1. Preliminarii

1.1 Ce este un algoritm?

Abu Ja`far Mohammed ibn Musa al-Khowarizmi (autor persan, sec. VIII-IX), a scris o carte de matematica cunoscuta in traducere latina ca “Algorithmi de numero indorum”, iar apoi ca “Liber algorithmi”, unde “algorithm” provine de la “al-Khowarizmi”, ceea ce literal inseamna “din orasul Khowarizm”. In prezent, acest oras se numeste Khiva si se afla in Uzbechistan. Atat al-Khowarizmi, cat si alti matematicieni din Evul Mediu, intelegeau prin *algoritm* o regula pe baza careia se efectuau calcule aritmetice. Astfel, in timpul lui Adam Riese (sec. XVI), algoritmii foloseau la: dublari, injumatatiri, inmultiri de numere. Alti algoritmi apar in lucrarile lui Stifer (“Arithmetica integra”, Nürnberg, 1544) si Cardano (“Ars magna sive de reguli algebraicis”, Nürnberg, 1545). Chiar si Leibniz vorbeste de “algoritmi de inmultire”. Termenul a ramas totusi multa vreme cu o intrebuintare destul de restransa, chiar si in domeniul matematicii.

Kronecker (in 1886) si Dedekind (in 1888) semneaza actul de nastere al teoriei functiilor recursive. Conceptul de recursivitate devine indisolubil legat de cel de algoritm. Dar abia in deceniile al treilea si al patrulea ale secolului nostru, teoria recursivitatii si algoritmilor incepe sa se constituie ca atare, prin lucrarile lui Skolem, Ackermann, Sudan, Gödel, Church, Kleene, Turing, Peter si altii.

Este surprinzatoare transformarea gandirii algoritmice, dintr-un instrument matematic particular, intr-o modalitate fundamentala de abordare a problemelor in domenii care aparent nu au nimic comun cu matematica. Aceasta universalitate a gandirii algoritmice este rezultatul conexiunii dintre algoritm si calculator. Astazi, intelegem prin *algoritm* o metoda generala de rezolvare a unui anumit tip de problema, metoda care se poate implementa pe calculator. In acest context, un algoritm este esenta absoluta a unei rutine.

Cel mai faimos algoritm este desigur algoritmul lui Euclid pentru aflarea celui mai mare divizor comun a doua numere intregi. Alte exemple de algoritmi sunt metodele invatate in scoala pentru a inmulti/imparti doua numere. Ceea ce da insa generalitate notiunii de algoritm este faptul ca el poate opera nu numai cu numere. Exista astfel algoritmi algebrici si algoritmi logici. Pana si o reteta culinara este in esenta un algoritm. Practic, s-a constatat ca nu exista nici un domeniu, oricat ar parea el de imprecis si de fluctuant, in care sa nu putem descoperi sectoare functionand algoritmic.

Un algoritm este compus dintr-o multime finita de pasi, fiecare necesitand una sau mai multe operatii. Pentru a fi implementabile pe calculator, aceste operatii trebuie sa fie in primul rand *definite*, adica sa fie foarte clar ce anume trebuie executat. In al doilea rand, operatiile trebuie sa fie *efective*, ceea ce inseamna ca – in principiu, cel putin – o persoana dotata cu creion si hartie trebuie sa poata efectua orice pas intr-un timp finit. De exemplu, aritmetica cu numere intregi este efectiva. Aritmetica cu numere reale nu este insa efectiva, deoarece unele numere sunt exprimabile prin secvente infinite. Vom considera ca un algoritm trebuie sa se termine dupa un numar finit de operatii, intr-un timp rezonabil de lung.

Programul este exprimarea unui algoritm intr-un limbaj de programare. Este bine ca inainte de a invata concepte generale, sa fi acumulat deja o anumita experienta practica in domeniul respectiv. Presupunand ca ati scris deja programe intr-un limbaj de nivel inalt, probabil ca ati avut uneori dificultati in a formula solutia pentru o problema. Alteori, poate ca nu ati putut decide care dintre algoritmii care rezolvau aceeasi problema este mai bun. Aceasta carte va va invata cum sa evitati aceste situatii nedorite.

