare/Ieşire : **Citeşte var_1, var_2, var_3, ...;** **Scrie var_1, var_2, var_3, ...;** **Sau** **Scrie $expresia_1$, $expresia_2$, $expresia_3$,...;**	

Var i,j:integer;perimetrul:real;
……..
Readln(i,j); { citirea variabilelor i şi j }
Perimetrul:=2*PI*i;
Writeln('Raza=',i:4,' Perimetrul=',perimetrul:6:2,' Aria=', PI*Sqr(i):6:2);
{ perimetrul si aria fiind valori reale, se afiseaza cu descriptorul de format de afisare :6:2 – pe 6 poziţii de ecran cu rotunjit la 2 zecimale
}
	#include <math.h> // declară constanta M_PI
	int i,j; float perimetrul;
	……..
	scanf("%i %i",&i,&j); // "%i %i" este descriptorul de format de citire, & este operatorul de adresare
	perimetrul=2*M_PI *i;
	printf("Raza=%4i Perimetrul= %6.2f Aria= %6.2f",i,perimetrul,M_PI*i*i); // %6.2f – descriptorul de format de afisare a unei valori reale(flotante) pe 6 poziţii rotunjit la 2 zecimale

Condiţionala : **Dacă <condiţie_logică> atunci $instrucţiune_1$ [altfel $instrucţiune_2$];**	

Var i,j,suma:integer;
……..
If i <= 2*j+1 then
suma:=suma+i
else suma:=suma+j;
	int i,j,suma;
	……..
	if (i<=2*j+1) suma+=i
	else suma+=j;

Ciclul de tipul Repeat-Until: **Repetă $instrucţiune_1$, $instrucţiune_2$, ... pînă cînd <condiţie_logică>;**	
Var i,j,suma:integer; …….. suma:=0;i:=1;	int i,j,suma; …….. suma=0;i=1;

Repeat
  suma:=suma+i; i:=i+1;
Until i>100;
	do
	  suma+=i;
	while (i++<100);

Ciclul de tipul Do-While: **Cît timp <condiţie_logică> execută instrucţiune;**	

Var i,j,suma:integer;
……..
suma:=0;i:=1;
While i<=100 do begin
  suma:=suma+i; i:=i+1;
End;
	int i,j,suma;
	……..
	suma=0;i=1;
	while (i++<100)
	  suma+=i;

Ciclul de tipul For (cu contor): **Pentru var_contor:=val_iniţială pînă la val_finală execută instrucţiune;**	

Var i,j,suma:integer;
……..
suma:=0;
For i:=1 to 100 do
  Suma:=suma+i;
	int i,j, suma;
	……..
	for(suma=0,i=1;i<=100;i++)
	  suma+=i;

Exemple de probleme rezolvate

Prezentăm în continuare, spre iniţiere, cîteva exemple de probleme rezolvate. Vom oferi programul rezultat atît în limbajul de programare Pascal cît şi în limbajul C. Deasemenea, fiecare program va fi precedat de o scurtă descriere a modului de elaborare a soluţiei.

1. Se citesc *a,b,c* coeficienţii reali a unei ecuaţii de gradul II. Să se afişeze soluţile ecuaţiei.

Descrierea algoritmului:

- ecuaţia de gradul II este de forma $ax^2+bx+c=0$

-presupunînd că $a \neq 0$ calculăm determinantul ecuatiei delta=b*b-4*a*c

- dacă delta >= 0 atunci ecuaţia are soluţiile reale $x_{1,2}=(-b\pm\sqrt{delta})/(2*a)$

- dacă delta < 0 atunci ecuaţia are soluţiile complexe $z_1=(-b/(2*a), \sqrt{(-delta)}/(2*a))$, $z_1=(-b/(2*a), -\sqrt{(-delta)}/(2*a))$

```
Program Ecuatie_grad_2;  { varianta Pascal }
Var a,b,c,delta:real;

BEGIN
Write('Introd. a,b,c:');Readln(a,b,c);
delta:=b*b-4*a*c;
If  delta>=0 then
     Begin
      Writeln('x1=',(-b-sqrt(delta))/(2*a):6:2);
      Writeln('x2=',(-b+sqrt(delta))/(2*a):6:2);
     End
   else Begin
Writeln('z1=(',-b/(2*a):6:2, ',' , -sqrt(-delta))/(2*a):6:2, ')');
Writeln('z2=(', -b/(2*a):6:2, ',' , sqrt(-delta))/(2*a):6:2, ')');
End
Readln;
END.
```

// versiunea C

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

float a,b,c;  // coeficientii ecuatiei de gradul II
      float delta;

      void main(){
      printf("Introd.coefic.ecuatiei a b c:");scanf("%f %f
%f",&a,&b,&c);
      delta=b*b-4*a*c;
      if (delta>=0) {
```

```
        printf ("Sol.reale: x1=%6.2f, x2=%6.2f",(-
b+sqrt(delta))/2./a,(-b-sqrt(delta))/2./a);
    } else {
      printf("Sol.complexe: x1=(%6.2f,%6.2f),
x2=(%6.2f,%6.2f)",-b/2./a,sqrt(-delta)/2./a,-b/2/a,-sqrt(-
delta)/2./a);
    }
    }
```

2. Să se determine dacă trei numere *a,b,c* reale pot reprezenta laturile unui triunghi. Dacă da, să se caculeze perimetrul şi aria sa.

Descrierea algoritmului:

- condiţia necesară pentru ca trei numere să poată fi lungimile laturilor unui triunghi este ca cele trei numere să fie pozitive (condiţie implicită) şi suma a oricăror două dintre ele să fie mai mare decît cel de-al treilea număr

- după condiţia este îndeplinită vom calcula perimetrul şi aria triunghiului folosind **formula lui Heron** s=sqrt(p(p-a)(p-b)(p-c)) unde p=(a+b+c)/2.

```
Program Laturile_Unui_Triunghi; { Pascal }
Var a,b,c,s,p:real;

function laturi_ok:boolean;
begin
  laturi_ok:= (a>0) and (b>0) and (c>0) and  (a+b>c) and
(a+c>b) and (b+c>a) ;
end;

BEGIN
write('introduceti laturile');readln(a,b,c);
IF laturi_ok then
begin
   p:=(a+b+c)/2;
```

```
    s:=sqrt(p*(p-a)*(p-b)*(p-c));
    writeln('Aria=',s:5:2);
    writeln('Perimetrul=',2*p:5:2);
  end
  else writeln('Nu formeaza triunghi');
  readln;
  END.
```

// versiunea C

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

float a,b,c,s,p;

int validare_laturi(float a,float b,float c){
  return(
(a>0)&&(b>0)&&(c>0)&&(a+b>c)&&(b+c>a)&&(a+c>b));
}

void main(void){
  printf("Introd.laturile a b c:");scanf("%f %f
%f",&a,&b,&c);
  if (validare_laturi(a,b,c)){
    p=(a+b+c)/2;s=sqrt(p*(p-a)*(p-b)*(p-c));
    printf("Aria=%6.2f, Perimetrul=%6.2f",s,2*p);
  }
}
```

3. Se citeşte *n* întreg. Să se determine suma primelor *n* numere naturale.

Descrierea algoritmului:

- vom oferi varianta în care suma primelor n numere naturale va fi calculata cu una dintre instructiunile repetitive

cunoscute(for,while ,repeat) fără a apela la formula matematică cunoscută S(n)=n*(n+1)/2

```
Program Suma_n; { Pascal }
Var  n,s,i:word;

BEGIN
Writeln('Introduceti limita n=');Readln(n);
s:=0;
For i:=1 to n do  s:=s+i;
Writeln('s=',s);
Readln;
END.
```

// versiunea C

```
#include <stdio.h>

int n,s;

void main(void){
  printf("Introd. n:"); scanf("%i",&n);
 for(;n>0;n--)s+=n;
 printf("S(n)=%i",s);
}
```

4. Se citeşte valoarea întreagă *p*. Să se determine daca *p* este număr prim.

Descrierea algoritmului:

- un număr p este prim dacă nu are nici un divizor înafară de 1 şi p cu ajutorul unei variabile contor *d* vom parcurge toate valorile intervalului [2.. √p]; acest interval este suficient pentru depistarea unui divizor, căci: $d_1 \mid p \Rightarrow p = d_1*d_2$ (unde $d_1 < d_2$) $\Rightarrow$ $d_1 \le \sqrt{d_1*d_2} = \sqrt{p}$ iar $d_2 \ge \sqrt{d_1*d_2} = \sqrt{p}$

```
Program Nr_prim;  { Pascal }
Var p,i:word;
   prim:boolean;

BEGIN
write('p=');readln(p);
prim:=true;
for i:=2 to trunc(sqrt(p)) do
     if n mod i=0 then  prim:=false;
prim:=true;
if prim then
   write(p,'  este nr prim')
  else
   write(p,'  nu e nr prim');
END.
```

// versiunea C (optimizată !)

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

int p,i,prim;

void main(void){
  printf("Introd. p:"); scanf("%i",&p);
  for(i=3, prim=p % 2; (i<=sqrt(p))&&(prim); i+=2)
     prim=p % i;
 printf("%i %s nr.prim", p, (prim ? "este": "nu este"));
}
```

5. Se citeşte o propoziţie (şir de caractere) terminată cu punct. Să se determine cîte vocale şi cîte consoane conţine propoziţia.

