Recursivitate

Capitolul

18

- Noțiunea de recursivitate
- Proiectarea unui algoritm recursiv
- Execuția apelurilor recursive
- Recursivitate indirectă (încrucișată)
- Greşeli frecvente în scrierea programelor recursive
- Când merită să utilizăm tehnica recursivității ?
- Implementări sugerate
- Probleme propuse
- Soluțiile problemelor

18.1. Noțiunea de recursivitate

Noțiunea de *recursivitate* din programare derivă în mod natural din noțiunea matematică cunoscută sub numele de *recurență*. Dar şi viața de zi cu zi oferă nenumărate exemple. Atunci când ne apucăm de făcut temele, deschidem o carte, citim din ea, dar la un moment dat ne trebuie o altă carte. Întrerupem studiul din prima carte şi citim din cealaltă. Este posibil să avem nevoie în continuare de alte cărți. Va sosi însă şi momentul când ne întoarcem la ultima carte întreruptă, continuăm (sau nu) să citim din ea, apoi ne întoarcem la precedenta şi în final la prima. Atunci când am terminat tema, o închidem şi pe aceasta.

Prezentăm în continuare câteva exemple clasice de *funcții definite recurent*. O funcție este recurentă dacă este definită prin sine însăşi.

Exemple

1. Definiția numerelor naturale conform axiomelor lui *Peano*:

- 1 este număr natural;
- Orice succesor al unui număr natural este un număr natural.

2. Secțiunea de aur

Secțiunea de aur, segmentul de aur sau proporția divină reprezintă toate acelaşi lucru, adică cea mai armonioasă împărțire, proporționare divină a figurilor geometrice. Încă din antichitate *Secțiunea de aur* era cunoscută ca fiind soluția ecuației:
$x^2 - x - 1 = 0$.

Rezolvând ecuația, obținem $x = \frac{1}{2}(\sqrt{5} + 1)$, ceea ce înseamnă aproximativ: **1.618033989.**

Această constantă este prezentă în tot ceea ce este viu, cum ar fi raportul dintre deget şi palmă, raportul dintre palmă şi antebraț şi aşa mai departe, motiv pentru care a fost adesea folosit de marii artişti, de la construcția piramidelor şi până la pictura renascentistă. Numeroase scrieri au fost consacrate secțiunii de aur, legilor sale şi participării sale la structurarea naturii şi a artei. Sub numele de „divine proportione" o găsim la *Luca Pacioli*, prieten şi colaborator al lui *Leonardo da Vinci*, sub formă de fracție continuă infinită:

$$x = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \ldots}}}}}$$

formă numită şi „combinarea unității cu infinitul".

3. Şirul lui Fibonacci

Se consideră că definirea şirului 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... a apărut în anul 1202, şi acesta se generează cu formulele:
$f_1 = 1, f_2 = 1, f_i = f_{i-1} + f_{i-2}$, pentru $i > 2$.

Acesta este datorat lui *Leonardo da Pisa*, fiul lui *Bonaccio* (ca urmare a formulării unei probleme privind înmulțirea iepurilor).

Primul termen este 0, următorii doi termeni ai şirului sunt 1, iar fiecare termen de indice mai mare decât 2 este egal cu suma ultimilor doi termeni care îl preced.

Observație

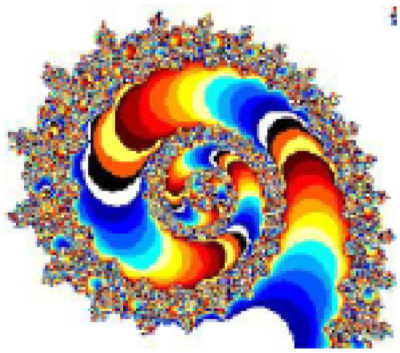
Pe măsură ce şirul continuă, raportul dintre doi termeni consecutivi se apropie de valoarea **1.618033989** (numărul de aur).

În continuare vom enumera câteva proprietăți interesante ale termenilor şirului lui *Fibonacci*:

1. $f_n^2 = f_{n-1} \cdot f_{n-1} + (-1)^n$
2. $f_{n-1} \cdot f_n = f_{n-2} \cdot f_{n-1} + (-1)^n$
3. $f_2 + f_4 + ... + f_{2n} = f_{2n-1} - 1$
4. $f_1 + f_3 + ... + f_{2n-1} = f_{2n}$
5. $f_1^2 + f_2^2 + ... + f_n^2 = f_n \cdot f_{n+1}$

4. Fractali
Fractalii sunt forme geometrice, definite recurent prin împărțirea unui segment sau a unei suprafețe în bucăți după o proporție dată. Fiecare bucată a unui fractal este o copie la scară redusă a întregului.

Din punct de vedere al modelului de generare a fractalilor, aceştia pot fi împărțiți în fractali naturali şi fractali artificiali. Fractalii naturali sunt cei existenți în natură sau care sunt creați în urma unor procese naturale. Norii, munții, copacii, malurile, pot fi considerați fractali naturali. De asemenea sistemul nervos şi sistemul de ramificație a bronhiilor sunt exemple de fractali naturali. În natură, cel mai adesea întâlnim proporția divină ca factor de proporționalitate al fractalilor.



18.2. Proiectarea unui algoritm recursiv

În programare, vorbim despre subprogram recursiv dacă acesta se autoapelează. Altfel spus, în corpul subprogramului apare un apel al subprogramului însuşi în timp ce acesta este activ. Pentru ca apelul să nu se realizeze la infinit este necesară existența în subprogram a unei condiții corecte de oprire a acestor apeluri.

Un alt punct delicat în realizarea unui subprogram recursiv este descrierea modelului. Există probleme în care putem aplica o formulă de recurență dată, dar şi probleme în care această relație trebuie determinată pe baza enunțului.

Prezentăm în continuare două exemple de proiectare a unor algoritmi recursivi:

a) Dacă formula recurentă este cunoscută, algoritmul va descrie această formulă.

Exemplu
Vom scrie un subprogram recursiv care calculează valoarea lui a^n, unde a este un număr real, iar n este număr natural diferit de 0.