Studiul algoritmilor cuprinde mai multe aspecte:

i) *Elaborarea algoritmilor*. Actul de creare a unui algoritm este o arta care nu va putea fi niciodata pe deplin automatizata. Este in fond vorba de mecanismul universal al creativitatii umane, care produce noul printr-o sinteza extrem de complexa de tipul:

tehnici de elaborare (reguli) + creativitate (intuitie) = solutie.

Un obiectiv major al acestei carti este de a prezenta diverse tehnici fundamentale de elaborare a algoritmilor. Utilizand aceste tehnici, acumuland si o anumita experienta, veti fi capabili sa concepeti algoritmi eficienti.

ii) *Exprimarea algoritmilor*. Forma pe care o ia un algoritm intr-un program trebuie sa fie clara si concisa, ceea ce implica utilizarea unui anumit stil de programare. Acest stil nu este in mod obligatoriu legat de un anumit limbaj de programare, ci, mai curand, de tipul limbajului si de modul de abordare. Astfel, incepand cu anii ‘80, standardul unanim acceptat este cel de programare structurata. In prezent, se impune standardul programarii orientate pe obiect.

iii) *Validarea algoritmilor*. Un algoritm, dupa elaborare, nu trebuie in mod necesar sa fie programat pentru a demonstra ca functioneaza corect in orice situatie. El poate fi scris initial intr-o forma precisa oarecare. In aceasta forma, algoritmul va fi *validat*, pentru a ne asigura ca algoritmul este corect, independent de limbajul in care va fi apoi programat.

iv) *Analiza algoritmilor*. Pentru a putea decide care dintre algoritmii ce rezolva aceeasi problema este mai bun, este nevoie sa definim un criteriu de apreciere a valorii unui algoritm. In general, acest criteriu se refera la timpul de calcul si la memoria necesara unui algoritm. Vom analiza din acest punct de vedere toti algoritmii prezentati.

v) *Testarea programelor*. Aceasta consta din doua faze: depanare (debugging) si trasare (profiling). *Depanarea* este procesul executarii unui program pe date de test si corectarea eventualelor erori. Dupa cum afirma insa E. W. Dijkstra, prin depanare putem evidentia prezenta erorilor, dar nu si absenta lor. O demonstrare a faptului ca un program este corect este mai valoroasa decat o mie de teste, deoarece garanteaza ca programul va functiona corect in *orice* situatie. *Trasarea* este procesul executarii unui program corect pe diferite date de test, pentru a-i determina timpul de calcul si memoria necesara. Rezultatele obtinute pot fi apoi comparate cu analiza anterioara a algoritmului.

Aceasta enumerare serveste fixarii cadrului general pentru problemele abordate in carte: ne vom concentra pe domeniile *i)*, *ii)* si *iv)*.

Vom incepe cu un exemplu de algoritm. Este vorba de o metoda, cam ciudata la prima vedere, de inmultire a doua numere. Se numeste “inmultirea a la russe”.

Vom scrie deinmultitul si inmultitorul (de exemplu 45 si 19) unul langa altul, formand sub fiecare cate o coloana, conform urmatoarei reguli: se imparte numarul de sub deinmultit la 2, ignorand fractiile, apoi se inmulteste cu 2 numarul

45	19	19
22	38	—
11	76	76
5	152	152
2	304	—
1	608	608
		855

de sub inmultitor. Se aplica regula, pana cand numarul de sub deinmultit este 1. In final, adunam toate numerele din coloana inmultitorului care corespund, pe linie, unor numere impare in coloana deinmultitului. In cazul nostru, obtinem: 19 + 76 + 152 + 608 = 855.

Cu toate ca pare ciudata, aceasta este tehnica folosita de hardware-ul multor calculatoare. Ea prezinta avantajul ca nu este necesar sa se memoreze tabla de inmultire. Totul se rezuma la adunari si inmultiri/impartiri cu 2 (acestea din urma fiind rezolvate printr-o simpla decalare).

Pentru a reprezenta algoritmul, vom utiliza un limbaj simplificat, numit *pseudo-cod*, care este un compromis intre precizia unui limbaj de programare si usurinta in exprimare a unui limbaj natural. Astfel, elementele esentiale ale algoritmului nu vor fi ascunse de detalii de programare neimportante in aceasta faza. Daca sunteti familiarizat cu un limbaj uzual de programare, nu veti avea nici o dificultate in a intelege notatiile folosite si in a scrie programul respectiv.

