INTRODUCERE

1. Diferenţe între informatică şi matematică

Vom prezenta numai două aspecte ce deosebesc cele două ştiinţe:

1) *În lucrul pe calculator, majoritatea proprietăţilor algebrice nu sunt satisfăcute*.

- *elementul neutru*: egalitatea `a+b=a` poate fi satisfăcută fără ca `b=0`: este situaţia în care $b\neq 0$, dar ordinul său de mărime este mult mai mic decât al lui `a`, astfel încât memorarea unui număr limitat de cifre şi aducerea la acelaşi exponent fac ca, la efectuarea adunării, `b` să fie considerat egal cu 0.

- *comutativitate*: să considerăm următorul program Java:

```
class Zero {
  public static void main(String[] ppp) {
    C Ob = new C();
    System.out.println(Ob.a + Ob.met());
  }
}

class  C {
  int a=1;
  int met() { return a++; }
}
```

La ieşire este produsă valoarea 2. Dacă schimbăm însă ordinea termenilor din instrucţiunea de scriere, rezultatul va fi 3.

- *asociativitate*: pentru simplificare (dar fără a reduce din generalitate) vom presupune că numerele sunt memorate cu o singură cifră semnificativă şi că la înmulţire se face trunchiere. Atunci rezultatul înmulţirilor `(0.5×0.  7)×0.9` este `0.3×0.9=0.2`, pe când rezultatul înmulţirilor `0.5×(0.7×0.9)` este `0.5×0.6=0.3`.

2) *Nu interesează în general demonstrarea teoretică a existenţei unei soluţii, ci accentul este pus pe elaborarea algoritmilor*.

Vom pune în vedere acest aspect prezentând o elegantă demonstraţie a următoarei propoziţii:

Propoziţie. Există $\alpha,\beta\in\mathbf{R}\setminus\mathbf{Q}$ cu $\alpha^{\beta}\in\mathbf{Q}$.

Pentru demonstraţie să considerăm numărul real $x=a^a$, unde $a=\sqrt{2}$.

Dacă $x\in\mathbf{Q}$, propoziţia este demonstrată.

Dacă $x\notin\mathbf{Q}$, atunci $x^a=a^2=2\in\mathbf{Q}$ şi din nou propoziţia este demonstrată.

Frumuseţea acestei scurte demonstraţii nu poate satisface pe un informatician, deoarece lui i se cere în general să furnizeze un rezultat şi nu să arate existenţa acestuia.

2. Aspecte generale care apar la rezolvarea unei probleme

Aşa cum am spus, informaticianului i se cere *să elaboreze un algoritm* pentru o problemă dată, care să furnizeze o soluţie (fie şi aproximativă, cu condiţia menţionării acestui lucru).

Teoretic, paşii sunt următorii:

1) demonstrarea faptului că este posibilă elaborarea unui algoritm pentru determinarea unei soluţii;
2) elaborarea unui algoritm (caz în care pasul anterior devine inutil);
3) demonstrarea corectitudinii algoritmului;
4) determinarea timpului de executare a algoritmului;
5) demonstrarea optimalităţii algoritmului (a faptului că timpul de executare este mai mic decât timpul de executarea al oricărui alt algoritm pentru problema studiată).

Evident, acest scenariu este idealist, dar el trebuie să stea în permanenţă în atenţia informaticianului.

În cele ce urmează, vom schiţa câteva lucruri legate de aspectele de mai sus, nu neapărat în ordinea menţionată.

3. Timpul de executare a algoritmilor

Un algoritm este elaborat nu numai pentru un set de date de intrare, ci pentru o mulţime de astfel de seturi. De aceea trebuie bine precizată mulţimea (seturilor de date) de intrare. Timpul de executare se măsoară în funcţie de lungimea `n` a datelor de intrare.

Ideal este să determinăm o formulă matematică pentru `T(n)`=timpul de executare pentru orice set de date de intrare de lungime `n`. Din păcate, acest lucru nu este în general posibil. De aceea, în majoritatea cazurilor ne mărginim la a evalua *ordinul de mărime* al timpului de executare.

Mai precis, spunem că timpul de executare este de ordinul `f(n)` şi exprimăm acest lucru prin `T(n)=O(f(n))`, dacă raportul între `T(n)` şi `f(n)` tinde la un număr real atunci când `n` tinde la ∞.

De exemplu, dacă $T(n) = nt_0+a \cdot n \cdot \log_2 n$, atunci $T(n)=0(n \cdot \log n)$.