Formula recurentă de calculare a funcției putere este:

$$a^n = \begin{cases} 1 & \text{dacă } n = 0 \\ a \cdot a^{n-1} & \text{dacă } n \neq 0 \end{cases}$$

Această formulă se poate rescrie:

$$Putere(a,n) = \begin{cases} 1 & \text{dacă } n = 0 \\ a \cdot Putere(a,n-1) & \text{dacă } n \neq 0 \end{cases}$$

O variantă de subprogram recursiv care corespunde acestei descrieri este:

```
Subalgoritm Putere(a,n):           { de tip funcţie, returnează un număr real }
  dacă n = 1 atunci
    Putere ← 1
  altfel
    Putere ← a * Putere(a,n-1)
  sfârşit dacă
sfârşit subalgoritm
```

b) Dacă formula recurentă nu este dată, aceasta trebuie să fie dedusă.

Exemplu

Vom scrie un subprogram recursiv care calculează cel mai mare divizor comun a două numere naturale *a* şi *b*, conform algoritmului lui *Euclid*.

Pentru a deduce formula recurentă de calcul a celui mai mare divizor comun prin algoritmul lui *Euclid*, vom urmări efectul algoritmului pe un exemplu.

Fie $a = 480$ şi $b = 220$.

a	:	*b*	=	*cât*	+	*rest*
480	:	220	=	2	+	40
220	:	40	=	5	+	20
40	:	20	=	2	+	0

Cmmdc(*a*, *b*) = 20 (ultima valoare a împărţitorului – când *a* se divide la *b*).

Se observă că algoritmul se opreşte atunci când *a* se divide la *b*. Această condiţie o vom folosi şi pentru a opri auto-apelările.

În termeni recursivi putem scrie:
cmmdc(480, 220) = *cmmdc*(220, 40) = *cmmdc*(40, 20) = 20.

Prin generalizare se obţine:

$$cmmdc(a,b) = \begin{cases} b & \text{dacă } rest[a/b] = 0 \\ cmmdc(b, rest[a/b] & \text{dacă } rest[a/b] \neq 0 \end{cases}$$

```
Subalgoritm Cmmdc(a,b):
  dacă rest[a/b] = 0 atunci
    Cmmdc ← b
  altfel
    Cmmdc ← Cmmdc(b,rest[a/b])
  sfârşit dacă
sfârşit subalgoritm
```

18.3. Execuția apelurilor recursive

Aşa cum s-a precizat în capitolul 7 (Subprograme), orice apel de subprogram (chiar şi atunci când este vorba de autoapel) are ca efect salvarea pe stiva calculatorului a adresei de revenire, a valorilor parametrilor transmişi prin valoare şi a adresei parametrilor transmişi prin referință, precum şi alocarea de spațiu pe stivă pentru variabilele locale ale subprogramului.

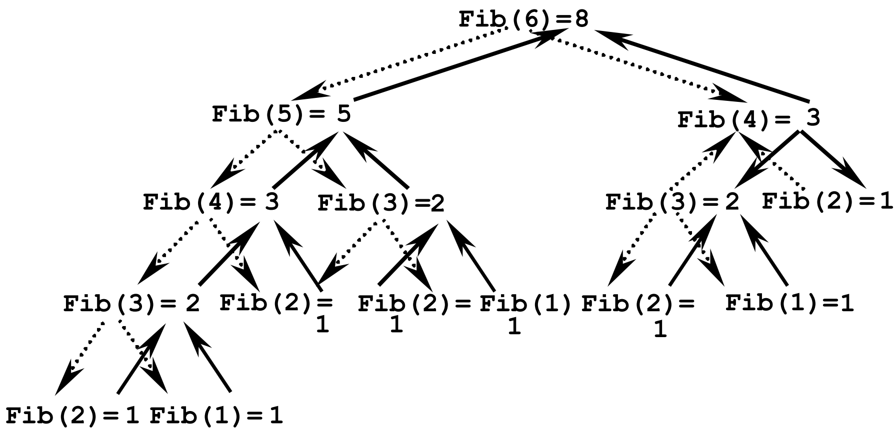
În continuare vom simula modul în care se execută subprogramul `Cmmdc(a,b)` prezentat mai sus pentru $a = 480$ şi $b = 220$, urmărindu-se pas cu pas apelurile succesive ale subprogramului, precum şi valorile existente pe stivă.

20	b		La revenire
40	a	În urma apelului `Cmmdc(40,20)`	`Cmmdc = 20`
40	b		
220	a	În urma apelului `Cmmdc(220,40)`	`Cmmdc = 20`
220	b		
480	a	În urma apelului `Cmmdc(480,220)`	`Cmmdc = 20`

18.3.1. Alte exemple

Vom scrie un algoritm care determină cel de-al *n*-lea termen al şirului lui *Fibonacci*, utilizând în acest scop un subprogram recursiv. Implementând acest algoritm, vom vedea că ***nu*** în cazul oricărei probleme este indicat să le rezolvăm recursiv. În cazul şirului lui *Fibonacci*, subprogramul recursiv este mare consumatoare de timp, deoarece în cazul fiecărui termen, calculele încep de la termenul de rang 1.

Ştiind că orice termen din şirul lui *Fibonacci* se calculează ca suma celor doi termeni care îl preced, se va urmări modul de calculare al celui de-al 6-lea termen. Fie $n = 6$. Considerăm că subprogramul recursiv poartă numele *Fib*. În urma apelului *Fib*(6), subprogramul se va autoapela pentru *Fib*(5) şi *Fib*(4). Să urmărim ce se întâmplă în cazul lui *Fib*(5). Se va apela *Fib*(4) şi *Fib*(3), apoi *Fib*(3) şi *Fib*(2), în final *Fib*(2) şi *Fib*(1). Valorile acestor termeni se cunosc, deci autoapelările pe acest „fir" se opresc. Astfel, se poate calcula valoarea lui *Fib*(3). Dar pentru a-l calcula pe *Fib*(4), mai întâi este nevoie de *Fib*(2). Apoi, după ce s-a calculat *Fib*(4), pentru a-l calcula pe *Fib*(5) din nou este nevoie de *Fib*(3). Înainte de a-l putea calcula pe *Fib*(6), din nou este nevoie, în afară de *Fib*(5) de *Fib*(4) care... Se observă că un acelaşi termen *Fib*(*i*) se calculează de foarte multe ori.