```
Program Vocale;
Var sir:string[80];
  Vocale,Consoane,i:integer;

BEGIN
 Write('Introd.propozitia terminata cu punct:');Realn(sir);
  i:=1;Vocale:=0;Consoane:=0;
  While sir[i]<>'.' do begin
    If Upcase(sir[i]) in ['A','E','I','O','U'] then Inc(Vocale)
    else If Upcase(sir[i]) in ['A'..'Z'] then Inc(Consoane);
    Inc(i);
  end;
  Writeln('Vocale:',Vocale,' Consoane:', Consoane,' Alte
caractere:',i-Vocale-Consoane);
END.
```

// versiunea C

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

int i,vocale=0,consoane=0;
char c,sir[80];

void main(void){
 printf("Introd.propozitia terminata cu punct:");gets(sir);
 for(i=0;sir[i]!='.';i++)
        switch (toupper(sir[i])){
          case 'A':
           case 'E':
           case 'I':
           case 'O':
           case 'U': vocale++; break;
          default: if (isalpha(sir[i])) consoane++;
         }
```

```
    printf("Vocale:%i, Consoane:%i, Alte car.:%i", vocale, consoane, i-vocale-consoane);
}
```

Metoda practică de învăţare ce garantează rezultate imediate

Dacă cele spuse mai sus cu privire la secretul învăţării rapide a programării, acum nu ne mai rămîne decît să începem să aplicăm practic ideile prezentate. Pentru aceasta, avem la dispoziţie următoarea metodă care garantează cu siguranţă rezultate. Iat-o, pe paşi:

1. se citeşte şi se înţelege cît mai bine exemplul de problemă rezolvată (se poate începe chiar cu primul exemplu de mai sus)
2. se acoperă (se ascunde) soluţia şi se încearcă reproducerea ei din memorie (reinventarea soluţiei) pe calculator
3. numai în cazuri excepţionale se poate apela (se poate trage cu ochiul) la soluţie

Oricare dintre noi poate recunoaşte aici metoda pe care o aplică copiii din primele clase primare: metoda trasului cu ochiul la rezultatul aflat la spatele manualului sau al culegerii de probleme. Din moment ce metoda este verificată şi garantată (am folosit-o şi noi cîndva), de ce ne-ar fi ruşine s-o aplicăm acum din nou ?

Iată în continuare o listă de probleme de "antrenament" care au majoritea rezolvarea într-unul din capitolele următoare. Este numai bine pentru a începe să aplicăm metoda oferită chiar acum !

Probleme selecţionate - Enunţuri

Probleme propuse spre rezolvare (probleme de antrenament)

1. Se citesc a, b, c trei variabile reale.

- Să se afişeze maximul şi minimul celor trei numere.
- Să se afişeze cele trei numere în ordine crescătoare.
- Să se determine dacă cele trei numere pot reprezenta laturile unui triunghi. Dacă da, să se determine dacă triunghiul respectiv este isoscel, echilateral sau oarecare.
- Să se determine dacă cele trei numere pot reprezenta laturile unui triunghi. Dacă da, să se determine mărimile unghiurilor sale şi dacă este ascuţit-unghic sau obtuz-unghic.
- Să se afişeze media aritmetică, geometrică şi hiperbolică a celor trei valori.

2. Se citeşte n o valoare întreagă pozitivă.

- Să se determine dacă n este divizibil cu 3 dar nu este divizibil cu 11.
- Să se determine dacă n este pătrat sau cub perfect.
- Să se afişeze primele n pătrate perfecte.
- Să se determine numărul cuburilor perfecte mai mici decît n.
- Să se găsească primul număr prim mai mare decît n.
- Să se afişeze primele n numere prime: 2, 3, 5, 7,..., p_n.
- Să se determine toate numerele de 4 cifre divizibile cu n.
- Să se determine suma cifrelor lui n.
- Să se afişeze răsturnatul lui n. (Ex: n=1993 => n_răsturnat =3991).
- Să se afişeze următorul triunghi de numere:

1
1 2
1 2 3
……..

1 2 3 … n

3. Se citesc m, n două variabile întregi pozitive.

- Să se determine toate pătratele perfecte cuprinse între m şi n, inclusiv.
- Să se determine toate numerele prime cuprinse între m şi n.
- Să se determine toate numerele de 4 cifre care se divid atît cu n cît şi cu m.
- Să se determine c.m.m.d.c. al celor două numere folosind algoritmul lui Euclid.

4. Să se calculeze u_{20} , u_{30} , u_{50} ai şirului cu formula recursivă $u_n=1/12u_{n-1}+1/2u_{n-2}$ pentru n>=2 şi $u_0=1$, $u_1=1/2$.

5. Se citeşte n gradul unui polinom şi şirul a_n, a_{n-1}, … , a_1, a_0 coeficienţilor unui polinom P.

- Se citeşte x, să se determine P(x).
- Se citesc x şi y, să se determine dacă polinomul P schimbă de semn de la x la y.
- Se citeşte a, să se determine restul împărţirii lui P la x-a.

6. Se citesc m, n gradele a două polinoame P şi Q, şi coeficienţii acestora. Să se determine polinomul produs R=PxQ.

7. Se citeşte o propoziţie (şir de caractere) terminată cu punct.

- Să se determine cîte vocale şi cîte consoane conţine propoziţia.
- Să se afişeze propoziţia în ordine inversă şi cu literele inversate (mari cu mici).
- Să se afişeze fiecare cuvînt din propoziţie pe cîte o linie separată.
- Să se afişeze propoziţia rezultată prin inserarea în spatele fiecărei vocale ‘v’ a şirului “pv” (“vorbirea găinească”).

8. Se citeşte m, n dimensiunea unei matrici $A=(a_{i,j})_{mxn}$ de valori reale.

- Se citesc l, c. Să se afişeze matricea obţinută prin eliminarea liniei l şi a coloanei c.
- Se citeşte n întreg pozitiv, să se afişeze matricea obţinută prin permutarea circulară a liniilor matricii cu n poziţii.
- Să se determine suma elementelor pe fiecare linie şi coloană.
- Să se determine numărul elementelor pozitive şi negative din matrice.
- Să se determine linia şi coloana în care se află valoarea maximă din matrice.
- Să se determine linia care are suma elementelor maximă.

9. Se citesc m, n, p şi apoi se citesc două matrici $A=\mathbf{(}a_{i,j}\mathbf{)}_{mxn}$ şi $B=\mathbf{(}b_{j,k}\mathbf{)}_{\mathbf{nxp}}$.Să se determine matricea produs C=AxB.

10. Se citeşte un fişier ce conţine mai multe linii de text.

- Să se afişeze linia care are lungime minimă.
- Să se afişeze liniile care conţin un anumit cuvînt citit în prealabil.
- Să se creeze un fişier care are acelaşi conţinut dar în ordine inversă.

Probleme dificile

După cum se poate bănui, informatica conţine şi ea, la fel ca matematica, o mulţime de probleme foarte dificile care îşi aşteaptă încă rezolvarea. Asemănarea cu matematica ne interesează mai ales în privinţa unui aspect "capcană" asupra căruia dorim să atragem atenţia aici.

Enunţurile problemelor dificile sau foarte dificile de informatică este, în 99% din cazuri, foarte simplu şi poate fi citit şi înţeles de orice student. Acest fapt consituie o capcană sigură pentru cei ignoranţi. Dacă în matematică lucrurile nu stau aşa, asta se datorează numai faptului că studiul matematicii are vechime şi problemele, împreună cu dificultăţile lor, sînt ceva mai bine cunoscute. În informatică nu avem însă aceeaşi situaţie. Ba chiar se întîmplă că probleme foarte dificile sînt amestecate în culegerile de probleme de informatică printre probleme uşoare, mai ales datorită lipsei de cultură de specialitate a autorilor.

Acest capitol îşi propune să pună în gardă în privinţa dificultăţii problemelor oferind o mică iniţiere în acest domeniu (mai multe se pot afla studiind *Complexitatea algoritmilor şi dificultatea problemelor*). Deasemeni el îşi propune să umple lacuna ce mai există încă la ora actuală în cultura de specialitate.

Dificultatea problemelor de programare a căror enunţuri urmează este considerată maximă de teoreticienii informaticii (ele se numesc *probleme NP-complete*). Nu vă lăsaţi păcăliţi de faptul că le-aţi întîlnit în unele culegeri de programare. Ele sînt depăşite în dificultate doar de problemele insolvabile algoritmic ! Dar în ce constă dificultatea lor ?

Spre deosebire de matematică, dificultatea problemelor de informatică nu este dată de faptul că nu se cunoaşte un algoritm de rezolvare a lor, ci datorită faptului că nu se cunoaşte un algoritm *eficient* **(!)** de rezolvare a lor. Existenţa unei metode de

proiectare a algoritmilor atît de general valabilă, cum este metoda back-tracking, face ca prea puţine probleme cu care ne putem întîlni să nu aibă o soluţie. Dar, întrucît în cazul metodei back-tracking, căutarea soluţiei se face într-un mod exhaustiv (se caută "peste tot", pentru ca să fim siguri că nu lăsăm nici o posibilitate neexplorată), durata căutării are o creştere exponenţial-proporţională cu dimesiunea datelor de intrare. De exemplu, timpul de căutare care depinde de valoarea de intrare *n* poate avea o expresie de forma $T(n)=c\cdot 2^n$ *secunde*, unde *c* este un factor de proporţionalitate ce poate varia, să zicem, de la *c=12.5* cînd algoritmul este executat pe un calculator sau *c=62.8* cînd el este rulat pe un calculator de cinci ori mai performant. Dar, indiferent de calculator, pentru *n=100* avem $2^{100}=(2^{10})^{10}\approx(10^3)^{10}=10^{30}$, deci timpul măsurat în secunde are ordinul de mărime mai mare de 30. Cea mai largă estimare pentru vîrsta Universului nostru nu depăşeşte *20 mild. ani* ceea ce transformat în secunde conduce la un ordin de mărime mai mic de 20. Deci, chiar şi pentru valori mici ale lui *n* (de ordinul sutelor) am avea de aşteptat pentru găsirea soluţiei de 10 miliarde de ori mai mult decît a trecut de la Big Bang încoace ! Pot fi în această situaţie considerate astfel de programe ca rezolvări rezonabile, doar pentru că ele găsesc soluţia în cazurile în care *n=2, 3, 4, ..., 10* ?