Spunem că algoritmul este *polinomial* dacă $f(n)=n^k$ pentru un anumit număr k,

Specificăm, fără a avea pretenţia că dăm o definiţie, că un algoritm se numeşte "acceptabil" dacă este este polinomial. În capitolul referitor la metoda backtracking este prezentat un mic tabel care scoate în evidenţă faptul că algoritmii exponenţiali necesită un timp de calcul mult prea mare chiar pentru un `n` mic şi indiferent de performanţele calculatorului pe care lucrăm.

De aceea, în continuare accentul este pus pe prezentarea unor algoritmi polinomiali.

4. Corectitudinea algoritmilor

În demonstrarea corectitudinii algoritmilor, există două aspecte importante:

- *Corectitudinea parţială*: presupunând că algoritmul se termină (într-un număr finit de paşi), trebuie demonstrat că rezultatul este corect;

- *Terminarea programului*: trebuie demonstrat că algoritmul se încheie în timp finit.

Evident, condiţiile de mai sus trebuie îndeplinite pentru *orice* set de date de intrare admis.

Modul tipic de lucru constă în introducerea în anumite locuri din program a unor *invarianţi*, adică relaţii ce sunt îndeplinite la orice trecere a programului prin acele locuri.

Ne mărginim la a prezenta două exemple simple.

Exemplul 1. Determinarea concomitentă a celui mai mare divizor comun şi a celui mai mic multiplu comun a două numere naturale.

Fie $a,b\in\mathbf{N}^*$. Se caută:

- (a,b)=cel mai mare divizor comun al lui a şi b;
- $[a,b]$=cel mai mic multiplu comun al lui a şi b.

Algoritmul este următorul:

```
x ← a; y ← b; u ← a; v ← b;
while x≠y
   { xv+yu = 2ab; (x,y)=(a,b) }                    (*)
   if x>y then x ← x-y; u ← u+v
          else y ← y-x; v ← u+v
write(x,(u+v)/2)
```

Demonstrarea corectitudinii se face în trei paşi:

1) $(*)$ *este invariant*:

La prima intrare în ciclul `while`, condiţia este evident îndeplinită.

Mai trebuie demonstrat că dacă relaţiile (*) sunt îndeplinite şi ciclul se reia, ele vor fi îndeplinite şi după reluare.

Fie (x,y,u,v) valorile curente la o intrare în ciclu, iar (x',y',u',v') valorile curente la următoarea intrare în ciclul `while`. Deci: $xv+yu=2ab$ şi $(x,y)=(a,b)$.

Presupunem că $x>y$. Atunci $x'=x-y$, $y'=y$, $u'=u+v$, $v'=v$.

$x'v'+y'u'=(x-y)v+y(u+v)=xv+yu=2ab$ şi

$(x',y')=(x-y,y)=(x,y)=(a,b)$.

Cazul $x<y$ se studiază similar.

2) *Corectitudinea parţială*:

La ieşirea din ciclul `while`, $x=(x,x)=(x,y)=(a,b)$. Notăm $d=(a,b)=x$. Conform relaţiilor $(*)$, avem $d(u+v)=2\alpha\beta d^2$, unde $a=\alpha d$ şi $b=\beta d$. Atunci $(u+v)/2=\alpha\beta d=ab/d=[a,b]$.

3) *Terminarea programului*:

Fie $\{x_n\}$, $\{y_n\}$, $\{u_n\}$, $\{v_n\}$ şirul de valori succesive ale variabilelor. Toate aceste valori sunt numere naturale pozitive. Se observă că şirurile $\{x_n\}$ şi $\{y_n\}$ sunt descrescătoare, iar şirul $\{x_n+y_n\}$ este *strict* descrescător. Aceasta ne asigură că după un număr finit de paşi vom obţine $x=y$.

Exemplul 2. Metoda de înmulţire a ţăranului rus.

Fie $a,b \in \mathbf{N}$. Se cere să se calculeze produsul ab.

Ţăranul rus ştie doar:

- să verifice dacă un număr este par sau impar;
- să adune două numere;
- să afle câtul împărţirii unui număr la 2;
- să compare un număr cu 0.

Cu aceste cunoştinţe, ţăranul rus procedează astfel:

```
x ← a; y ← b; p ← 0
while x>0
   { xy+p=ab }                                    (*)
   if x impar then p ← p+y
   x ← x div 2; y ← y+y
write(p)
```

Să urmărim cum decurg calculele pentru $x=54$, $y=12$:

x	y	p
54	12	0
27	24	
13	48	24
6	96	72
3	192	
1	384	264
0	?	648

Ca şi pentru exemplul precedent, demonstrarea corectitudinii se face în trei paşi:

1) $(*)$ *este invariant*:

La prima intrare în ciclul `while`, relaţia este evident îndeplinită.