Cunoasteti atunci si diferenta dintre o functie si o procedura. In notatia pe care o folosim, o functie va returna uneori un tablou, o multime, sau un mesaj. Veti intelege ca este vorba de o scriere mai compacta si in functie de context veti putea alege implementarea convenabila. Vom conveni ca parametrii functiilor (procedurilor) sa fie transmisi *prin valoare*, exceptand tablourile, care vor fi transmise *prin adresa primului element*. Notatia folosita pentru specificarea unui parametru de tip tablou va fi diferita, de la caz la caz. Uneori vom scrie, de exemplu:

procedure *proc*1(*T*)

atunci cand tipul si dimensiunile tabloului *T* sunt neimportante, sau cand acestea sunt evidente din context. Intr-un astfel de caz, vom nota cu #*T* numarul de elemente din tabloului *T*. Daca limitele sau tipul tabloului sunt importante, vom scrie:

procedure *proc*2(*T*[1 .. *n*])

sau, mai general:

procedure *proc*3(*T*[*a* .. *b*])

In aceste cazuri, n, a si b vor fi considerati parametri formali.

De multe ori, vom atribui unor elemente ale unui tablou *T* valorile $\pm\infty$, intelegand prin acestea doua valori numerice extreme, astfel incat pentru oricare alt element $T[i]$ avem $-\infty < T[i] < +\infty$.

Pentru simplitate, vom considera uneori ca anumite variabile sunt globale, astfel incat sa le putem folosi in mod direct in proceduri.

Iata acum si primul nostru algoritm, cel al inmultirii "a la russe":

```
function russe(A, B)
    arrays X, Y
    {initializare}
    X[1] ← A; Y[1] ← B
    i ← 1 {se construiesc cele doua coloane}
    while X[i] > 1 do
        X[i+1] ← X[i] div 2 {div reprezinta impartirea intreaga}
        Y[i+1] ← Y[i]+Y[i]
        i ← i+1
    {aduna numerele Y[i] corespunzatoare numerelor X[i] impare}
    prod ← 0
    while i > 0 do
        if X[i] este impar then prod ← prod+Y[i]
        i ← i−1
    return prod
```

Un programator cu experienta va observa desigur ca tablourile X si Y nu sunt de fapt necesare si ca programul poate fi simplificat cu usurinta. Acest algoritm poate fi programat deci in mai multe feluri, chiar folosind acelasi limbaj de programare.

Pe langa algoritmul de inmultire invatat in scoala, iata ca mai avem un algoritm care face acelasi lucru. Exista mai multi algoritmi care rezolva o problema, dar si mai multe programe care pot descrie un algoritm.

Acest algoritm poate fi folosit nu doar pentru a inmulti pe 45 cu 19, dar si pentru a inmulti orice numere intregi pozitive. Vom numi (45, 19) un *caz* (*instance*). Pentru fiecare algoritm exista un *domeniu de definitie* al cazurilor pentru care algoritmul functioneaza corect. Orice calculator limiteaza marimea cazurilor cu care poate opera. Aceasta limitare nu poate fi insa atribuita algoritmului respectiv. Inca o data, observam ca exista o diferenta esentiala intre programe si algoritmi.

1.2 Eficienta algoritmilor

Ideal este ca, pentru o problema data, sa gasim mai multi algoritmi, iar apoi sa-l alegem dintre acestia pe cel optim. Care este insa criteriul de comparatie? Eficienta unui algoritm poate fi exprimata in mai multe moduri. Putem analiza *a posteriori* (empiric) comportarea algoritmului dupa implementare, prin rularea pe calculator a unor cazuri diferite. Sau, putem analiza *a priori* (teoretic) algoritmul, inaintea programarii lui, prin determinarea cantitativa a resurselor (timp, memorie etc) necesare *ca o functie de marimea cazului considerat.*

Marimea unui caz x, notata cu $|x|$, corespunde formal numarului de biti necesari pentru reprezentarea lui x, folosind o codificare precis definita si rezonabil de compacta. Astfel, cand vom vorbi despre sortare, $|x|$ va fi numarul de elemente de sortat. La un algoritm numeric, $|x|$ poate fi chiar valoarea numerica a cazului x.