Exemplele următoare sînt doar cîteva, uşor de întîlnit "din greşeală", dintr-o listă cunoscută ce conţine la ora actuală peste şase sute de astfel de probleme. Pentru fiecare din aceste probleme nu li se cunosc alte soluţii decît inutilii algoritmi de gen back-tracking. În listă apare des noţiunea de *graf*, aşa că o vom introduce în continuare cît mai simplu cu putinţă: printr-un graf se înţelege o mulţime de *vîrfuri* şi o mulţime de *muchii* care unesc unele vîrfuri între ele. Orice hartă (schematizată) rutieră, feroviară sau de trafic aerian reprezintă desenul unui graf.

1. **Problema partiţionării sumei.** Fie *C* un întreg pozitiv şi $d_1, d_2, \ldots, d_n$ o mulţime de *n* valori întregi pozitive. Se cere să se

găsească o partiţionare a mulţimii $d_1, d_2, ..., d_n$ astfel încît suma elementelor partiţiei să fie exact *C*.

2. **Problema rucsacului.** Avem un rucsac de capacitate întreagă pozitivă *C* şi *n* obiecte cu dimensiunile $d_1, d_2, ..., d_n$ şi avînd asociate profiturile $p_1, p_2, ..., p_n$ (în caz că ajung în rucsac). Se cere să se determine profitul maxim ce se poate obţine prin încărcarea rucsacului (fără ai depăşi capacitatea).

3. **Problema colorării grafului.** Să se determine numărul minim de culori (*numărul cromatic*) necesar pentru colorarea unui graf astfel încît oricare două vîrfuri unite printr-o muchie (*adiacente*) să aibă culori diferite.

4. **Problema împachetării.** Presupunînd că dispunem de un număr suficient de mare de cutii fiecare avînd capacitatea *1* şi *n* obiecte cu dimensiunile $d_1, d_2, ..., d_n$, cu $0<d_i<1$, se cere să se determine numărul optim (cel mai mic) de cutii necesar pentru împachetarea tutror celor *n* obiecte.

5. **Problema comisului voiajor.** (varianta simplificată) Dîndu-se un graf (o hartă), se cere să se găsească un circuit (un şir de muchii înlănţuite) care trece prin fiecare vîrf o singură dată.

Majoritatea acestor probleme apar ca probleme centrale la care se reduc în ultimă instanţă problemele concrete ale unor domenii capitale ale economiei şi industriei, cum sînt de exemplu planificarea investiţiile, planificarea împrumuturilor şi eşalonarea plăţii dobînzilor, alocarea şi distribuirea resurselor primare (mai ales financiare), etc. Pentru nici una din aceste probleme strategice nu se cunoaşte un *algoritm optim* de rezolvare, ci doar soluţii aproximative. Dacă s-ar cunoaşte algoritmii de soluţionare optimă atunci majoritatea sectoarelor şi proceselor macro- şi micro-economice ar putea fi conduse *în timp real* şi *optim* (!!) cu calculatorul, fără a mai fi necesară prezenţa umană.

Un exemplu cert de domeniu care s-a dezvoltat extraordinar şi în care rolul soft-ului a fost esenţial este chiar domeniul construcţiei de calculatoare, mai ales domeniul

proiectării şi asamblării de micro-procesoare. Dacă aţi văzut că schema electronică internă de funcţionare a unui microprocesor din familia Pentium, dacă ar fi desenată clasic, ar ocupa o planşă de dimensiuni 5x5 metri (!), nu mai aveţi cum să vă îndoiţi de faptul că numai un soft de proiectare şi cablare performant mai poate controla şi stăpîni super-complexitatea rezultată. Puţină lume ştie însă că astfel de programe de proiectare performante au putut să apară numai datorită faptului că problema ce stă în spatele funcţionării lor, *problema desenării grafurilor planare,* nu se află pe lista de mai sus a problemelor foarte dificile ale informaticii !

Probleme nesoluţionate încă

Aşa cum s-a putut constata în capitolul anterior, există multe probleme în informatică pentru care încă nu se cunosc soluţii eficiente. În continuare vom oferi o listă de probleme nesoluţionate încă. De fapt, ele apar mai ales în matematică, fiind cunoscute sub numele de *conjecturi,* şi au toate ca specific un fapt care este de mare interes pentru programatori. Incertitudinea asupra lor ar putea fi definitiv înlăturată nu numai prin demonstraţie matematică ci şi cu ajutorul formidabilei puteri de calcul a computerelor. Astfel, fiecare din aceste conjecturi numerice ar putea fi infirmată (concluzia ar fi atunci că *conjectura este falsă*) dacă i s-ar găsi un contraexemplu. Este necesar doar să se găsească un set de numere pentru care propoziţia respectivă să fie falsă. Ori, acest efort nu este la îndemîna niciunui matematician dar este posibil pentru un programator înzestrat şi pasionat. El nu are decît să scrie un program eficient şi să pună calculatorul să caute un contra-exemplu.

Atragem atenţia asupra unui aspect important. Fiecare problemă conţine aceeaşi capcană ca şi în problemele capitolului anterior: algoritmii de căutare a contra-exemplelor pot fi concepuţi rapid, relativ simpli şi cu efort de programare redus

(de exemplu, prin trei-patru cicluri *for* imbricate sau printr-o soluţie gen *back-tracking*) dar ei vor deveni în scurt timp total ineficienţi şi vor conduce la programe mari consumatoare de timp. De aceea, vă sugerăm să trataţi cu multă atenţie problemele din acest capitol. După părerea noastră, abordarea acestui tip de probleme cere din partea programatorului un anumit grad de măiestrie !

Rezolvînd **numai una** dintre ele veţi fi recompensaţi pe măsură: riscaţi să deveniţi celebri !

1. **Conjectura lui Catalan.** Singurele puteri naturale succesive sînt $8=2^3$ şi $9=3^2$.

Observaţie: într-o exprimare matematică riguroasă, singura soluţie în numere naturale *m, n, p, q* a ecuaţiei $n^m+1=p^q$ este *n=2, m=3, p=3 şi q=2*.

Comentariu: avem şirul numerelor naturale *1, 2, 3, 4, 5,…*; încercuind toate puterile de gradul 2: *1, 4, 9, 16, 25,…* apoi toate cele de gradul 3: *1, 8, 27, 64, 125, …* apoi cele de grad 4, 5, … vom constata că singurele două numere încercuite alăturate sînt 8 şi 9 ! Adică puterile obţinute, cu cît sînt mai mari, cu atît au tendinţa să se "împrăştie" şi să se "distanţeze" unele de altele tot mai tare. În mod misterios, ele nu-şi suportă vecinătatea unele cu altele !

2. Conjectura cutiei raţionale. Nu se cunoaşte existenţa unei cutii paralelipipedice avînd lungimile celor trei laturi, ale celor trei diagonale ale feţelor şi a diagonalei principale întregi.

Observaţie: într-o exprimare matematic riguroasă, nu se cunoaşte să existe trei întregi *a, b, c* astfel încît a^2+b^2, b^2+c^2 , c^2+a^2 şi $a^2+b^2+c^2$ să fie toate patru pătrate perfecte.

Comentariu: în multe subdomenii ale construcţiilor ,de exemplu să ne gîndim la stîlpii de înaltă tensiune ridicaţi pe vîrfuri înalte de munte şi asamblaţi în întregime "la faţa locului" numai din bare îmbinate cu şuruburi (fără sudură), este de mare interes ca dintr-un număr cît mai mic de subansamble simple (un

fel de "cărămizi") să se asambleze obiecte mari cu cît mai multe configuraţii. Evident, dimensiunile obiectelor rezultate vor avea mărimea ca o combinaţie întreagă ale dimensiunilor subansamblelor iniţiale. După cum rezultă însă din conjectură, se pare că este imposibil să se construiască scheletul întărit (pe diagonale) al unei cutii paralelipipedice din bare de lungimi tipizate. Cel puţin una din diagonale necesită ajustarea lungimii unei bare !

3. **Problema umplerii pătratului unitate.** Întrebare: este posibil ca mulţimea dreptunghiurilor de forma *1/k* x *1/(k+1)*, pentru fiecare k întreg pozitiv, să umple în întregime şi fără suprapuneri pătratul unitate, de latură *1*x*1* ?

Observaţie: este evident că suma infinită a ariilor dreptunghiurilor este egală cu aria pătratului unitate. Avem $\Sigma_{k>0} 1/(k(k+1)) = \Sigma_{k>0} (1/k - 1/(k+1)) = 1.$

Comentariu: aparent, descoperirea dezvoltărilor în serie pare să fi plecat de la unele evidente propietăţi geometrice, uşor de sesizat chiar din desene simple în care valorilor numerice li se asociază segmente de lungimi corespunzătoare. Iată însă o surpriză în această situaţie: suma seriei numerice este evidentă analitic însă reprezentarea geometrică a "fenomenului" este "imposibilă" !

4. **Conjectura fracţiilor egiptene (atribuită lui Erdös şi Graham).** Orice fracţie de forma *4/n* se descompune ca sumă de trei fracţii egiptene (de forma *1/x*).

Observaţie: într-o exprimare matematic riguroasă, pentru orice *n* natural există trei valori naturale, nu neapărat distincte, *x, y,* şi *z* astfel încît *4/n=1/x+1/y+1/z*.

Comentariu: este încă un mister motivul pentru care egiptenii preferau descompunerea fracţiilor numai ca sumă de fracţii egiptene. Descoperiseră ei această descompunere

minimală a fracţiilor de forma *4/n* ? Dar mai ales, ce procese fizice reale erau astfel mai bine modelate ? Înclinăm să credem că există o legătură între fenomenele fizice ondulatorii, transformata Fourier şi fracţiile egiptene !

5. Problema punctului raţional. Există un punct în plan care să se afle la o distanţă raţională de fiecare din cele patru vîrfuri ale pătratului unitate ?

Observaţie: dacă considerăm un pătrat unitate avînd vîrfurile de coordonate *(0,0), (1,0), (0,1)* şi *(1,1)* atunci se cere găsirea unui punct *(x,y)* astfel încît x^2+y^2, $(x-1)^2+y^2$, $x^2+(y-1)^2$ şi $(x-1)^2+(y-1)^2$ să fie toate patru pătrate perfecte. Atenţie, *x* şi *y* nu este obligatoriu să fie întregi ! Acest fapt ridică foarte serioase probleme la proiectarea unui algoritm de căutare a unui astfel de punct *(x,y).*

Comentariu: la fel ca şi în cazul cutiei raţionale, se pare că există limitări serioase şi neaşteptate în încercarea de optimizare a numărului de subansamble necesare pentru construierea scheletelor sau cadrelor de susţinere. Se pare că cele două dimensiuni pe care geometria plană se bazează conduce la o complexitate inerentă neaşteptat de mare !