Mai trebuie demonstrat că dacă relaţia (*) este îndeplinită şi ciclul se reia, ea va fi îndeplinită şi la reluare.

Fie (x,y,p) valorile curente la o intrare în ciclu, iar (x',y',p') valorile curente la următoarea intrare în ciclul `while`. Deci: $xy+p=ab$.

Presupunem că x este impar. Atunci $(x',y',p')=((x-1)/2,2y,p+y)$. Rezultă $x'y'+p'=(x-1)/2 \cdot 2y+p+y=xy+p=ab$.

Presupunem că x este par. Atunci $(x',y',p')=(x/2,2y,p)$. Rezultă $x'y'+p'=xy+p=ab$.

2) *Corectitudinea parţială*:

Dacă programul se termină, atunci $x=0$, deci $p=ab$.

3) *Terminarea programului*:

Fie $\{x_n\}$, $\{y_n\}$ şirul de valori succesive ale variabilelor corespunzătoare.

Se observă că şirul $\{x_n\}$ este *strict* descrescător. Aceasta ne asigură că după un număr finit de paşi vom obţine $x=0$.

5. Optimalitatea algoritmilor

Să presupunem că pentru o anumită problemă am elaborat un algoritm şi am calculat timpul său de executare $T(n)$. Este natural să ne întrebăm dacă algoritmul nostru este "cel mai bun" sau există un alt algoritm cu timp de executare mai mic.

Problema demonstrării optimalităţii unui algoritm este foarte dificilă, în mod deosebit datorită faptului că trebuie să considerăm *toţi* algoritmii posibili şi să arătăm că ei au un timp de executare superior.

Ne mărginim la a enunţa două probleme şi a demonstra optimalitatea algoritmilor propuşi, pentru a pune în evidenţă dificultăţile care apar.

Exemplul 3. Determinarea celui mai mic element al unui vector.

Se cere să determinăm $m=\min(a_1,a_2,\ldots,a_n)$.

Algoritmul binecunoscut este următorul:

```
m ← a1
for i=2,n
   if ai<m then m ← ai
```

care necesită $n-1$ comparări între elementele vectorului $a=(a_1,a_2,\ldots,a_n)$.

Propoziţia 1. Algoritmul de mai sus este optimal.

Trebuie demonstrat că orice algoritm bazat pe comparări necesită cel puţin $n-1$ comparări. Demonstrarea optimalităţii acestui algoritm se face uşor prin inducţie.

Pentru $n=1$ este evident că nu trebuie efectuată nici o comparare.

Presupunem că orice algoritm care rezolvă problema pentru n numere efectuează cel puţin $n-1$ comparări şi să considerăm un algoritm oarecare care determină cel mai mic dintre $n+1$ numere. Considerăm prima comparare efectuată de acest algoritm; fără reducerea generalităţii, putem presupune că s-au comparat a_1 cu a_2 şi că $a_1<a_2$. Atunci $m=\min(a_1,a_3,\ldots,a_{n+1})$. Dar pentru determinarea acestui minim sunt necesare cel puţin $n-1$ comparări, deci numărul total de comparări efectuat de algoritmul considerat este cel puţin egal cu n.

Exemplul 4. Determinarea minimului şi maximului elementelor unui vector.

Se cere determinarea valorilor:

$m=\min(a_1,a_2,\ldots,a_n)$ şi $M=\max(a_1,a_2,\ldots,a_n)$.

Determinarea succesivă a valorilor m şi M necesită timpul $T(n)=2(n-1)$.

O soluţie mai bună constă în a considera câte două elemente ale vectorului, a determina pe cel mai mic şi pe cel mai mare dintre ele, iar apoi în a compara pe cel mai mic cu minimul curent şi pe cel mai mare cu maximul curent:

```
if n impar then m ← a1; M ← a1; k ← 1
          else if a1<a2 then m ← a1; M ← a2
                        else m ← a2; M ← a1
               k ← 2
{ k = numărul de elemente analizate }
while k≤n-2
   if ak+1<ak+2 then if ak+1<m then m ← ak+1
                     if ak+2>M then M ← ak+2
                else if ak+2<m then m ← ak+2
                     if ak+1>M then M ← ak+1
```

$k \leftarrow k+2$

Să calculăm numărul de comparări efectuate:

- pentru $n=2k$, în faza de iniţializare se face o comparare, iar în continuare se fac $3(k-1)$ comparări; obţinem $T(n)=1+3(k-1)=3k-2=3n/2-2=\lceil 3n/2 \rceil-2$.
- pentru $n=2k+1$, la iniţializare nu se face nici o comparare, iar în continuare se fac $3k$ comparări; obţinem $T(n)=(3n-3)/2=(3n+1)/2-2=\lceil 3n/2 \rceil-2$.