Subprogramul corespunzător se poate implementa pe baza subalgoritmului:

```
Subalgoritm Fib(n):
  dacă (n = 1) sau ( n = 2) atunci
    Fib ← 1
  altfel
    Fib ← Fib(n-1) + Fib(n-2)
  sfârşit dacă
sfârşit subalgoritm
```

Am văzut că pentru n = 6 avem 15 apeluri. Putem reduce numărul autoapelurilor, dacă valorile calculate le păstrăm într-un vector. Fie F acest vector, definit ca variabilă globală. Subalgoritmul s-ar rescrie astfel:

```
Subalgoritm Fib(n):
  dacă (n = 1) sau ( n = 2) atunci
    Fib ← 1
    F[n] ← 1
  altfel
    dacă F[n-1] ≠ 0 atunci
      Fib1 ← F[n-1]                          { determinăm Fib(n-1) }
    altfel
      Fib1 ← Fib(n-1)
      dacă F[n-2] ≠ 0 atunci Fib2 ← F[n-2]  { determinăm Fib(n-2) }
                     altfel Fib2 ← Fib(n-2)
      sfârşit dacă
      Fib ← Fib1 + Fib2
    sfârşit dacă
  sfârşit dacă
sfârşit subalgoritm
```

În această variantă pentru $n = 6$ se vor efectua 9 apeluri. Cu cât n are o valoare mai mare, cu atât prin a doua variantă se economiseşte un număr mai mare de calcule.

Despre subprogramul *Fib*(n) putem spune că are un grad mai înalt de recursivitate decât subprogramul *Cmmdc*(a, b). Există şi subprograme cu un grad şi mai înalt de recursivitate decât *Fib*(n), de exemplu subprogramul recursiv care ar implementa funcţia lui *Ackermann* definită astfel:

$$F(m,n) = \begin{cases} n+1 & \textit{dacă} \quad m = 0 \\ F(m-1,1) & \textit{dacă} \quad n = 0 \\ F(m-1, F(m,n-1)) & \textit{dacă} \quad m \neq 0 \textit{ şi } n \neq 0 \end{cases}$$

Se observă din formula recurentă a acestei funcţii că în situaţia în care $m \neq 0$ şi $n \neq 0$ se efectuează autoapel în autoapel.

Cu cât un subprogram are un grad mai înalt de recursivitate, cu atât creşte ineficienţa lui. Viteza de calcul scade şi se ajunge la depăşirea spaţiului stivei pentru date de intrare relativ mici.

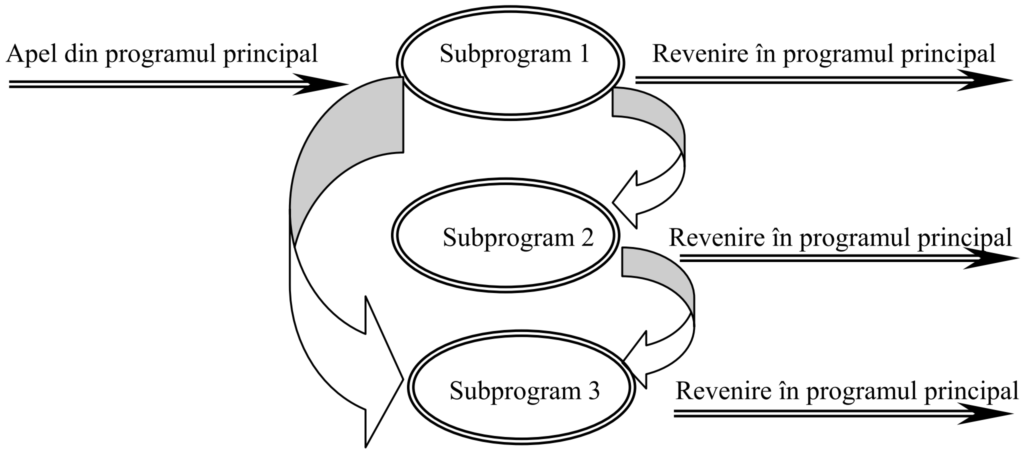
O soluţie de reducere a numărului de autoapeluri în aceste cazuri este construirea unei structuri de date (tablou uni- sau bidimensional, în funcţie de caz) în care să se păstreze valorile deja calculate, aşa cum s-a arătat în exemplul prezentat mai sus.

Încheiem această discuţie cu recomandarea de a evita alternativele recursive în rezolvarea problemelor în cazul cărora există variantă iterativă, care să nu fie mare consumatoare de spaţiu de memorie, chiar dacă programele recursive sunt mai compacte.

18.4. Recursivitate indirectă (încrucişată)

În situaţia în care două sau mai multe subprograme se apelează reciproc se spune că recursivitatea este *indirectă* sau încrucişată.

Exemplu



Observație

Este absolut necesar să se asigure ieşirea din recursivitate. Pentru aceasta va exista o condiție de ieşire cel puțin într-unul dintre subprograme.

Exemplu

Prezentăm în continuare un ***program demonstrativ*** care exemplifică utilizarea tehnicii recursivității încrucişate într-un program care simulează un cronometru care poate funcționa atât crescător, cât şi descrescător.

Folosim două proceduri, una care creşte valoarea cronometrului, iar cealaltă care scade valoarea de afişat. Tasta spațiu o vom folosi pentru a comuta de la un stil de cronometrare la altul.

Algoritmul se opreşte când cronometrul ajunge la +120 sau –120, această valoare fiind aleasă arbitrar.

Exemplul ales este unul dintre cele mai simple, rezolvarea bazându-se pe apelurile reciproce a două proceduri. Una care creşte valoarea contorului, cealaltă scade valoarea aceluiaşi contor.

După cum se poate observa din algoritmul următor, subalgoritmii se apelează unul pe celălalt, atunci când se apasă tasta spațiu.