6. **Problema sumei de puteri.** Care este suma seriei de inverse de puteri $1/1+1/2^3+1/3^3+1/4^3+1/5^3+\ldots$?

Observaţie: se cere să se spună către ce valoare converge seria $\Sigma_{k>0} 1/k^3$ sau $\Sigma_{k>0} k^{-3}$. Se ştie că în cazul în care în locul puterii a 3-ia (cu minus) punem puterea a 2-a (cu minus) seria converge la $\pi^2/6$, în cazul în care în locul puterii a 3-ia punem puterea a 4-a seria converge la $\pi^4/90$.

Comentariu: deşi pare a fi o problemă de analiză matematică pură deoarece ni se cere să găsim expresia sintetică şi nu cea numerică aproximativă a sumei seriei, există însă uluitoare descoperiri asemănătoare ale unor formule de analiză

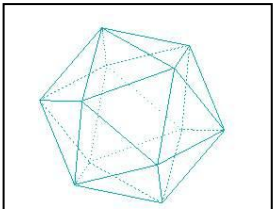
numerică sau chiar dezvoltări în serie (cea mai celebră fiind cea a lui cifrelor hexazecimale ale lui π) făcute cu ajutorul calculatorului prin calcul simbolic ! Mai multe amănunte găsiţi la adresa corespunzătoare de Internet pe care am trecut-o în ultimul capitol.

7. **Problema ecuaţiei diofantice de gradul 5.** Există *a, b, c,* and *d* întregi pozitivi astfel încît $a^5+b^5=c^5+d^5$?

Observaţie: Se cunoaşte că în cazul în care puterea este 3 avem soluţia: $1^3+12^3=9^3+10^3$ iar în cazul în care puterea este 4 avem soluţia: $133^4+134^4=59^4+158^4$.

Comentariu: căutarea unor algoritmi generali de rezolvare a ecuaţiilor diofantice a condus la importante descoperiri în matematică dar şi în informatică. De exemplu, celebrul matematician *Pierre Fermat,* "stîrnit" fiind de problemele conţinute în lucrarea *Arithmetika* a matematicianului antic *Diofant din Alexandria* (de unde şi numele *ecuaţiilor diofantice*), a descoperit în 1637 faimoasa sa teoremă: *Ecuaţia diofantică $x^n+y^n=z^n$ nu admite soluţie pentru n>2*. Dar tot în aceeaşi perioadă a descoperit şi faptul că cea mai mică soluţie a ecuaţiei diofantice $x^2 - 109*y^2 = 1$ este perechea *x=158 070 671 986 249* şi *y= 15 140 424 455 100*. Dumneavoastră încercaţi doar să verificaţi această soluţie fără ajutorul calculatorului şi vă veţi putea da seama de performanţele pe care le-a realizat Fermat ! În informatică este acum cunoscut şi demonstrat că este imposibil să se construiască un algoritm general pentru rezolvarea ecuaţiilor diofantice !

8. **Problema celor 13 oraşe.** Puteţi localiza 13 oraşe pe o planetă sferică astfel încît distanţa minimă dintre oricare două dintre ele să fie cît mai mare cu putinţă ?


Icosahedron

Observaţie: de fapt nu se cunoaşte *cît de mult poate fi mărită* distanţa minimală ce

se obţine dintre cele 78 de distanţe (date de cele $78=C^2_{13}$ de împerecheri posibile de oraşe).

Comentariu: dacă s-ar cere localizarea a doar 12 puncte pe sferă, nu este greu de arătat că aşezarea care îndeplineşte condiţia cerută este în vîrfurile unui icosaedru (vezi figura alăturată). În acest caz, distanţa minimă maximizată este egală cu latura icosaedrului. Este greu de crezut că în cazul descoperirii aşezării a 13 puncte pe sferă se poate porni tocmai de la icosaedru ! Evident că în rezolvarea aplicativ-practică a acestui tip de probleme nesoluţionate geometric pînă în prezent rolul programatorului poate fi capital. La ora actuală pentru astfel de situaţii se oferă soluţii aproximative. Acestea constau din algoritmi care încearcă să aproximeze cît mai exact soluţia optimă într-un timp rezonabil de scurt. Evident că în aceste condiţii algoritmii de căutare exhaustivă (gen back-tracking) sînt cu totul excluşi !

9. **Conjectura lui Collatz.** Se pleacă de la un *n* întreg pozitiv. Dacă *n* este par se împarte la doi; dacă *n* este impar se înmulţeşte cu trei şi i se adună unu. Repetînd în mod corespunzător doar aceşti doi paşi se va ajunge întotdeauna la *1* indiferent de la ce valoare *n* se porneşte ?

Observaţie: de exemplu, pornind de la *n=6* obţinem în 8 paşi şirul valorilor: *6, 3, 10, 5, 16, 8, 4, 2, 1*.

Comentariu: valoarea finală *1* este ca o "gaură neagră" care absoarbe în final şirul obţinut. "Raza" de-a lungul căreia are loc "căderea" în gaura neagră *1* este dată mereu de şirul puterilor lui 2: *2, 4, 8, 16, 32, 64, ...* cu ultima valoare de forma *3k+1*, adică *4, 16, 64, 256, ...* Se pare că valorile obţinute prin cele două operaţii nu pot "să nu dea" nicicum peste acest şir care le va face apoi să "cadă în gaura neagră" 1!

10. **Problema înscrierii pătratului.** Dîndu-se o curbă simplă închisă în plan, vom putea întotdeauna găsi patru puncte pe curbă care pot să constituie vîrfurile unui pătrat ?

Observaţie: în cazul curbelor închise regulate (ce au axe de simetrie: cerc, elipsă, ovoid) nu este greu de arătat prin construire efectivă că există un pătrat ce se "sprijină" pe curbă. Pare însă de nedovedit acelaşi fapt în cazul unor curbe închise foarte neregulate ! Găsirea celor patru puncte, într-o astfel de situaţie, este de neimaginat fără ajutorul calculatorului !

Comentariu: o consecinţă surprinzătoare a acestei conjecturi este faptul că pe orice curbă de nivel (curbă din teren care uneşte punctele aflate toate la aceaşi altitudine) am putea găsi patru puncte de sprijin pentru o platformă pătrată (un fel de masă) perfect orizontală, de mărime corespunzătoare. Acest fapt ar putea să explice ampla răspîndire a meselor cu patru picioare (!?) în detrimentul celor cu trei: dacă îi cauţi poziţia, cu siguranţă o vei găsi şi o vei putea aşeza pe toate cele patru picioare, astfel masa cu patru picioare va oferi o perfectă stabilitate şi va sta perfect orizontală, pe cînd cea cu trei picioare deşi stă acolo unde o pui din prima (chiar şi înclinată) nu oferă aceeaşi stabilitate.

În speranţa că am reuşit să vă stîrnim interesul pentru astfel de probleme nesoluţionate încă şi care sînt grupate pe Internet în liste cuprinzînd zeci de astfel de exemple (vezi adresa oferită în ultimul capitol), încheiem acest capitol cu următoarea constatare: descoperirile deosebite din matematica actuală au efecte rapide şi importante nu numai în matematică ci şi în informatică. Să oferim doar un singur exemplu de mare interes actual: algoritmii de încriptare/decriptare cu cheie publică, atît de folosiţi în comunicaţia pe Internet, se bazează în întregime pe proprietăţile matematice ale divizibilităţii numerelor prime.

Ceea ce este interesant şi chiar senzaţional este faptul că în informatică nevoia de programe performante a condus la implementarea unor algoritmi care se bazează pe cele mai noi descoperiri din matematică, chiar dacă acestea sînt încă în stadiul de conjecturi! De exemplu, pentru acelaşi domeniu al criptării cu cheie publică există deja algoritmi de primalitate

senzaţional de performanţi care se bazează pe Ipoteza (conjectura) lui Riemman. (Mai multe amănunte puteţi găsi la adresele de Internet pe care le oferim în ultimul capitol.)

Este acest fapt legitim ? Ce încredere putem avea în astfel de programe ? După părerea noastră putem acorda o totală încredere acestor algoritmi dar numai în limitele "orizontului" atins de programele de verificare a conjecturii folosite. Dacă programul de verificare a verificat conjectura numerică pe intervalul 1- 10^{30} atunci orizontul ei de valabilitate este 10^{30}. Domeniile numerice pe care le pot acoperi calculatoarele actuale sînt oricum foarte mari şi implicit oferă o precizie suficientă pentru cele mai multe calcule cu valori extrase din realitatea fizică.

Noţiuni aprofundate de programare

Metode şi strategii de proiectare a algoritmilor (alias tehnici de programare)

În rezolvarea sa cu ajutorul calculatorului orice problemă trece prin trei etape obligatorii: *Analiza problemei, Proiectarea algoritmului de soluţionare* şi *Implementarea algoritmului într-un program pe calculator.* În ultima etapă, sub acelaşi nume, au fost incluse în plus două subetape cunoscute sub numele de *Testarea* şi *Întreţinerea programului.* Aceste subetape nu lipsesc din "ciclul de viaţă" a oricărui produs-program ce "se respectă" dar , pentru simplificare, în continuare ne vom referi doar la cele trei mari etape..