Avantajul analizei teoretice este faptul ca ea nu depinde de calculatorul folosit, de limbajul de programare ales, sau de indemanarea programatorului. Ea salveaza timpul pierdut cu programarea si rularea unui algoritm care se dovedeste in final ineficient. Din motive practice, un algoritm nu poate fi testat pe calculator pentru cazuri oricat de mari. Analiza teoretica ne permite insa studiul eficientei algoritmului pentru cazuri de orice marime.

Este posibil sa analizam un algoritm si printr-o metoda *hibrida*. In acest caz, forma functiei care descrie eficienta algoritmului este determinata teoretic, iar valorile numerice ale parametrilor sunt apoi determinate empiric. Aceasta metoda permite o predictie asupra comportarii algoritmului pentru cazuri foarte mari, care nu pot fi testate. O extrapolare doar pe baza testelor empirice este foarte imprecisa.

Este natural sa intrebam ce unitate trebuie folosita pentru a exprima eficienta teoretica a unui algoritm. Un raspuns la aceasta problema este dat de *principiul invariantei*, potrivit caruia doua implementari diferite ale aceluiasi algoritm nu difera in eficienta cu mai mult de o constanta multiplicativa. Adica, presupunand ca avem doua implementari care necesita $t_1(n)$ si, respectiv, $t_2(n)$ secunde pentru a rezolva un caz de marime n, atunci exista intotdeauna o constanta pozitiva c, astfel incat $t_1(n) \le ct_2(n)$ pentru orice n suficient de mare. Acest principiu este valabil indiferent de calculatorul (de constructie conventionala) folosit, indiferent de limbajul de programare ales si indiferent de indemanarea programatorului (presupunand ca acesta nu modifica algoritmul!). Deci, schimbarea calculatorului ne poate permite sa rezolvam o problema de 100 de ori mai repede, dar numai modificarea algoritmului ne poate aduce o imbunatatire care sa devina din ce in ce mai marcanta pe masura ce marimea cazului solutionat creste.

Revenind la problema unitatii de masura a eficientei teoretice a unui algoritm, ajungem la concluzia ca nici nu avem nevoie de o astfel de unitate: vom exprima eficienta in limitele unei constante multiplicative. Vom spune ca un algoritm necesita timp in *ordinul lui t*, pentru o functie t data, daca exista o constanta pozitiva c si o implementare a algoritmului capabila sa rezolve fiecare caz al problemei intr-un timp de cel mult $ct(n)$ secunde, unde n este marimea cazului considerat. Utilizarea secundelor in aceasta definitie este arbitrara, deoarece trebuie sa modificam doar constanta pentru a margini timpul la $at(n)$ ore, sau $bt(n)$ microsecunde. Datorita principiului invariantei, orice alta implementare a algoritmului va avea aceeasi proprietate, cu toate ca de la o implementare la alta se poate modifica constanta multiplicativa. In Capitolul 5 vom reveni mai riguros asupra acestui important concept, numit *notatie asimptotica*.

Daca un algoritm necesita timp in ordinul lui n, vom spune ca necesita timp *liniar*, iar algoritmul respectiv putem sa-l numim *algoritm liniar*. Similar, un algoritm este *patratic*, *cubic*, *polinomial*, sau *exponential* daca necesita timp in ordinul lui n^2, n^3, n^k, respectiv c^n, unde k si c sunt constante.

Un obiectiv major al acestei carti este analiza teoretica a eficientei algoritmilor. Ne vom concentra asupra criteriului timpului de executie. Alte resurse necesare (cum ar fi memoria) pot fi estimate teoretic intr-un mod similar. Se pot pune si probleme de compromis memorie - timp de executie.