Algoritmul va fi prezentat implementat în limbajul Pascal, folosind câteva subprograme predefinite din unit-ul *Crt*.

```
procedure Descreste(var nr:Integer); forward;
      { prin cuvântul forward se anunță compilatorul despre faptul că Descreste }
                                  { este o procedură care se va dezvolta ulterior }

procedure Creste(var nr:Integer);
begin
  ClrScr;                                                   { ştergem ecranul }
  Gotoxy(10,10);        { se poziționează cursorul pe ecran în vederea scrierii }
  Write(nr);                                       { se afişează contorul curent }
  if Abs(nr)<>120 then
                     { dacă nu schimbăm direcția timpul creşte, altfel descreşte }
    if ReadKey<>' ' then begin
      Inc(nr);
      Creste(nr)
    end else begin
      Dec(nr);
      Descreste(nr)
    end
end;
```

```
procedure Descreste(var nr:Integer);
begin
  ClrScr;
  Gotoxy(10,10);
  Write(nr);
  if Abs(nr)<>120 then
    if ReadKey<>' ' then begin
      Dec(nr);
      Descreste(nr)
    end else begin
      Inc(nr);
      Creste(nr)
    end
end;

Begin
  nr:=1;                { se porneşte de la valoarea 1 a contorului }
  Creste(nr)            { ... şi la început creşte contorul }
End.
```

18.5. Greşeli frecvente în scrierea programelor recursive

- O condiţie de ieşire incorectă din recursivitate va duce cel mai adesea la depăşirea capacităţii de memorare a stivei, caz în care programul se opreşte afişând mesajul: `Stack overflow`.
- Un autoapel incorect formulat duce la un rezultat incorect sau la umplerea stivei.
- În cazul în care se declară parametri formali transmişi prin valoare şi/sau variabile locale de tipuri care ocupă mult spaţiu de memorie, stiva calculatorului se va umple extrem de repede, ajungându-se la depăşirea spaţiului rezervat acestuia chiar şi pentru un număr mic de autoapeluri.

18.6. Când merită să utilizăm tehnica recursivităţii ?

Atunci când algoritmul care urmează să fie implementat descrie o noţiune recurentă sau algoritmul în sine este recursiv, se va lua în considerare oportunitatea descrierii acestuia utilizând tehnica recursivităţii. Se pune însă problema optimalităţii variantei recursive a algoritmului. Dacă acelaşi algoritm se poate realiza relativ simplu, utilizând tehnica iterativă (folosind structuri repetitive în locul apelului recursiv), atunci se preferă varianta iterativă datorită faptului că astfel se vor realiza programe mai rapide. Se evită astfel operaţiile mult prea dese de salvare pe stiva calculatorului, precum şi încărcarea acesteia în cazul apelurilor repetate. De asemenea, depanarea programelor recursive este mai anevoioasă decât a celor iterative.

Principalul avantaj al utilizării tehnicii recursive este că permite o descriere concisă a algoritmului, oglindind perfect definiția recurentă a respectivei noțiuni. Textul sursă al unui astfel de algoritm este, de regulă, mult mai scurt şi mai clar decât alte variante de algoritmi. Acest mod de scriere permite o divizare uşoară a problemei în subprobleme de acelaşi tip.

18.7. Implementări sugerate

Pentru a vă familiariza cu implementarea subprogramelor recursive, vă recomandăm să realizați următoarele exerciții:

1. inversarea cifrelor din configurația unui număr dat (fără *string*-uri);
2. inversarea numerelor dintr-un şir dat (fără tablouri);
3. calculul recursiv al factorialului;
4. descompunerea recursivă al unui număr dat în factori primi;
5. calculul recursiv al termenilor unui şir pe baza unei relații de recurențe;
6. generarea submulțimilor mulțimii $\{1, 2, \ldots, n\}$;
7. determinarea obiectelor din care se compune o fotografie.

18.8. Probleme propuse

18.8.1. Cifra maximă

Scrieți un program care citeşte un număr natural şi determină cel mai mic rang al cifrei maxime din număr, utilizând un subprogram recursiv.

Date de intrare

Valoarea numărului natural se citeşte din fişierul **`CIFRA.IN`**.

Date de ieşire

Cifra maximă şi rangul ei, separate de un spațiu, vor fi scrise în fişierul **`CIFRA.OUT`**.

Restricții şi precizări

- $1 \leq$ *numărul dat* ≤ 1000000000.

Exemplu

CIFRA.IN	**CIFRA.OUT**
28387625	8 4

18.8.2. Număr maxim

Scrieți un program care citește un număr natural *n* și un număr de cifre *k* (mai mic decât numărul de cifre ale numărului *n*) și determină numărul maxim care se poate obține din *n* prin eliminarea a *k* cifre. Cifrele numărului rezultat își vor păstra ordinea în număr.

Date de intrare

Cele două numere naturale se vor citi din fișierul **`MAXIM.IN`**.

Date de ieșire

Numărul obținut prin eliminarea celor *k* cifre se va scrie în fișierul **`MAXIM.OUT`**.

Restricții și precizări

- numărul natural *n* poate avea cel mult 255 de cifre;
- $1 \leq k \leq$ *numărul de cifre ale numărului* dat $n - 1$.

Exemplu

`MAXIM.IN`	**`MAXIM.OUT`**
`43869 3`	`89`

18.8.3. Sumă de numere *Fibonacci*

Scrieți un program care descompune un număr natural *n* ca sumă de număr minim de numere *Fibonacci*, utilizând pentru aceasta un subprogram recursiv.

Date de intrare

Numărul natural *n* se va citi din fișierul **`SUMA.IN`**.

Date de ieșire

În fișierul de ieșire **`SUMA.OUT`** se va scrie numărul dat urmat de semnul `'='` și de termenii sumei de numere *Fibonacci*.

Restricții și precizări

- $1 \leq n \leq 1000000$.

Exemplu

`SUMA.IN`	**`SUMA.OUT`**
`80`	`80=55+21+3+1`

18.8.4. La cules

Un fermier are o livadă dreptunghiulară şi doreşte să culeagă fructele cât mai uşor. Un vecin binevoitor se oferă să-l ajute cu un camion, dar are doar o singură zi la dispoziție, ceea ce înseamnă că poate face o singură parcurgere a livezii. Livada se află în pantă şi maşina poate să se deplaseze numai spre dreapta şi în jos. Camionul intră în livadă în colțul stânga-sus al caroiajului corespunzător livezii şi va ieşi în colțul din dreapta-jos.

Cunoscând cele două dimensiuni ale livezii şi numărul de mere din fiecare pom, să se găsească un drum optim posibil pentru camion, astfel încât până la ieşirea din livadă să fie culese cât mai multe mere.

Intrare

33	70	25	90
20	50	40	32
70	55	80	43
22	27	40	21
80	45	78	43

Ieşire

Date de intrare

Cele două numere naturale (*m* şi *n*), reprezentând dimensiunile livezii, se citesc de pe prima linie a fişierului **`LIVADA.IN`**. De pe următoarele *m* linii ale aceluiaşi fişier se vor citi câte *n* numere naturale, despărțite prin câte un spațiu, reprezentând numărul de mere din fiecare pom al rândului respectiv din livadă.