Dacă etapa *implementării* algoritmului într-un program executabil este o etapă exclusiv practică, realizată "în faţa calculatorului", celelalte două etape au un caracter teoretic pronunţat. În consecinţă, primele două etape sînt caracterizate de un anumit grad de abstractizare. Din punct de vedere practic şi în ultimă instanţă criteriul decisiv ce conferă succesul rezolvării problemei este dat de calitatea implementării

propriuzise. Mai precis, succesul soluţionării este dat de performanţele programului: utilitate, viteză, fiabilitate, manevrabilitate, lizibilitate, etc. Este imatură şi neprofesională "strategia" programatorilor începători care neglijînd primele două etape sar direct la a treia, fugind de analiză şi de componenta abstractă a efortului de soluţionare. Ei oferă cu toţii aceeaşi justificare: "Eu nu vreau să mai pierd vremea cu …, am să fac programul cum ştiu eu. Pînă cînd nu o să facă cineva altul mai bun decît al meu, pînă atunci…nu am cu cine sta de vorbă !".

Este adevărat că ultima etapă în rezolvarea unei probleme – implementarea – este într-adevăr decisivă şi doveditoare, dar primele două etape au o importanţă capitală. Ele sînt singurele ce pot oferi răspunsuri la următoarele întrebări dificile: *Avem certitudinea că soluţia găsită este corectă ? Avem certitudinea că problema este complet rezolvată ? Cît de eficientă este soluţia găsită ? Cît de departe este soluţia aleasă de o soluţie optimă ?*

Să menţionăm în plus că literatura de specialitate conţine un număr impresionant de probleme "capcană" pentru începători şi nu numai. Ele sînt toate inspirate din realitatea imediată dar pentru fiecare dintre ele nu se cunosc soluţii eficiente în toată literatura de profil. Există printre ele chiar unele probleme extrem de dificile pentru care **s-a demonstrat riguros** că nu admit soluţie cu ajutorul calculatorului. (Mai precis, s-a demonstrat că ele nu admit soluţie prin metode algoritmice, în spiritul tezei Turing-Church). Cîţi dintre programatorii începători n-ar fi surprinşi să afle că problema "atît de simplă" (ca enunţ) a cărei soluţionare tocmai au abandonat-o este de fapt o problemă dovedită ca fiind intratabilă sau chiar insolvabilă algoritmic ? Partea proastă a lucrurilor este că, aşa cum ciupercile otrăvite nu pot fi cu uşurinţă deosebite de cele comestibile, tot astfel problemele netratabile pot fi cu uşurinţă confundate cu nişte probleme uşoare la o privire rapidă şi lipsită de experienţă.

Să înţelegem mai întîi care este "cheia" ce conduce la răspunsuri pentru întrebările de mai sus iar apoi vom trece la prezentarea metodelor clasice de proiectare a soluţiilor. Aceste metode de proiectare a algoritmilor-soluţie sînt cunoscute în literatura de specialitate sub numele de *tehnici de programare* şi sînt considerate metode sau instrumente soft eficiente şi cu arie largă de acţiune.

Dacă ar fi să sintetizăm în cîte un cuvînt efortul asupra căruia se concentrează fiecare din primele două etape – *analiza* şi *proiectarea* – acestea ar fi: *corectitudine* şi *eficienţă*. Etapa de analiză este singura care permite *dovedirea cu argumente riguroase* a corectitudinii soluţiei, iar etapa de proiectare este singura care poate *oferi argumente precise* în favoarea eficienţei soluţiei propuse.

În general problemele de informatică au în forma lor iniţială sau în enunţ o caracteristică pragmatică. Ele sînt foarte ancorate în realitatea imediată şi aceasta le conferă o anumită asemănare. Totuşi ele au în forma iniţială un grad mare de *eterogenitate*, *diversitate* şi *lipsă de rigoare.* Fiecare dintre aceste atribute "negative" este un obstacol major pentru demonstrarea corectitudinii soluţiei. Rolul esenţial al etapei de analiză este deci acela de a transfera problema "de pe nisipurile mişcătoare" ale realităţii imediate de unde ea provine într-un plan abstract, adică de *a o modela*. Acest "univers paralel" este dotat cu mai multă rigoare şi disciplină internă, avînd legi precise, şi poate oferi instrumentele logice şi formale necesare pentru demonstrarea riguroasă a corectitudinii soluţiei problemei.

Planul abstract în care sînt "transportate" toate problemele este planul sau *universul obiectelor matematice.* Acest *univers al matematicii* este unanim acceptat (de ce ?!) iar corespondentul problemei în acest plan va fi *modelul matematic abstract* asociat problemei. Demonstrarea corectitudinii proprietăţilor ce leagă obiectele universului matematic a fost şi este sarcina matematicienilor. Celui ce analizează problema din punct de vedere informatic îi revine sarcina (nu tocmai uşoară) de a

dovedi printr-o **demonstraţie constructivă** că există o *corespondenţă precisă (bijectivă)* între părţile componente ale problemei reale, "dezasamblată" în timpul analizei, şi părţile componente ale modelului abstract asociat. Odată descoperită, formulată precis şi dovedită, această "perfectă oglindire" a problemei reale în planul obiectelor matematice oferă certitudinea că toate proprietăţile şi legăturile ce există între subansamblele modelului abstract se vor regăsii precis (prin reflectare) între părţile interne ale problemei reale, şi invers. Atunci, soluţiei abstracte descoperită cu ajutorul modelului matematic abstract îi va corespunde o soluţie reală concretizată printr-un algoritm ce poate fi implementat într-un program executabil.

Aceasta este calea generală de rezolvare a problemelor şi orice poate verifica. Ca şi exerciţiu, să se demonstreze corectitudinea (să se aducă argumente precise, clare şi convingătoare în favoarea *corectitudinii*) metodei de extragere a radicalului învăţată încă din şcoala primară sau a algoritmului lui Euclid de determinare a celui mai mare divizor comun a două numere prin împărţiri întregi repetate. Argumentele elevilor de forma: "Este corect pentru că aşa ne-a învăţat doamna profesoară!" sau "Este corect pentru că aşa face toată lumea !" sînt "normale" atît timp cît nu li se oferă o argumentaţie matematică riguroasă.

Ideea centrală a etapei a doua – proiectarea uni algoritm de soluţionare eficient poate fi formulată astfel: din studiul *proprietăţilor* şi *limitelor* modelului matematic abstract asociat problemei se deduc *limitele inferioare ale complexităţii minimale ("efortului minimal obligatoriu")* inerente **oricărui** algoritm ce va soluţiona problema în cauză. *Complexitatea internă a modelului abstract* şi *complexitatea soluţiei abstracte* se va reflecta imediat asupra *complexităţii reale a algoritmului*, adică asupra *eficienţei*, de soluţionare a problemei. Acest fapt permite *prognosticarea* încă din această fază – faza de proiectare a algoritmului de

soluţionare – a eficienţei practice, măsurabilă ca *durată de execuţie*, a programului.

Această corespondenţă exactă între complexitatea modelului abstract şi complexitatea algoritmului de soluţionare oferă cheia unor demonstraţii riguroase a imposibilităţii existenţei soluţiei prin metode algoritmice pentru o listă întreagă de probleme (cum ar fi de exemplu Problema a 10-a a lui Hilbert, formulată încă din 1900).

Detailînd cele prezentate deja, vom construi în continuare cadrul teoretic general pentru înţelegerea strategiilor de proiectare a algoritmilor.

Creşterea impresionantă a puterii de calcul a calculatoarelor i-a "obligat" pe informaticienii ultimilor treizeci de ani să nu se mai eschiveze de la abordarea problemelor dificile cu caracter algoritmic din diverse domenii care au intrat în atenţia matematicienilor încă de la începutul acestui secol. De altfel, astfel de probleme cu soluţii algoritmice nu constituiau neapărat o noutate pentru matematicienii începutului de secolul. Încă de pe vremea lui Newton matematicienii şi-au pus, de exemplu, problema descoperirii unor metode precise (adică algoritmi!) de determinare în paşi de aproximare succesivă a soluţiei unei ecuaţii ce nu poate fi rezolvată prin radicali. Dar "boom-ul" dezvoltării tehnicii de calcul din a doua jumătate a secolului a creat posibilitatea abordării unor probleme cheie pentru anumite domenii strategice (de exemplu, controlul şi dirijarea sateliţilor pe orbită, probleme de planificare sau optimizare în economie, etc.) care se reduc în fapt la soluţionarea eficientă a unor probleme de optimizare matematică prin metode iterative (algoritmi).

Spre deosebire de aceste probleme a căror succes în soluţionare a fost total şi cu consecinţele ce se văd, există însă o serie de probleme dificile inspirate din realitate care se cer imperios rezolvate eficient cu ajutorul calculatorului.

Principală caracteristică a acestora este că, datorită generalităţii lor sau datorită dificultăţii "ascunse", în literatura de

specialitate nu există metode iterative eficiente de rezolvare a lor şi nu se ştie dacă ele admit astfel de soluţii. Singurul fapt ce poate fi stabilit dinainte în cazul soluţionării unor astfel de probleme este "spaţiul" în care soluţia trebuie căutată. Ceea ce trebuie atunci construită este o *strategie* corectă şi cît mai generală de căutare a soluţiei (soluţiilor) în acel spaţiu de căutare a soluţiilor.

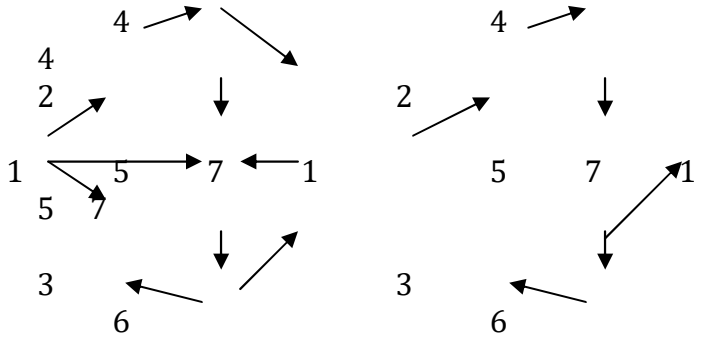
Exemplu concret: există o clasă întreagă de probleme ce cer implicit să se genereze toate obiectele unei mulţimi (cum ar fi problema generării tuturor permutărilor unei mulţimi cu *n* elemente). În acest caz este cunoscută dinainte proprietatea ce trebuie să o îndeplinească fiecare soluţie ca să fie un obiect al spaţiului de căutare a soluţiilor. Efortul de soluţionare va fi redus atunci la aflarea, căutarea sau generarea pe baza proprietăţii respective a *tuturor* obiectelor posibile, fără însă a lăsa vreunul pe dinafară.