1.3 Cazul mediu si cazul cel mai nefavorabil

Timpul de executie al unui algoritm poate varia considerabil chiar si pentru cazuri de marime identica. Pentru a ilustra aceasta, vom considera doi algoritmi elementari de sortare a unui tablou T de n elemente:

```
procedure insert(T[1 .. n])
    for i ← 2 to n do
        x ← T[i]; j ← i–1
        while j > 0 and x < T[ j]  do
            T[ j+1] ← T[ j]
            j ← j–1
        T[ j+1] ← x

procedure select (T[1 .. n])
    for i ← 1 to n–1 do
        minj ← i; minx ← T[i]
        for j ← i+1 to n do
            if T[ j] < minx then  minj ← j
                                  minx ← T[ j]
        T[minj] ← T[i]
        T[i] ← minx
```

Ideea generala a sortarii *prin insertie* este sa consideram pe rand fiecare element al sirului si sa il inseram in subsirul ordonat creat anterior din elementele precedente. Operatia de inserare implica deplasarea spre dreapta a unei secvente. Sortarea *prin selectie* lucreaza altfel, plasand la fiecare pas cate un element direct pe pozitia lui finala.

Fie U si V doua tablouri de n elemente, unde U este deja sortat crescator, iar V este sortat descrescator. Din punct de vedere al timpului de executie, V reprezinta cazul cel mai nefavorabil iar U cazul cel mai favorabil.

Vom vedea mai tarziu ca timpul de executie pentru sortarea prin selectie este patratic, independent de ordonarea initiala a elementelor. Testul "**if** $T[j] < minx$" este executat de tot atatea ori pentru oricare dintre cazuri. Relativ micile variatii ale timpului de executie se datoreaza doar numarului de executari ale atribuirilor din ramura **then** a testului.

La sortarea prin insertie, situatia este diferita. Pe de o parte, *insert*(U) este foarte rapid, deoarece conditia care controleaza bucla **while** este mereu falsa. Timpul necesar este liniar. Pe de alta parte, *insert*(V) necesita timp patratic, deoarece bucla **while** este executata de $i-1$ ori pentru fiecare valoare a lui i. (Vom analiza acest lucru in Capitolul 5).

Daca apar astfel de variatii mari, atunci cum putem vorbi de un timp de executie care sa depinda doar de marimea cazului considerat? De obicei consideram analiza pentru cel mai nefavorabil caz. Acest tip de analiza este bun atunci cand timpul de executie al unui algoritm este critic (de exemplu, la controlul unei centrale nucleare). Pe de alta parte insa, este bine uneori sa cunoastem timpul *mediu* de executie al unui algoritm, atunci cand el este folosit foarte des pentru cazuri diferite. Vom vedea ca timpul mediu pentru sortarea prin insertie este tot patratic. In anumite cazuri insa, acest algoritm poate fi mai rapid. Exista un

algoritm de sortare (*quicksort*) cu timp patratic pentru cel mai nefavorabil caz, dar cu timpul mediu in ordinul lui $n \log n$. (Prin log notam logaritmul intr-o baza oarecare, lg este logaritmul in baza 2, iar ln este logaritmul natural). Deci, pentru cazul mediu, *quicksort* este foarte rapid.

Analiza comportarii in medie a unui algoritm presupune cunoasterea a priori a distributiei probabiliste a cazurilor considerate. Din aceasta cauza, analiza pentru cazul mediu este, in general, mai greu de efecuat decat pentru cazul cel mai nefavorabil.

Atunci cand nu vom specifica pentru ce caz analizam un algoritm, inseamna ca eficienta algoritmului nu depinde de acest aspect (ci doar de marimea cazului).

1.4 Operatie elementara

O *operatie elementara* este o operatie al carei timp de executie poate fi marginit superior de o constanta depinzand doar de particularitatea implementarii (calculator, limbaj de programare etc). Deoarece ne intereseaza timpul de executie in limita unei constante multiplicative, vom considera doar numarul operatiilor elementare executate intr-un algoritm, nu si timpul exact de executie al operatiilor respective.

Urmatorul exemplu este testul lui Wilson de primalitate (teorema care sta la baza acestui test a fost formulata initial de Leibniz in 1682, reluata de Wilson in 1770 si demonstrata imediat dupa aceea de Lagrange):

```
function Wilson(n)
   {returneaza true daca si numai daca n este prim}
   if n divide ((n−1)! + 1)   then  return true
                              else  return false
```

Daca consideram calculul factorialului si testul de divizibilitate ca operatii elementare, atunci eficienta testului de primalitate este foarte mare. Daca consideram ca factorialul se calculeaza in functie de marimea lui n, atunci eficienta testului este mai slaba. La fel si cu testul de divizibilitate.