În concluzie, timpul de calcul este $T(n)=\lceil 3n/2 \rceil-2$.

Propoziţia 2. Algoritmul de mai sus este optimal.

Considerăm următoarele mulţimi, ce formează o partiţie a mulţimii elementelor/jucătorilor, precum şi cardinalul lor:

- A= mulţimea elementelor care nu au participat încă la comparări; $a=|A|$;
- B= mulţimea elementelor care au participat la comparări şi au fost totdeauna mai mari decât elementele cu care au fost comparate; $b=|B|$;
- C= mulţimea elementelor care au participat la comparări şi au fost totdeauna mai mici decât elementele cu care au fost comparate; $c=|C|$;
- D= mulţimea elementelor care au participat la comparări şi au fost cel puţin o dată mai mari şi cel puţin o dată mai mici decât elementele cu care au fost comparate; $d=|D|$.

Numim *configuraţie* un quadruplu (a,b,c,d). Problema constă în determinarea numărului de comparări necesare pentru a trece de la quadruplul $(n,0,0,0)$ la quadruplul $(0,1,1,n-2)$.

Considerăm un algoritm arbitrar care rezolvă problema şi arătăm că el efectuează cel puţin $\lceil 3n/2 \rceil-2$ comparări.

Vom proceda astfel:

1) identificăm cazurile cele mai defavorabile;
2) pentru aceste cazuri determinăm numărul minim de comparări.

Să analizăm trecerea de la o configuraţie oarecare (a,b,c,d) la următoarea. Este evident că nu are sens să efectuăm comparări în care intervine vreun element din D. Apar următoarele situaţii posibile:

1) Compar două elemente din A: se va trece în configuraţia $(a-2,b+1,c+1,d)$.
2) Compar două elemente din B: se va trece în configuraţia $(a,b-1,c,d+1)$.
3) Compar două elemente din C: se va trece în configuraţia $(a,b,c-1,d+1)$.
4) Se compară un element din A cu unul din B. Sunt posibile două situaţii:
   - elementul din A este mai mic: se trece în configuraţia $(a-1,b,c+1,d)$;
   - elementul din A este mai mare: se trece în configuraţia $(a-1,b,c,d+1)$.

   Cazul cel mai defavorabil este primul, deoarece implică o deplasare "mai lentă" spre dreapta a componentelor quadruplului. De aceea vom lua în considerare acest caz.
5) Se compară un element din A cu unul din C. Sunt posibile două situaţii:
   - elementul din A este mai mic: se trece în configuraţia $(a-1,b,c,d+1)$;
   - elementul din A este mai mare: se trece în configuraţia $(a-1,b+1,c,d)$.

   Cazul cel mai defavorabil este al doilea, deoarece implică o deplasare "mai lentă" spre dreapta a componentelor quadruplului. De aceea vom lua în considerare acest caz.
6) Se compară un element din B cu unul din C. Sunt posibile două situaţii:
   - elementul din B este mai mic: se trece în configuraţia $(a,b-1,c-1,d+2)$;
   - elementul din B este mai mare: se rămâne în configuraţia (a,b,c,d).

Cazul cel mai defavorabil este al doilea, deoarece implică o deplasare "mai lentă" spre dreapta a componentelor quadruplului. De aceea vom lua în considerare acest caz.

Observaţie. Cazurile cele mai favorabile sunt cele în care `d` creşte, deci ies din calcul elemente candidate la a fi cel mai mic sau cel mai mare.

Odată stabilită trecerea de la o configuraţie la următoarea, ne punem problema cum putem trece mai rapid de la configuraţia iniţială la cea finală.

Analizăm cazul în care $n=2k$ (cazul în care n este impar este propus ca exerciţiu). Trecerea cea mai rapidă la configuraţia finală se face astfel:

- plecăm de la $(n,0,0,0)=(2k,0,0,0)$;
- prin k comparări între perechi de elemente din A ajungem la $(0,k,k,0)$;
- prin $k-1$ comparări între perechi de elemente din B ajungem la $(0,1,k,k-1)$;
- prin $k-1$ comparări între perechi de elemente din C ajungem la $(0,1,1,n-2)$.

În total au fost necesare $k+(k-1)+(k-1)=3k-2=\lceil 3n/2 \rceil-2$ comparări.

6. Existenţa algoritmilor

Acest aspect este şi mai delicat decât precedentele, pentru că necesită o definiţie matematică riguroasă a noţiunii de algoritm. Nu vom face decât să prezentăm (fără vreo demonstraţie) câteva definiţii şi rezultate. Un studiu mai amănunţit necesită un curs aparte!