Date de ieşire

În fişierul **`LIVADA.OUT`** se va scrie un şir de caractere `'D'` şi `'J'`, unde `'D'` are semnificația dreapta, iar `'J'` înseamnă direcție de deplasare în jos, litere care corespund deplasărilor succesive pe care le va avea camionul pe drumul optim găsit.

Restricții şi precizări

- $1 \leq m, n \leq 100$.

Exemplu

`LIVADA.IN`
```
5 4
33 70 25 90
20 50 40 32
70 55 80 43
22 27 40 21
80 45 78 34
```

`LIVADA.OUT`
```
DJJDJJD
```

18.8.5. Numere romane

Scrieți un program care afişează un număr dat în sistemul de numerație zecimal (cu cifre arabe), în sistem de numerație roman, utilizând un subprogram recursiv.

Date de intrare

Numărul natural dat cu cifre arabe se va citi din fişierul de intrare **`ROMAN.IN`**.

Date de ieşire

Şirul de caractere reprezentând numărul roman se va scrie în fişierul **`ROMAN.OUT`**.

Restricții şi precizări

- $1 \leq$ *numărul dat* ≤ 3999

Exemplu

`ROMAN.IN`	**`ROMAN.OUT`**
`995`	`CMXCV`

18.9. Soluțiile problemelor propuse

18.9.1. Cifră maximă

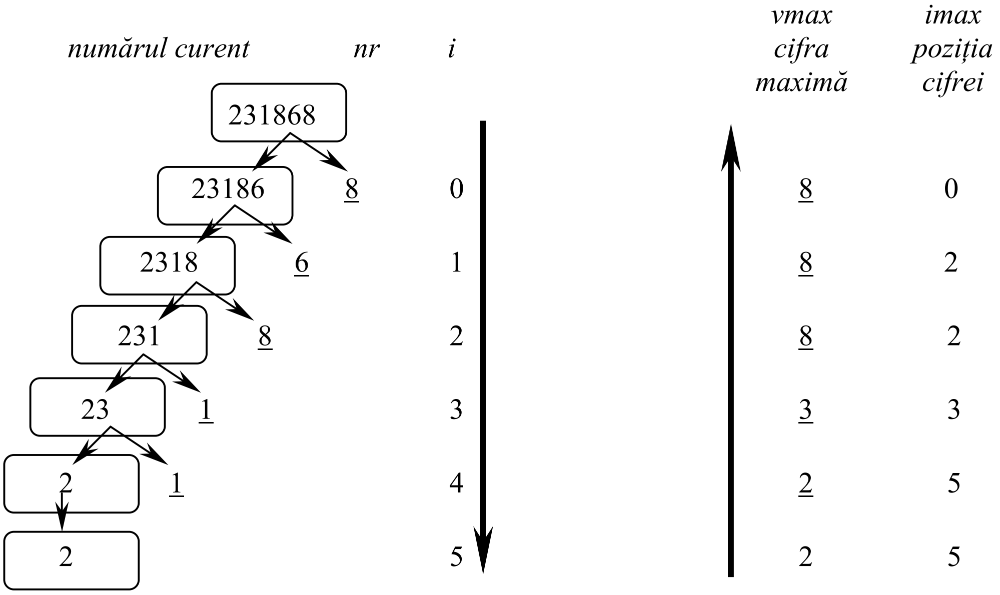
Prezentăm această problemă ca exercițiu de scriere a unui subprogram recursiv (rezolvarea iterativă (utilizând un ciclu **`cât timp`**) este mai simplă şi mai rapidă).

Am putea determina întâi cifra maximă din număr şi apoi prima poziție pe care aceasta apare. Dar aceste două obiective pot fi atinse printr-o singură parcurgere a cifrelor numărului.

Ideea de realizare recursivă ar suna astfel: dacă avem mai mult de o cifră în număr, cifra maximă va fi cea mai mare cifră dintre ultima cifră a numărului curent şi cifra maximă a numărului din care am eliminat ultima cifră. Dacă numărul curent are o cifră, aceasta este cifra maximă.

Prezentăm în continuare o schemă de aplicare a acestui algoritm recursiv pentru numărul 231868.

Apeluri recursive **Revenirea din apeluri**



Subalgoritmul trebuie să determine două valori, motiv pentru care acesta va fi de tip procedură şi aceste valori vor fi parametri transmişi prin referință. Pseudocudul de mai jos reprezintă algoritmul sugerat. Avem următoarele semnificații ale variabilelor:

- *nr* este numărul dat;
- *i* păstrează rangul cifrei curente;
- *vmax* conține valoarea cifrei maxime (până în prezent);
- *imax* păstrează rangul cifrei *vmax*.

```
Subalgoritm Cifra_maximă(nr,i,imax,vmax):
  dacă nr > 9 atunci                        { dacă numărul are mai multe cifre }
    { se apelează recursiv determinarea cifrei maxime din restul cifrelor numărului }
    Cifra_maximă([nr/10],i+1,imax,vmax)
    dacă rest[nr/10] > vmax atunci  { dacă ultima cifră a numărului este }
                                  { mai mare decât cifra maximă de până acum }
      vmax ← rest[nr/10]                                  { se păstrează cifra }
      imax ← i                                                    { şi rangul }
    sfârşit dacă
    altfel                              { când am ajuns la cifra de rang maxim }
      vmax ← nr                                   { se inițializează cifra maximă }
      imax ← i                                        { şi rangul cifrei maxime }
    sfârşit dacă
  sfârşit dacă
sfârşit subalgoritm
```

Observație

Algoritmul se poate încheia mai repede în cazul în care se găsește o cifră 9, deoarece nu mai poate exista alta mai mare decât ea.

18.9.2. Număr maxim

Să presupunem că numerele citite sunt 43881796 și 5. Numărul de cifre ale rezultatului va fi egal cu numărul cifrelor numărului dat minus numărul de cifre care trebuie eliminat, deci 8 – 5 = 3.

În algoritmul propus vom determina la fiecare pas câte o cifră din rezultat.

Pasul 1:

La primul pas se determină prima cifră (cea de rang maxim) a rezultatului. Această cifră se alege dintre cifrele numărului dat, exceptând ultimele două cifre (acestea nu au cum să fie pe prima poziție a unui număr de trei cifre, deoarece trebuie păstrată ordinea cifrelor în numărul dat). În exemplul considerat, cifra maximă a numărului 438817 este **8**, cifră pe care o punem în rezultat.