Modelul matematic abstract cel mai general care permite modelarea acestui tip de probleme este *graful*. Un graf este un obiect matematic riguros care, simplificat, poate fi privit ca fiind o diagramă formată din noduri unite prin săgeţi (muchii). De exemplu, orice hartă feroviară sau rutieră poate fi privită ca un graf cu mulţimea nodurilor formată din localităţi iar mulţimea muchiilor formată din rutele de transport directe dintre localităţile respective. Graful permite descrierea legăturilor şi a relaţiilor ce există între diferite obiecte abstracte reprezentate prin noduri. Experienţa arată că acest *model matematic abstract* este *cel mai general* şi cel mai potrivit pentru descrierea unui spaţiu de căutare a soluţiilor unei probleme. În cazul spaţiului de căutare, nodurile sînt soluţiile posibile (ipotetice). Două noduri în graf vor fi unite prin săgeţi (muchii) dacă cele două soluţii posibile au în comun o aceeaşi proprietate. Muchiile grafului sînt "punţile" ce vor permite algoritmului trecerea de la un nod la altul, de la o soluţie ipotetică la alta, în timpul procesului de căutare a soluţiei (sau a tuturor soluţiilor). Rezultă că strategiile cele mai generale de căutare a soluţiei (soluţiilor) pe modelul

abstract asociat problemei sînt reductibile la *metodele generale de traversare a unui graf*.

Ordinea de traversare a grafului determină precis *arborele de traversare* a grafului. Acest arbore este de fapt un subgraf particular al grafului iniţial, avînd acelaşi număr de noduri şi ca rădăcină vîrful iniţial de pornire. Cele două metode clasice de traversare a unui graf (căutare într-un graf) poartă în literatura de specialitate numele: *BreathFirstSearch* (BFS) şi *DepthFirstSearch* (DFS), respectiv *Traversarea în lăţime* (sau traversarea pe nivele) şi *Traversarea în adîncime* (traversarea "labirintică") a grafului. Ambele metode stau la baza celei mai cunoscute strategii de proiectare a algoritmilor (impropriu denumită tehnică de programare): *BackTracking* respectiv *căutare (traversare) în spaţiul de căutare a soluţiilor (a grafului) cu revenire pe "urma" lăsată*.

Iată un exemplu de graf (7 noduri şi 10 arce-săgeţi) şi ordinea sa de traversare prin cele două metode:



Ordinea de parcurgere a celor 7 vîrfuri ale grafului, ţinînd cont şi de sensul dat de săgeţi, este în cazul DFS (în adîncime): 1,2,4,5,6,3,7 aşa cum se vede din arborele parcurgerii în adîncime. Din fiecare nod se continuă cu nodul (nevizitat încă) dat de prima alegere posibilă: de exemplu, din 4 se continuă cu 5 (ales în favoarea lui 7). Se poate observa cum din nodul 3,

nemaiexistînd continuare, are loc o revenire pe "pista lăsată" pînă în nodul 6 de unde se continuă parcurgerea în adîncime cu prima alegere posibilă. În cazul BFS (în lăţime) ordinea de traversare este: 1,2,3,4,5,7,6 aşa cum se poate vedea în arborele parcurgerii în lăţime. În această situaţie, dintr-un nod sînt vizitaţi toţi vecinii (nevizitaţi încă), iar apoi se face repetă acelaşi lucru pentru fiecare nod vecin, în ordinea vizitării. Se observă cum nodul 7 este vizitat înaintea nodului 6, fiind vecin cu nodul 4. (De fapt, aceasta se explică prin faptul că distanţa de la 1 la 7 este mai mică cu o unitate decît distanţa de la 1 la 6.) Putem spune că în cazul traversării în lăţime ordinea de traversare este dată de depărtarea nodurilor faţă de nodul de start.

Iată cum arată procedura generală DepthFirstSearch (DFS) *de traversare a unui graf descrisă în pseudo-cod în varianta recursivă:*

Procedura DFS(v:nod);
Vizitează v;
Marchează v; // v devine un nod vizitat //
Cît timp (există w nemarcat nod adiacent lui v) execută **DFS**(w);

Să nu uităm că această procedură poate fi privită ca "scheletul" pe care se construieşte orice procedură backtracking recursivă.

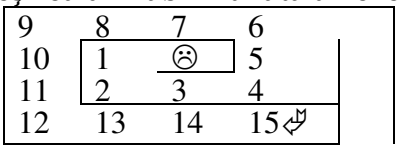
BackTracking.

Pentru a preciza mai exact în ce constă această metodă, vom relua pe un exemplu concret cele spuse deja. Avem următoarea problemă: se cere generarea tuturor permutărilor unei mulţimi de *n* elemente ce nu conţin elementul *x* (dat dinainte) pe primele două poziţii. Conform celor afirmate, este suficient să "construim" modelul abstract - graful - (mai precis arborele) tuturor permutărilor celor *n* elemente. Apoi, printr-o

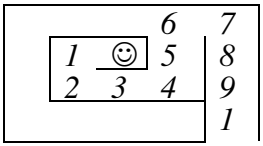
parcurgere exhaustivă a nodurilor sale, prin una din metodele BFS sau DFS, să păstrăm numai acele noduri ce verifică în momentul "vizitării" condiţia impusă (lipsa lui *x* de pe primele două poziţii).

Observăm că această metodă necesită folosirea în scopul memorării dinamice a drumului parcurs (în timpul căutării soluţiei) a mecanismului de *stivă*, fapt sugerat chiar de numele ei: tracking, adică înregistrarea pistei parcurse. Acest mecanism de stivă, care permite atît memorarea pistei cît şi revenirea – back-tracking-ul, este unul din mecanismele de bază ce este folosit pe scară largă în procedurile de gestiune dinamică a datelor în memorie. În plus, există unele cazuri particulare de probleme în care soluţia căutată se obţine în final prin "vărsarea" întregului conţinut al stivei şi nu doar prin "nodul" ultim vizitat, aflat în vîrful stivei.

Exemplul cel mai potrivit de problemă ce necesită o strategie de rezolvare backtracking este *Problema Labirintului*: se cere să se indice, pentru o configuraţie labirintică dată, traseul ce conduce către ieşirea din labirint. Iată un exemplu sugestiv:



Observaţi cum, după 15 paşi, este necesară o revenire (backtracking) pînă la căsuţa 6, de unde se continuă pe o altă pistă. "Pista falsă" a fost memorată în stivă, element cu element, iar revenirea se va realiza prin eliminarea din stivă tot element cu element. Cînd în vîrful stivei reapare căsuţa cu numărul 6, stiva începe din nou să crească memorînd elementele noului drum. În final stiva conţine în întregime soluţia: drumul corect către ieşirea din labirint.



În consecinţă, indiferent de forma particulară ce o poate lua sau de modul în care este "citită" în final soluţia, esenţialul constă în faptul că *backtracking-ul este o metodă de programare ce conţine obligatoriu gestiune de stivă*. Lipsa instrucţiunilor, explicite sau "transparente", de gestionare a stivei într-un program (de exemplu, lipsa apelului recursiv), este un indiciu sigur de recunoaştere a faptului că acel algoritm nu foloseşte metoda sau strategia de rezolvare BackTracking.

Tot o metodă back-tracking este şi metoda de programare cunoscută sub numele *programare recursivă.* Ea este mai utilizată decît metoda clasică BackTracking, fiind mai economicoasă din punctul de vedere al minimizării efortului de programare. Această metodă se reduce la construirea, în mod transparent pentru programator, a arborelui apelurilor recursive, traversarea acestuia prin apelarea recursivă (repetată) şi efectuarea acţiunilor corespunzătoare în momentul "vizitării" fiecărui nod al arborelui. Apelarea recursivă constituie "motorul vehiculului" de traversare şi are doar rolul de a permite traversarea arborelui. Gestionarea stivei apelurilor recursive şi revenirea - back-tracking-ul rămîne în sarcina mediului de programare folosit şi se efectuează într-un mod mascat pentru programator. Din acest punct de vedere, programatorului îi revine sarcina scrierii corecte a instrucţiunii de apel recursiv şi a instrucţiunii ce "scurt-circuitează" bucla infinită a apelurilor recursive. Singurele instrucţiuni care "fac treabă", în sensul rezolvării propriuzise a problemei respective, sînt cele cuprinse în corpul procedurii.

De exemplu, iată cum arată în limbajul de programare Pascal procedura generală de generare a permutărilor în varianta recursivă şi arborele de generare a permutărilor mulţimii {1,2,3} (*n=3*), pe nivele:

```
Procedure Permut(k:byte;s:string);
{ k – nivelul în arbore, s - şirul}
Var i:byte;tmp:char; Begin
 If k=n then begin    { scurt-circuitarea recursivităţii}
  For i:=1 to n do Write(s[i]);
                                   { prin afişarea permutării      }
   Write(';');              { urmată de un punct-virgulă     }
  end else
    For i:=k to n do begin
                        { singurele instrucţiuni "ce fac treabă" }
     tmp:=s[i];s[i]:=s[k];s[k]:=tmp;   { sînt for-ul şi cele trei
atribuiri }
       Permut(k+1,s);
                         { apelul recursiv ce permite parcugerea  }
     end;
                         { arbor. de generare a celor n! permutări}
End;
```

Nivelele arborelui (răsturnat pe orizontală)

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

2 ---- 3 Fiecare nivel al arborelui corespunde unei poziţii în şirul permutărilor. Astfel, pe prima

1 <

3 ---- 2 poziţie (nivelul 1) pot fi oricare din cele trei elemente: 1, 2, 3. Pe poziţia următoare pot

/

1 ---- 3 fi oricare din celelalte două elemente rămase: 2, 3; 1, 3; 1, 2. Pe al treilea nivel şi ultimul

Start -- 2 <

3 ---- 1 vor fi numai elementele rămase (cîte unul). Generarea permutărilor constă în construirea

\

1 ---- 2 şi parcurgerea arborelui permutărilor: odată ajunşi cu parcurgerea la un capăt din dreapta

3 <

2 ---- 1 vom afişa de fiecare dată "drumul" de la "rădăcină" la "frunza" terminală.