Începem prin a preciza că problema existenţei algoritmilor a stat în atenţia matematicienilor încă înainte de apariţia calculatoarelor. În sprijinul acestei afirmaţii ne rezumăm la a spune că un rol deosebit în această teorie l-a jucat matematicianul englez Alan Turing (1912-1954), considerat părintele inteligenţei artificiale.

Deci *ce este un algoritm*?

Noţiunea de algoritm nu poate fi definită decât pe baza unui limbaj sau a unei maşini matematice abstracte.

Prezentăm în continuare o singură definiţie, care are la bază *limbajul S* care operează asupra numerelor naturale.

Un program în limbajul *S* foloseşte variabilele:

- $x_1, x_2, \ldots$ care constituie datele de intrare (nu există instrucţiuni de citire);
- y (în care va apărea rezultatul prelucrărilor);
- $z_1, z_2, \ldots$ care constituie variabile de lucru.

Variabilele $y, z_1, z_2, \ldots$ au iniţial valoarea 0.

Instrucţiunile pot fi etichetate (nu neapărat distinct) şi au numai următoarele forme, în care v este o variabilă, iar L este o etichetă:

- $v \leftarrow v+1$ { valoarea variabilei v creşte cu o unitate }
- $v \leftarrow v-1$ { valoarea variabilei v scade cu o unitate dacă era strict pozitivă }
- `if v>0 goto L` { se face transfer condiţionat la prima instrucţiune cu eticheta L, dacă o astfel de instrucţiune există; în caz contrar programul se termină }.

Programul se termină fie dacă s-a executat ultima instrucţiune din program, fie dacă se face transfer la o instrucţiune cu o etichetă inexistentă.

Observaţii:
- faptul că se lucrează numai cu numere naturale nu este o restricţie, deoarece în memorie există doar secvenţe de biţi (interpretate în diferite moduri);
- nu interesează timpul de executare a programului, ci numai existenţa sa;
- dacă rezultatul dorit constă din mai multe valori, vom scrie câte un program pentru calculul fiecăreia dintre aceste valori;
- programul vid corespunde calculului funcţiei identic egală cu 0: pentru orice $x_1, x_2, \ldots$ valoarea de ieşire a lui y este 0 (cea iniţială).

Este naturală o neîncredere iniţială în acest limbaj, dacă îl comparăm cu limbajele evoluate din ziua de azi. Se poate însă demonstra că în limbajul *S* se pot efectua calcule "oricât" de complexe asupra numerelor naturale.

***Teza lui Church* (1936).** *Date fiind numerele naturale* $x_1, x_2, \ldots, x_n$*, numărul* y *poate fi "calculat" pe baza lor dacă şi numai dacă există un program în limbajul S care pentru valorile de intrare* $x_1, x_2, \ldots, x_n$ *produce la terminarea sa valoarea* y.

Cu alte cuvinte, înţelegem prin ***algoritm*** ce calculează valoarea y plecând de la valorile $x_1, x_2, \ldots, x_n$ un program în limbajul *S* care realizează acest lucru.

Există mai multe definiţii ale noţiunii de algoritm, bazate fie pe calculul cu numere naturale fie pe calcul simbolic, folosind fie limbaje de programare fie maşini matematice, dar toate s-au dovedit a fi echivalente (cu cea de mai sus)!

Mai precizăm că orice program în limbajul *S* poate fi codificat ca un număr natural (în particular mulţimea acestor programe este numărabilă).

Numim *problemă nedecidabilă* o problemă pentru care nu poate fi elaborat un algoritm. Definirea matematică a noţiunii de algoritm a permis detectarea de probleme nedecidabile. Câteva dintre ele sunt următoarele:
1) *Problema opririi programelor*: pentru orice program şi orice valori de intrare să se decidă dacă programul se termină.
2) *Problema opririi programelor* (*variantă*): pentru un program dat să se decidă dacă el se termină pentru orice valori de intrare.
3) *Problema echivalenţei programelor*: să se decidă pentru orice două programe dacă sunt echivalente (produc aceeaşi ieşire pentru aceleaşi date de intrare).

În continuare vom lucra cu noţiunea de algoritm în accepţiunea sa uzuală, aşa cum în liceu (şi la unele facultăţi) se lucrează cu numerele reale, fără a se prezenta o definiţie riguroasă a lor.

Am dorit însă să evidenţiem că există probleme nedecidabile şi că (uimitor pentru unii) studiul existenţei algoritmilor a început înainte de apariţia calculatoarelor.