Pentru a pregăti pasul următor vom șterge din numărul inițial toate cifrele de la început și până inclusiv la prima apariție a cifrei determinate (cifrele 4, 3 și 8), deoarece acestea nu mai pot face parte din numărul rezultat. Numărul rămas este 81796.

Pasul 2:

La pasul doi se determină a doua cifră a rezultatului urmărind același algoritm. Vom căuta cifra maximă din numărul 8179 (lăsând la o parte 1 cifră de la sfârșit, deoarece, dacă am lua-o și pe aceasta în considerare s-ar putea întâmpla s-o aleagă și după eliminarea ei, respectiv a cifrelor care o preced nu am mai avea cifră pentru a completa rezultatul conform cerinței). Se găsește cifra **9** care se adaugă la rezultat și se renunță din nou la cifrele care nu pot apărea la pasul următor, deoarece sunt anterioare cifrei găsite. Se șterge secvența 8179 (până la prima apariție a cifrei maxime inclusiv) din număr.

Numărul rămas acum este 6.

Pasul 3:

La pasul trei cifra căutată este 6, iar numărul rămas după eliminarea lui 6 este 0.

Numărul maxim căutat este deci **896**.

În subalgoritmul recursiv corespunzător acestui algoritm avem următoarele semnificații ale variabilelor:

- *nr* este numărul dat (sub formă de șir de caractere);
- *câte* păstrează numărul cifrelor rămase de ales petru *rezultat*;
- *rezultat* este numărul care se caută;
- *p* ține evidența numărului cifrelor care se pot alege.

```
Subalgoritm Construieşte(nr,p,rezultat):
  dacă p > 0 atunci
    max ← 1                          { considerăm prima cifră ca fiind maximă }
    pentru i=2,(lungimea numărului nr) - p + 1 execută:
                        { se determină cifra maximă de pe pozițiile „permise" }
      dacă nr[i] > nr[max] atunci
        max ← i
      sfârşit dacă
    sfârşit pentru
    rezultat ← rezultat + nr[max]          { se păstrează cifra găsită }
    se şterg cifrele care nu mai pot face parte din rezultat
                                     { determinăm următoarea cifră a soluției }
    Construieşte(nr,p-1,rezultat)
  sfârşit dacă
sfârşit subalgoritm
```

18.9.3. Sumă de numere *Fibonacci*

Soluția prezentată în continuare determină cel mai apropiat număr *Fibonacci* de numărul dat, după care algoritmul repetă recursiv aceeaşi acțiune pentru diferența dintre numărul dat şi numărul *Fibonacci* găsit.

În varianta recursivă a algoritmului de rezolvare vom inițializa în prealabil elementele şirului lui *Fibonacci* până la cel mai apropiat număr de numărul dat. Astfel vom evita recalcularea repetată a unor termeni din şirul lui *Fibonacci*, ceea ce ar încetini execuția programului.

```
Subalgoritm Inițializare(nr,i):
          { generăm i termeni ai şirului lui Fibonacci, ultimul termen generat este }
            { cel mai apropiat număr Fibonacci mai mic sau egal cu numărul dat }
  fib[1] ← 1
  fib[2] ← 1
  i ← 2
  cât timp fib[i] < nr execută
    i ← i+1
    fib[i] ← fib[i-1] + fib[i-2]
  sfârşit cât timp
  n ← i
sfârşit subalgoritm
```

Un alt subprogram util în rezolvarea problemei este cel care determină indicele din şirul *Fibonacci* care corespunde unui număr *Fibonacci* dat.

```
Subalgoritm Caută(nr):
                { returnează poziţia pe care apare numărul nr în şirul Fibonacci }
  cât timp fib[n] > nr execută:
    n ← n-1
  sfârşit cât timp
  Caută ← n
sfârşit subalgoritm
```

Cu aceste precizări preliminarii, subprogramul recursiv propus pentru afişarea descompunerii unui număr în sumă de numere *Fibonacci* este:

```
Subalgoritm Recursiv(nr):
  k ← Caută(nr)          { se determină indicele acelui număr Fibonacci }
  scrie fib[k]              { care este cel mai apropiat de numărul nr }
  dacă nr ≠ fib[k] atunci
    scrie '+'
    Recursiv(nr-fib[k])
  sfârşit dacă
sfârşit subalgoritm
```

În programul principal sunt descrise următoarele acţiuni:

- se citeşte numărul dat;
- se apelează iniţializarea care determină şi indicele celui mai apropiat număr *Fibonacci* mai mic sau egal cu numărul dat;
- se afişează numărul dat urmat de semnul `'='`;
- se apelează subprogramul recursiv, pornind de la cel mai mare număr *Fibonacci* din descompunerea numărului dat.

18.9.4. La cules

Rezolvarea acestei probleme înseamnă găsirea răspunsului la două cerinţe:

a. determinarea numărului maxim de mere care pot fi culese printr-o parcurgere conform enunţului;

b. determinarea drumului care a condus la soluţia optimă.

a. Ţinând cont de faptul că deplasarea se poate face doar pe două direcţii, numărul maxim de mere care pot fi culese se poate determina până în orice punct al livezii. Aceste informaţii se vor construi pas cu pas şi se vor stoca într-un tablou bidimensional *b*.

Primul element al tabloului *b* va fi *b*[1,1] care va fi iniţializat cu numărul de mere ale primului pom (*a*[1,1]) aflat pe direcţia de pornire. În continuare vom putea completa prima linie (dacă păstrăm direcţia de pornire până la capătul livezii) şi prima coloană a tabloului bidimensional dacă se urmează direcţia în jos. Să urmărim aceşti paşi pe exemplul dat în enunţ:

```
33 70 25 90
20 50 40 32
70 55 80 43
22 27 40 21
80 45 78 34
```

Construirea liniei 1:

33	33 + 70 = 103	103 + 25 = 128	128 + 50 = 218
…			

Secvența în pseudocod corespunzătoare acestor acțiuni este:

```
...                                                    { crearea primei linii }
  b[1,1] ← a[1,1]
  pentru j=2,n execută:  { se adună la numărul maxim de mere de pe poziția }
                          { anterioară numărul de mere din pomul curent }
    b[1,j] ← b[1,j-1] + a[1,j]
  sfârşit pentru
...
```

Construirea coloanei 1:

33	103	128	218
33 + 20 = 53			
53 + 70 = 123			
123 + 22 = 145			
145 + 80 = 225			

Algoritmul este similar celui de mai sus:

```
...                                                  { crearea primei coloane }
 pentru i=2,m execută:                         { se adună la numărul maxim }
   b[i,1] ← b[i-1,1]+a[i,1]   { de mere de pe poziția de deasupra poziției }
 sfârşit pentru               { curente numărul de mere din pomul curent }
...
```

Din acest moment se poate începe completarea restului tabloului pe linii (începând cu linia 2) sau pe coloane (începând cu coloana 2), calcularea valorii fiecărui element (neapărat în ordine) se va face prin alegerea numărului maxim de pomi între valoarea de deasupra şi cea din stânga (care sunt deja calculate) plus numărul de mere din pomul la care ne aflăm.