Deci, este foarte important ce anume definim ca operatie elementara. Este oare adunarea o operatie elementara? In teorie, nu, deoarece si ea depinde de lungimea operanzilor. Practic, pentru operanzi de lungime rezonabila (determinata de modul de reprezentare interna), putem sa consideram ca adunarea este o operatie elementara. Vom considera in continuare ca adunarile, scaderile, inmultirile, impartirile, operatiile modulo (restul impartirii intregi), operatiile booleene, comparatiile si atribuirile sunt operatii elementare.

1.5 De ce avem nevoie de algoritmi eficienti?

Performantele hardware-ului se dubleaza la aproximativ doi ani. Mai are sens atunci sa investim in obtinerea unor algoritmi eficienti? Nu este oare mai simplu sa asteptam urmatoarea generatie de calculatoare?

Sa presupunem ca pentru rezolvarea unei anumite probleme avem un algoritm exponential si un calculator pe care, pentru cazuri de marime n, timpul de rulare este de $10^{-4} \times 2^n$ secunde. Pentru $n = 10$, este nevoie de 1/10 secunde. Pentru $n = 20$, sunt necesare aproape 2 minute. Pentru $n = 30$, o zi nu este de ajuns, iar pentru $n = 38$, chiar si un an ar fi insuficient. Cumparam un calculator de 100 de ori mai rapid, cu timpul de rulare de $10^{-6} \times 2^n$ secunde. Dar si acum, pentru $n = 45$, este nevoie de mai mult de un an! In general, daca in cazul masinii vechi intr-un timp anumit se putea rezolva problema pentru cazul n, pe noul calculator, in acest timp, se poate rezolva cazul n+7.

Sa presupunem acum ca am gasit un algoritm cubic care rezolva, pe calculatorul vechi, cazul de marime n in $10^{-2} \times n^3$ secunde. In Figura 1.1, putem urmari cum

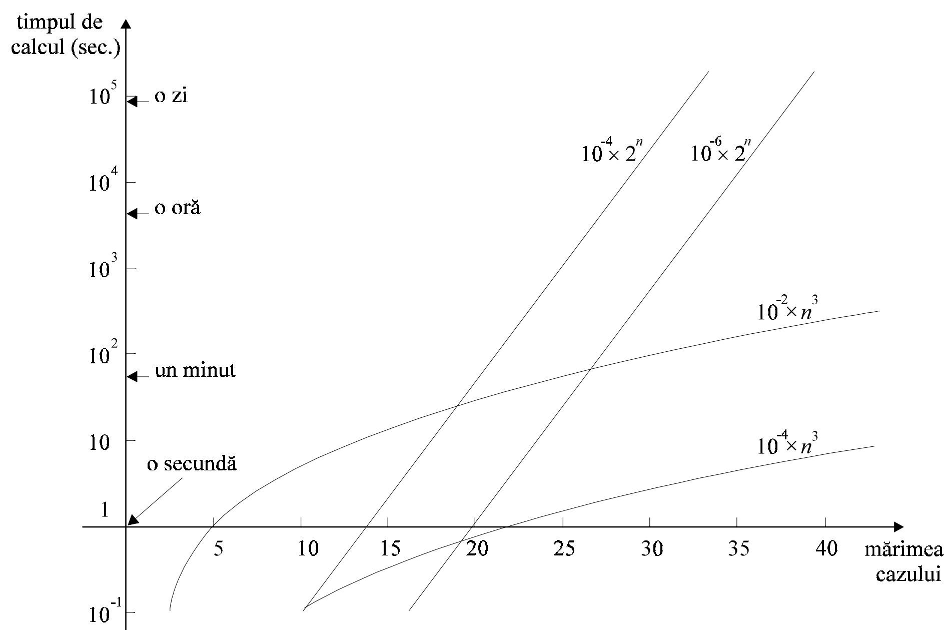

Figura 1.1 Algoritmi sau hardware?

evolueaza timpul de rulare in functie de marimea cazului. Pe durata unei zile, rezolvam acum cazuri mai mari decat 200, iar in aproximativ un an am putea rezolva chiar cazul $n = 1500$. Este mai profitabil sa investim in noul algoritm decat intr-un nou hardware. Desigur, daca ne permitem sa investim atat in software cat si in hardware, noul algoritm poate fi rulat si pe noua masina. Curba $10^{-4} \times n^3$ reprezinta aceasta din urma situatie.