Observăm că arborele permutărilor este identic cu arborele apelurilor recursive şi că controlul şi gestiunea stivei se face automat, transparent faţă de programator. Instrucţiunilor de control (din background) le revine sarcina de a păstra şi de a memora, de la un apel recursiv la altul, string-ul *s* ce conţine permutările. Deşi această procedură recursiv de generare a permutărilor pare o variantă de procedură simplă din punctul de vedere al programatorului, în realitate, ea conţine într-un mod ascuns efortul de gestionare a stivei: încărcarea-descărcarea stringului *s* şi a întregului *k*. Acest efort este preluat în întregime de instrucţiunile incluse automat de mediul de programare pentru realizarea recursivităţii.

Avantajul metodei back-tracking este faptul că efortul programatorului se reduce la doar trei sarcini:

1. "construirea" grafului particular de căutare a soluţiilor
2. adaptarea corespunzătoare a uneia din metodele generale de traversare-vizitare a grafului în situaţia respectivă (de exemplu, prin apel recursiv)
3. adăugarea instrucţiunilor "ce fac treabă" care, fiind apelate în mod repetat în timpul vizitării nodurilor (grafului soluţiilor posibile), rezolvă gradat problema, găsind sau construind soluţia.

Acţiunea de revenire ce dă numele metodei, *backtracking - revenire pe "pista lăsată"*, este inclusă şi este efectuată de subalgoritmul de traversare a grafului soluţiilor posibile. Acest subalgoritm are un caracter general şi face parte din "zestrea" oricărui programator. În cazul particular în care graful soluţiilor este arbore, atunci se poate aplica întotdeauna cu succes metoda

programării recursive care conduce la un cod-program redus şi compact.

Prezentăm din nou procedura generală DepthFirstSearch (DFS) *de traversare a unui graf în varianta recursivă (ce "construieşte" de fapt arborele de traversare a grafului avînd ca rădăcină nodul de pornire) pentru a pune în evidenţă cele spuse.*

Procedura DFS(v:nod);

Vizitează v;

{ aici vor fi acele instrucţiuni "care fac treabă" }

Marchează v; // v devine un nod vizitat // { poate să lipsească în anumite implementări }

Cît timp (există w nemarcat nod adiacent lui v)

execută **DFS**(w); { apelul recursiv este "motorul vehiculului" }

{ ce permite parcurgerea grafului şi gestiunea stivei de revenire }

Există situaţii în care, la unele probleme, putem întîlni soluţii tip-backtracking fără însă a se putea sesiza la prima vedere prezenţa grafului de căutare asociat şi acţiunea de traversare a acestuia, ci doar prezenţa stivei. O privire mai atentă însă va conduce obligatoriu la descoperirea arborelui de căutare pe graful soluţiilor, chiar dacă el există doar într-o formă mascată. Acest fapt este inevitabil şi constituie esenţa metodei – căutare (generare) cu revenire pe pista lăsată.

Back-tracking-ul, metodă generală şi cu o largă aplicabilitate, fiind reductibilă în ultimă instanţă la traversarea spaţiului -grafului de căutare- a soluţiilor, are marele avantaj că determină cu certitudine toate soluţiile posibile, cu condiţia ca graful asociat de căutare a soluţiilor să fie corect. Dar ea are marele dezavantaj că necesită un timp de execuţie direct proporţional cu numărul nodurilor grafului de căutare asociat (sau numărul cazurilor posibile). În cele mai multe cazuri acest număr este exponenţial (e^n) sau chiar mai mare, factorial ($n!$),

unde *n* este dimensiunea vectorului datelor de intrare. Acest fapt conduce la o durată de execuţie de mărime astronomică făcînd într-un astfel de caz algoritmul complet inutilizabil, chiar dacă el este corect teoretic. (De exemplu, dacă soluţionarea problemei ar necesita generarea tuturor celor *100!* permutări (*n=100*), timpul de execuţie al algoritmului depăşeşte orice imaginaţie.) În astfel de situaţii, în care dimensiunea spaţiului de căutare-generare a soluţiilor are o astfel de dependenţă în funcţie de *n* (fiind o funcţie de ordin mai mare decît funcţia polinomială), este absolut necesară îmbunătăţirea acestei metode sau înlocuirea ei. Nu este însă necesară (şi de multe ori nici nu este posibilă!) abandonarea modelului abstract asociat - graful soluţiilor posibile, cu calităţile şi proprietăţile sale certe - ci este necesară doar obţinerea unei durate de execuţie de un ordin de mărime inferior printr-o altă strategie de parcurgere a spaţiului de căutare.

Greedy.

În strategia backtracking căutarea soluţiei, adică vizitarea secvenţială a nodurilor grafului soluţiilor cu revenire pe urma lăsată, se face oarecum "orbeşte" sau rigid, după o regulă simplă care să poată fi rapid aplicată în momentul "părăsirii" unui nod vizitat. În cazul metodei (strategiei) greedy apare suplimentar ideea de a efectua în acel moment o alegere. Dintre toate nodurile următoare posibile de a fi vizitate sau dintre toţi paşii următori posibili, se alege acel nod sau pas care asigură un maximum de "cîştig", de unde şi numele metodei: *greedy = lacom*. Evident că în acest fel poate să scadă viteza de vizitare a nodurilor – adică a soluţiilor ipotetice sau a soluţiilor parţiale – prin adăugarea duratei de execuţie a subalgoritmului de alegere a următorului nod după fiecare vizitare a unui nod. Există însă numeroşi algoritmi de tip greedy veritabili care nu conţin subalgoritmi de alegere. Asta nu înseamnă că au renunţat la alegerea greedy ci, datorită "scurtăturii" descoperite în timpul etapei de analiză a problemei, acei algoritmi efectuează la fiecare pas o alegere fără efort şi în mod optim a pasului (nodului)

următor. Această alegere, **dedusă în etapa de analiză**, conduce la maximum de "profit" pentru fiecare pas şi scurtează la maximum drumul către soluţia căutată.

Aparent această metodă de căutare a soluţiei este cea mai eficientă, din moment ce la fiecare pas se trece dintr-un optim (parţial) într-altul. Totuşi, ea nu poate fi aplicată în general ci doar în cazul în care există certitudinea alegerii optime la fiecare pas, certitudine rezultată în urma etapei anterioare de analiză a problemei. Ori, dezavantajul este că, la majoritatea problemelor dificile, etapa de analiză nu poate oferi o astfel de "pistă optimă" către soluţie. Un alt dezavantaj al acestei strategii este că nu poate să conducă către toate soluţiile posibile ci doar către soluţia optimă (din punct de vedere a alegerii efectuate în timpul căutării soluţiei), dar poate oferi toate soluţiile optime echivalente.

Programarea dinamică.

Este o metodă sau strategie ce îşi propune să elimine dezavantajele *metodei recursive* care, în ultimă instanţă, am văzut că se reduce la parcurgerea în adîncime a arborelui apelurilor recursive (adică backtracking). Această metodă se apropie ca idee strategică de metoda Greedy, avînd însă unele particularităţi.

Pentru a o înţelege este necesară evidenţierea dezavantajului major al recursivităţii. El constă din creşterea exagerată şi nerentabilă a efortului de execuţie prin repetarea ineficientă a unor paşi. Urmărind arborele apelurilor recursive se observă repetarea inutilă a unor cazuri rezolvate anterior, calculate deja înainte pe altă ramură a arborelui. Metodă eminamente iterativă, programarea dinamică elimină acest dezavantaj prin "răsturnarea" procedeului de obţinere a soluţiei şi implicit a arborelui apelurilor recursive. Printr-o abordare *bottom-up* (de la bază spre vîrf) ea reuşeşte să elimine operaţiile repetate inutil în cazul abordării *top-down* (de la vîrf spre bază).

Cu toţii am învăţat că, dacă vrem să calculăm "cu mîna" o combinare sau un tabel al combinărilor, în loc să calculăm de fiecare dată combinări de *n* luate cîte *k* pe baza definiţiei recursive: *C(n,k)=C(n-1,k-1)+C(n-1,k) cînd n,k>0, sau, C(n,k)=1 cînd k=0 sau n=k*, este mult mai eficient să construim *Triunghiul lui Pascal*, pornind de la aceeaşi definiţie a combinărilor.

C(4,2)
C(3,1) + C(3,2)
C(2,0) + C(2,1) C(2,1) + C(2,2)
1 **C(1,0)** + ***C(1,1)*** **C(1,0)** + ***C(1,1)*** 1
1 1 1 1

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
……..

Observaţi cum în arborele apelurilor recursive apar apeluri în mod repetat pentru calculul aceleaşi combinări. Acest efort repetat este evitat prin calcularea triunghiului lui Pascal în care fiecare combinare va fi calculată o singură dată.

În mod asemănător, aceeaşi diferenţă de abordare va exista între doi algoritmi de soluţionare a aceleaşi probleme, unul recursiv – backtracking - şi altul iterativ - proiectat prin metoda programării dinamice.

Dezavantajele acestei metode provin din faptul că, pentru a ţine minte paşii gata calculaţi şi a evita repetarea calculării lor (în termeni de grafuri, pentru a evita calcularea repetată a unor noduri pe ramuri diferite ale arborelui apelurilor recursive), este nevoie de punerea la dispoziţie a extra-spaţiului de memorare necesar şi de un efort suplimentar dat de gestiunea de memorie suplimentară.

Branch & Bound.