De exemplu:

$b[2,2] = \text{maxim}\{103, 53\} + a[2,2] = 103 + 50 = 153$

$b[2,3] = \text{maxim}\{153, 128\} + a[2,3] = 153 + 40 = 193$

şi aşa mai departe, până când s-au calculat toate elementele tabloului bidimensional *b*.

Valoarea din colţul dreapta jos (*b*[*m*, *n*]) va fi egală cu numărul maxim de mere care pot fi culese la o parcurgere a livezii.

33	103	128	218
53	103 + 50 = 153	153+40=193	218+32=250
123	153 + 55 = 208	208 + 80 = 288	288 + 43 = 321
145	208 + 27 = 235	288 + 40 = 328	328 + 21 = 349
225	235 + 45 = 280	328 + 78 = 406	406 + 34 = 440

Algoritmul cu care calculăm această valoare este:

```
...
                                        { completarea tabloului pe celelalte linii şi coloane }
  pentru i=2,m execută:
    pentru j=2,n execută:
      dacă b[i-1,j] > b[i,j-1] atunci
        b[i,j] ← b[i-1,j] + a[i,j]
      altfel
        b[i,j] ← b[i,j-1] + a[i,j]
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
...
```

b. În faza a doua trebuie să ne întoarcem, pornind de la elementul *b*[*m*, *n*] pe drumul pe care s-a obţinut această valoare maximă.

Acest proces de „regăsire a drumului" este cel mai adesea descris prin tehnică recursivă.

Modalitatea de regăsire a drumului pe care a fost determinată valoarea maximă se poate intui din tabelul anterior. Se porneşte de la elementul *b*[*m*, *n*] şi se verifică dacă la acea valoare am ajuns de la elementul de deasupra sau de la elementul din stânga sa. Calculăm 440 – 406 = 34 şi 440 – 349 = 91. Deoarece *a*[*m*, *n*] = 34, se trage concluzia că am ajuns în *b*[*m*, *n*] din *b*[*m*, *n* – 1]. În vectorul rezultat, introducem caracterul `'D'` (deoarece am avansat în această poziţie mergând spre dreapta).

În mod similar se reface tot drumul. Restul caracterelor le alipim mereu în faţa şirului rezultat corespunzător unui pas care pe drum este mai aproape de punctul de start.

În cazul subprogramelor recursive este esenţial să se determine corect condiţia de oprire şi formula recurentă după care se face autoapelul. În cazul de faţă ne vom opri atunci când am ajuns la elementul de indice (1,1). Observăm că întotdeauna se va ajunge în această poziţie a tabloului, deoarece acesta a fost punctul de plecare în prima parte a algoritmului. Acum se reface drumul optim pe care s-a ajuns din *b*[1,1] în *b*[*m*, *n*].

Observație

În algoritmul descris am aplicat metoda programării dinamice[*)]. Se observă cele două faze ale algoritmului, specifice acestei metode au fost:
1) Construirea soluției optime;
2) Refacerea drumului pe care a fost obținută această soluție.

Subalgoritmul recursiv care regăsește drumul pe care a fost construită soluția optimă este:

```
Subalgoritm Drumul_optim(i,j,rezultat):
  dacă (i ≠ 1) or (j ≠ 1) atunci
    dacă b[i,j] = b[i,j-1] + a[i,j] atunci
      rezultat ← 'D' + rezultat
      Drumul_optim(i,j-1,rezultat)
    sfârşit dacă
  altfel
    dacă b[i,j] = b[i-1,j] + a[i,j] atunci
      rezultat ← 'J' + rezultat
      Drumul_optim(i-1,j,rezultat)
    sfârşit dacă
  sfârşit dacă
sfârşit subalgoritm
```

18.9.5. Numere romane

După cum se știe, există două metode de scriere a unui număr în sistemul de numerație roman: una aditivă și una prin diferență, ambele metode având totuși limitări de aplicare.

În varianta aditivă, de exemplu, nu putem scrie mai mult de trei litere consecutive identice. Varianta prin diferență a apărut pentru a prescurta numerele care altfel erau mult prea lungi. Pentru ca scrierea să fie totuși unică s-au emis următoarele reguli:

1. Numai cifrele I, X, și C pot fi scăzute.
2. Doar valoarea unei singure cifre poate fi scăzută (nu se permite scăderea unui grup de litere).
3. Numărul care se scade trebuie să aibă o valoare mai mare sau egală cu o zecime din descăzut.

Exemplu

n = 449, CDXLIX CD = 500 – 100 = 400 apoi XL= 50 – 10 = 40 și IX = 10 – 1 = 9. Deci, în total: 400 + 40 + 9 = 449

Deosebim următoarele situații:

[*)] Metoda programării dinamice se va studia în clasa a X-a.

1. Dacă cifra de reprezentat este 1, 10, 100 sau 1000, se foloseşte varianta aditivă şi alipim simbolurile I, X, şi C sau M.
2. Dacă valoarea cifrei de reprezentat începe cu 5, 6, 7, sau 8, se foloseşte tot varianta aditivă, prin alipirea simbolurilor V, L sau D şi după caz, cu alipirea suplimentară a simbolurilor I, X, sau C în numărul corespunzător.
3. Dacă valoarea cifrei de reprezentat începe cu 4 se foloseşte varianta prin diferență față de simbolul V, L sau D.
4. Dacă valoarea cifrei de reprezentat începe cu 9, se foloseşte varianta prin diferență față de simbolul X, C sau M.