Pentru cazuri de marime mica, uneori este totusi mai rentabil sa investim intr-o noua masina, nu si intr-un nou algoritm. Astfel, pentru $n = 10$, pe masina veche, algoritmul nou necesita 10 secunde, adica de o suta de ori mai mult decat algoritmul vechi. Pe vechiul calculator, algoritmul nou devine mai performant doar pentru cazuri mai mari sau egale cu 20.

1.6 Exemple

Poate ca va intrebati daca este intr-adevar posibil sa acceleram atat de spectaculos un algoritm. Raspunsul este afirmativ si vom da cateva exemple.

1.6.1 Sortare

Algoritmii de sortare prin insertie si prin selectie necesita timp patratic, atat in cazul mediu, cat si in cazul cel mai nefavorabil. Cu toate ca acesti algoritmi sunt excelenti pentru cazuri mici, pentru cazuri mari avem algoritmi mai eficienti. In capitolele urmatoare vom analiza si alti algoritmi de sortare: *heapsort*, *mergesort*, *quicksort*. Toti acestia necesita un timp mediu in ordinul lui $n \log n$, iar *heapsort* si *mergesort* necesita timp in ordinul lui $n \log n$ si in cazul cel mai nefavorabil.

Pentru a ne face o idee asupra diferentei dintre un timp patratic si un timp in ordinul lui $n \log n$, vom mentiona ca, pe un anumit calculator, *quicksort* a reusit sa sorteze in 30 de secunde 100.000 de elemente, in timp ce sortarea prin insertie ar fi durat, pentru acelasi caz, peste noua ore. Pentru un numar mic de elemente insa, eficienta celor doua sortari este asemanatoare.

1.6.2 Calculul determinantilor

Fie $\det(M)$ determinantul matricii

$$M = (a_{ij})_{i,\, j\, =\, 1,\, \ldots,\, n}$$

si fie M_{ij} submatricea de $(n-1) \times (n-1)$ elemente, obtinuta din M prin stergerea celei de-a i-a linii si celei de-a j-a coloane. Avem binecunoscuta definitie recursiva

$$\det(M) = \sum_{j=1}^{n} (-1)^{j+1} a_{1j} \det(M_{1j})$$

Daca folosim aceasta relatie pentru a evalua determinantul, obtinem un algoritm cu timp in ordinul lui $n!$, ceea ce este mai rau decat exponential. O alta metoda clasica, eliminarea Gauss-Jordan, necesita timp cubic. Pentru o anumita implementare s-a estimat ca, in cazul unei matrici de 20×20 elemente, in timp ce algoritmul Gauss-Jordan dureaza 1/20 secunde, algoritmul recursiv ar dura mai mult de 10 milioane de ani!

Nu trebuie trasa de aici concluzia ca algoritmii recursivi sunt in mod necesar neperformanti. Cu ajutorul algoritmului recursiv al lui Strassen, pe care il vom studia si noi in Sectiunea 7.8, se poate calcula $\det(M)$ intr-un timp in ordinul lui $n^{\lg 7}$, unde $\lg 7 \cong 2{,}81$, deci mai eficient decat prin eliminarea Gauss-Jordan.

1.6.3 Cel mai mare divizor comun

Un prim algoritm pentru aflarea celui mai mare divizor comun al intregilor pozitivi m si n, notat cu cmmdc(m, n), se bazeaza pe definitie:

```
function cmmdc-def (m, n)
    i ← min(m, n) + 1
    repeat i ← i−1 until i divide pe m si n
    return i
```

Timpul este in ordinul diferentei dintre min(m, n) si cmmdc(m, n).