Este strategia cea mai sofisticată de proiectare a algoritmilor. Ea a apărut datorită existenţei problemelor pentru care soluţia de tip backtracking poate necesita un timp astronomic de rulare a programului. În rezolvarea acestor probleme apare o asemenea penurie de informaţii încît modelul abstract asociat problemei - graful de căutare a soluţiilor – nu poate fi precizat în avans, din etapa de analiză. Singura soluţie care rămîne este includerea unui subalgoritm suplimentar ce permite construirea acestui graf pe parcurs, din aproape în aproape. Apariţia acelui subalgoritm suplimentar dă numele metodei: *branch&bound*.

Este posibilă compararea algoritmului branch&bound cu un robot ce învaţă să se deplaseze singur şi eficient printr-un labirint. Acel robot va fi obligat să-şi construiască în paralel cu căutarea ieşirii o hartă (un graf !) a labirintului pe care va aplica apoi , pas cu pas, metode eficiente de obţinere a drumului cel mai scurt.

La strategia de căutare a soluţiei în spaţiul (graful) de căutare - backtracking, fiecare pas urma automat unul după altul pe baza unei reguli încorporate, în timp ce la strategia greedy alegerea pasului următor era făcută pe baza celei mai bune alegeri. În cazul acestei strategii – branch&bound, pentru pasul următor algoritmul nu mai este capabil să facă vreo alegere pentru că este obligat mai întîi să-şi determine singur nodurile vecine ce pot fi vizitate. Numele metodei, *branch=ramifică* şi *bound=delimitează*, provine de la cele două acţiuni ce ţin locul acţiunii de alegere de la strategia Greedy. Prima acţiune este construirea sau determinarea prin ramificare a drumurilor de continuare, iar a doua este eliminarea continuărilor (ramurilor) ineficiente sau eronate. Prin eliminarea unor ramuri, porţiuni întregi ale spaţiului de căutare a soluţiei rămînînd astfel dintr-o dată delimitate şi “izolate”. Această strategie de delimitare din mers a anumitor “regiuni” ale spaţiului de căutare a soluţiilor

este cea care permite reducerea ordinului de mărime a acestui spaţiu. Soluţia aceasta este eficientă doar dacă cîştigul oferit prin reducerea spaţiului de căutare (scăzînd *efortul suplimentar* depus pentru determinarea şi eliminarea din mers a continuărilor ineficiente) este substanţial.

Soluţiile de tip backtracking, avînd la bază un schelet atît de general (algoritmul de traversare a grafului de căutare a soluţiilor) sînt relativ simplu de adaptat în rezolvarea unor probleme. Poate acesta este motivul care a condus pe unii programatori lipsiţi de experienţă la convingerea falsă că *"Orice este posibil de rezolvat prin backtracking"*.

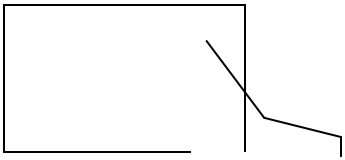
La ora actuală, lista problemelor pentru care nu se cunosc decît soluţii exponenţiale, total nerezonabile ca durată de execuţie a programului de soluţionare, cuprinde cîteva sute de probleme, una mai celebră ca cealaltă. Reamintim doar de "banala" (dar agasanta) Problemă a Orarului unei instituţii de învăţămînt care nu admite o soluţie backtracking datorită duratei astronomice de aşteptare a soluţiei.

Datorită totalei lor ineficienţe în execuţie, soluţiile backtracking obţinute după o analiză şi o proiectare "la prima mînă" (*brute-force approach,* în limba engleză) ajung să fie reanalizate din nou cu mai multă atenţie. Se constată atunci că modelul abstract asociat problemei, fie este prea sărac în informaţii pentru determinarea grafului de căutare a soluţiilor, fie conduce la un graf de căutare avînd dimensiunea nerezonabilă (exponenţială sau factorială, faţă de dimensiunea *n* a vectorului de intrare). Singura soluţie care rămîne în această situaţie la dispoziţie este ca aceste soluţii să fie reproiectate prin metoda branch&bound.

Un exemplu uşor de înţeles de "problemă branch&bound" îl oferă Problema Generală a Labirintului. Spre deosebire de Problema Labirintului prezentată anterior (care admitea o soluţie de tip backtracking), în varianta extinsă a acestei probleme, numărul direcţiilor posibile de urmat la fiecare pas

poate fi oricît de mare, iar obstacolele pot avea orice formă şi dimensiune. În acest caz, singura posibilitate este construirea *"din mers"* a spaţiului de căutare a soluţiei. Astfel, pentru determinarea unui drum de ieşire din labirint sau a drumului cel mai scurt (dacă este posibilă determinarea acestuia în timp rezonabil!) este obligatorie adoptarea strategiei branch&bound.

Oferim în continuare o situaţie concretă, ilustrată. Sesizaţi că obstacolele, avînd forme şi dimensiuni diferite, nu pot fi ocolite decît pe un traseu "razant" sau pe un traseu ce urmează contorul exterior al acestora. Acest fapt complică mult problema şi impune luarea unor decizii "la faţa locului", în momentul întîlnirii şi ocolirii fiecărui obstacol, ceea ce *impune* o strategie de rezolvare de tip branch&bound – ramifică şi delimitează:



Deşi această strategie poate să crească uneori surprinzător de mult eficienţa algoritmilor de soluţionare (din nerezonabili ca timp de execuţie ei pot ajunge rezonabili, datorită reducerii dimensiunii exponenţiale a spaţiului de căutare a soluţiei), aplicarea ei este posibilă doar printr-un efort suplimentar în etapa de analiză şi în cea de proiectare a algoritmului. Dezavantajul major al acestei metode constă deci în efortul major depus în etapa de analiză a problemei (analiză care însă se va face o singură dată şi bine!) şi efortul suplimentar depus în etapa proiectării algoritmului de soluţionare.

Din experienţa practică este cunoscut faptul că, pentru a analiza o problemă dificilă un analist poate avea nevoie de săptămîni sau chiar luni de zile de analiză, în timp ce algoritmul de soluţionare proiectat va dura, ca timp de execuţie, doar cîteva zeci de minute. Dacă programul obţinut nu este necesar a fi rulat decît o dată, aceasta este

prea puţin pentru “a se amortiza” costul mare al analizei şi proiectării sale. În acea situaţie, soluţia branch&bound este nerentabilă şi, probabil că ar fi mai ieftină strategia backtracking de soluţionare, chiar şi cu riscul de a obţine o execuţie (singura de altfel) a programului cu durata de o săptămînă (ceea ce poate să însemne totuşi economie de timp).

Bibliografie, adrese şi locaţii de interes pe Internet

Internetul e foarte mare, stufos şi, de multe ori, labirintic. Tocmai de aceea, ne-am gîndit să venim în ajutorul celor foarte pasionaţi de informatică şi de matematica aplicată în informatică. Oferim în continuare doar cîteva adrese pe care şi noi le-am utilizat cu succes. Fiecare din aceste site-uri conţine la rîndul lui liste de adrese şi legături (links) către alte site-uri cu subiecte asemănătoare. Iată, aveţi la dispoziţie "un capăt al ghemului" !

- www-groups.dcs.st-and.ac.uk/~history/ - conţine multe pagini interesante despre istoria descoperirilor în matematică, utile celor care doresc să afle cum se face cu adevărat descoperiri în matematică şi cum s-a ajuns la necesitatea apariţiei calculatoarelor
- www.mathpages.com/KsBrown/ - conţine o colecţie impresionantă de informaţii, idei şi descoperiri de ultimă oră din matematică şi informatică
- www.mathsoft.com/asolve/ - conţine o listă substanţială de probleme de matematică (şi nu numai) care îşi aşteaptă încă rezolvarea, multe dintre ele putînd fi abordate cu ajutorul calculatorului
- www.ee.Surrey.ac.uk/Personal/R.Knott/Fibonacci/fib.html - este o "portiţă" de intrare în domeniul fascinant al numerelor lui Fibonacci, cu multiple corelaţii matematice şi informatice
- mans.cee.hw.ac.uk/ctl.html *Computer Teaching and Learning Resources* - numele site-ului spune totul
- www.k12tlc.net/Penrose/ *K-12 Teaching & Learning Center* - noi am ales pagina care prezintă biografia lui Sir Roger Penrose, dar aveţi încă multe altele la dispoziţie

- www.ioccc.org *The International Obfuscated C Code Contest (IOCCC)* – Concursul internaţional de programare C ofuscată (încîlcită şi intenţionat confuză)

Suplimentar, tot pentru cei foarte pasionaţi de matematică, informatică, de legătura dintre ele şi nu numai, oferim o selecţie minimală de cărţi şi articole care au constituit, direct sau indirect, o sursă de inspiraţie în scrierea acestei culegeri:

- *Turbo Pascal 6.0. Ghid de utilizare*, Microinformatica, Cluj-Napoca, 1992
- **Bălănescu T. ...,** *Limbajul Turbo Pascal*, Editura tehnică, Bucureşti, 1992
- **Grigore Albeanu,** *Programarea în Pascal şi Turbo Pascal. Culegere de probleme*, Editura tehnică, Bucureşti, 1994
- **Tudor Sorin,** *Tehnici de programare,* Editura L&S Infomat, Bucureşti, 1998
- *Manual de programare C,* (după **Kernigham** şi **Ritchie**) Microinformatica, Cluj-Napoca, 1986
- **Muşlea I.**, *Programarea în C,* Microinformatica, Cluj-Napoca, 1992
- **Roger Penrose,** *Mintea noastră...cea de toate zilele*, (titlul original: *Emperor's mind*), Editura tehnică, Bucureşti, 2001
- **Roger Penrose,** *Incertitudinile raţiunii. Umbrele minţii*, (titlul original: *Shadows of the mind*), Editura tehnică, Bucureşti, 2000
- **Keith Devlin,** *Vîrsta de aur a matematicii*, (titlul original: *Matemathics: The New Golden Age*), Editura Thetha, Bucureşti, 2000
- **Solomon Marcus,** *Gîndirea algoritmică*, Editura tehnică, Bucureşti, 1982
- **L. Livovschi, H. Georgescu**, *Bazele informaticii*, Editura didactică şi pedagogică, Bucureşti, 1981