În variantele prin diferență formula de calcul diferă în funcție de modul în care se îndeplinesc condițiile 1 şi 3. De exemplu, numărul 9 se calculează ca diferență între 10 şi 1, deoarece 1 este egal cu o zecime din 10. Reprezentarea romană a numărului este IX. Numărul 99 nu se poate calcula ca diferență între 100 şi 1 (cum am fi îndreptățiți să sperăm) pentru că 1 este mai puțin de o zecime din 100. În acest caz aplicăm cealaltă variantă în care exprimăm 90 ca diferență între 100 şi 10 (ceea ce este permis) urmând să reluăm algoritmul pentru diferența rămasă: 100 – 90 = 9. În final, numărul în reprezentarea romană este XCIX.

Am ales să descriem acest algoritm în varianta recursivă, deoarece modalitatea practică de rezolvare a acestei probleme este definită recurent: se determină o literă romană, după care algoritmul se aplică identic şi pentru numărul rămas.

În varianta următoare folosim două constante de tip vector, una conținând valorile corespunzătoare cifrelor romane, iar cealaltă conținând chiar literele folosite în scrierea romană. Este de înțeles că pentru orice indice, valoarea din primul vector va corespunde literei din cel de-al doilea vector astfel:

```
const Lit=(' ','M','D','C','L','X','V','I');     { vector de caractere }
      v=(10000,1000,500,100,50,10,5,1);          { vector de numere }
```

Elementul de indice 0 are o valoare nesemnificativă, pentru simplitatea rezolvării.

Determinarea ordinului de mărime a cifrei romane corespunzând unui număr dat se realizează în următorul subalgoritm:

```
Subalgoritm Aduce(rangul,număr):
          {7 este ordinul de mărime maxim corespunde literei I având valoarea 1 }
  rangul ← 7
  cât timp v[rangul-1] ≤ număr execută:
    rangul ← rangul - 1
  sfârşit cât timp
```

```
  dacă prima_cifră_arabă(număr) = 9 atunci
    rangul ← rangul + 1
  sfârşit dacă
sfârşit subalgoritm
```

Pentru numerele care încep cu cifra 9 se efectuează o corecţie, astfel rangul returnat va corespunde simbolurilor de scăzut I, X sau C după caz (în loc de V, L sau D).

Următorul subalgoritm recursiv determină numărul roman; cu *p* am notat un şir de caractere în care am alipit literele identice atunci când acestea trebuie să se repete.

```
Subalgoritm NR(număr):
  dacă număr = 0 atunci
    NR ← ''
  altfel
    Aduce(rangul,număr)
    Prima_cifră ← [număr/v[rangul]]
                        { variantă aditivă care foloseşte literele I, X, C sau M }
    dacă prima_cifră este 0, 1, 2 sau 3 atunci
      p ← ''
      pentru i=1,prima_cifră execută:
        p ← p + Lit[rangul]
      sfârşit pentru
      NR ← p + NR(număr - prima_cifră*v[rangul])
    sfârşit dacă
        { variantă aditivă în care cifra curentă se scrie folosind litera V, L, sau D }
    dacă prima_cifra este 5, 6, 7 sau 8 atunci
      p ← ''
      pentru i=6,prima_cifră execută:
        p ← p + Lit[rangul+1]
      sfârşit pentru
      NR ← Lit[rangul] + p + NR(număr - prima_cifră*v[rangul])
    sfârşit dacă
                                                      { varianta prin diferenţă }
    dacă prima_cifră este 4 atunci
      dacă (v[rangul-1] - număr ∈ {1,10,100}) şi
           (v[rangul-1] – număr ≥ [v[rangul-1]/10]) atunci
        nr ← NR(v[rangul-1]- număr) + Lit[rangul-1]
      altfel
        nr ← Lit[rangul] + Lit[rangul-1]+
              NR(număr - prima_cifra_romana*v[rangul])
      sfârşit dacă
    sfârşit dacă
```

```
    dacă prima_cifră este 9 atunci
      dacă (v[rangul-2] - număr ∈ {1,10,100}) şi
                  (v[rangul-2]- număr ≥ [v[rangul-2]/10]) atunci
        nr ← NR(v[rangul-2]- număr) + Lit[rangul-2]
      altfel
        nr ← Lit[rangul] + Lit[rangul-2] +
                               NR(număr - prima_cifra*v[rangul])
      sfârşit dacă
    sfârşit dacă
  sfârşit dacă
sfârşit subalgoritm
```

În încheiere, „nu putem rezista tentației" şi vă prezentăm şi o alternativă iterativă pentru rezolvarea acestei probleme. Cu *arab* am notat numărul arab, cu *roman*, numărul roman. În subalgoritmul `Parteroman(roman,arab,partearab,r1,r2,r3)` se caută părți din numărul arab aparținând anumitor intervale. Acestea se scad din numărul arab şi în numărul roman se alipesc literele corespunzătoare. `partearab` *poate fi* 1000, 100 sau 10, iar caracterele `r1`, `r2`, `r3` au valori corespunzătoare ordinului de mărime a lui `partearab`: 1000: `'M'`, `'D'`, `'C'`, 100: `'C'`, `'L'`, `'X'`, 10: `'X'`, `'V'`, `'I'`.

```
Subalgoritm Parteroman(roman,arab,partearab,r1,r2,r3):
  cât timp arab ≥ partearab execută:
    roman ← roman + r1
    arab ← arab - partearab
  sfârşit cât timp
  dacă arab ≥ [9*partearab/10] atunci
    roman ← roman + r3 + r1
    arab ← arab – [9*partearab/10]
  altfel
    dacă arab ≥ [partearab/2] atunci
      roman ← roman + r2
      arab ← arab – [partearab/2]
    altfel
      dacă arab > [4*partearab/10] atunci
        roman ← roman + r3 + r2
        arab ← arab – [4*partearab/10]
      sfârşit dacă
    sfârşit dacă
  sfârşit dacă
sfârşit subalgoritm
```

Acest subalgoritm se apelează de trei ori din programul principal, apoi, în cazul în care mai există valori diferite de 0 în numărul arab, acestea se scriu cu cifre I.

```
...
  roman ← ''
  Parteroman(roman,arab,1000,'M','D','C')
  Parteroman(roman,arab,100,'C','L','X')
  Parteroman(roman,arab,10,'X','V','I')
  cât timp arab >= 1 execută: { ceea ce a rămas din arab, se scrie cu cifre I }
    roman ← roman + 'I'
    arab ← arab - 1
  sfârşit cât timp
...
```