Exista, din fericire, un algoritm mult mai eficient, care nu este altul decat celebrul algoritm al lui Euclid.

```
function Euclid(m, n)
    if n = 0  then  return m
              else  return Euclid(n, m mod n)
```

Prin m **mod** n notam restul impartirii intregi a lui m la n. Algoritmul functioneaza pentru orice intregi nenuli m si n, avand la baza cunoscuta proprietate

$$\text{cmmdc}(m, n) = \text{cmmdc}(n, m \textbf{ mod } n)$$

Timpul este in ordinul logaritmului lui min(m, n), chiar si in cazul cel mai nefavorabil, ceea ce reprezinta o imbunatatire substantiala fata de algoritmul precedent. Pentru a fi exacti, trebuie sa mentionam ca algoritmul originar al lui Euclid (descris in “*Elemente*”, aprox. 300 a.Ch.) opereaza prin scaderi succesive, si nu prin impartire. Interesant este faptul ca acest algoritm se pare ca provine dintr-un algoritm si mai vechi, datorat lui Eudoxus (aprox. 375 a.Ch.).

1.6.4 Numerele lui Fibonacci

Sirul lui Fibonacci este definit prin urmatoarea recurenta:

$$\begin{cases} f_0 = 0;\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \quad \text{pentru} \quad n \geq 2 \end{cases}$$

Acest celebru sir a fost descoperit in 1202 de catre Leonardo Pisano (Leonardo din Pisa), cunoscut sub numele de Leonardo Fibonacci. Cel de-al n-lea termen al sirului se poate obtine direct din definitie:

```
function fib1(n)
   if n < 2  then  return n
             else  return fib1(n−1) + fib1(n−2)
```

Aceasta metoda este foarte ineficienta, deoarece recalculeaza de mai multe ori aceleasi valori. Vom arata in Sectiunea 5.3.1 ca timpul este in ordinul lui ϕ^n, unde $\phi = (1+\sqrt{5})/2$ este *sectiunea de aur*, deci este un timp exponential.

Iata acum o alta metoda, mai performanta, care rezolva aceeasi problema intr-un timp liniar.

```
function fib2(n)
   i ← 1; j ← 0
   for k ← 1 to n do  j ← i + j
                      i ← j − i
   return j
```

Mai mult, exista si un algoritm cu timp in ordinul lui log n, algoritm pe care il vom argumenta insa abia in Capitolul 7:

```
function fib3(n)
   i ← 1; j ← 0; k ← 0; h ← 1
   while n > 0 do
      if n este impar then  t ← jh
                            j ← ih+jk+t
                            i ← ik+t
      t ← h^2
      h ← 2kh+t
      k ← k^2+t
      n ← n div 2
   return j
```

Va recomandam sa comparati acesti trei algoritmi, pe calculator, pentru diferite valori ale lui n.

1.7 Exercitii

1.1 Aplicati algoritmii *insert* si *select* pentru cazurile $T = [1, 2, 3, 4, 5, 6]$ si $U = [6, 5, 4, 3, 2, 1]$. Asigurati-va ca ati inteles cum functioneaza.

1.2 Inmultirea "a la russe" este cunoscuta inca din timpul Egiptului antic, fiind probabil un algoritm mai vechi decat cel al lui Euclid. Incercati sa intelegeti rationamentul care sta la baza acestui algoritm de inmultire.

Indicatie: Faceti legatura cu reprezentarea binara.

1.3 In algoritmul *Euclid*, este important ca $n \geq m$?

1.4 Elaborati un algoritm care sa returneze cel mai mare divizor comun a trei intregi nenuli.

Solutie:

function *Euclid-trei*(m, n, p)
  return *Euclid*(m, *Euclid*(n, p))

1.5 Programati algoritmul *fib*1 in doua limbaje diferite si rulati comparativ cele doua programe, pe mai multe cazuri. Verificati daca este valabil principiul invariantei.

1.6 Elaborati un algoritm care returneaza cel mai mare divizor comun a doi termeni de rang oarecare din sirul lui Fibonacci.

Indicatie: Un algoritm eficient se obtine folosind urmatoarea proprietate*, valabila pentru oricare doi termeni ai sirului lui Fibonacci:

$$\text{cmmdc}(f_m, f_n) = f_{\text{cmmdc}(m,\ n)}$$

1.7 Fie matricea $M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Calculati produsul vectorului (f_{n-1}, f_n) cu matricea M^m, unde f_{n-1} si f_n sunt doi termeni consecutivi oarecare ai sirului lui Fibonacci.

---

* Această surprinzătoare proprietate a fost descoperită în 1876 de Lucas.