Programarea calculatoarelor

Recursivitate. Citirea caracterelor

Marius Minea

11 martie 2008

Recursivitate: putere cu înjumătățirea exponentului

– recursiv, rezolvăm o problemă reducând-o la o problemă mai simplă
– adesea, e eficientă împărțirea în două probleme cât mai egale
= strategie *divide et impera* (divide and conquer)

$$x^n = \begin{cases} 1 & n = 0 \\ (x^{n/2})^2 & n \text{ par} \\ x \cdot (x^{n/2})^2 & n \text{ impar} \end{cases}$$

```
double sqr(double x) { return x*x; }
double pow2(double x, unsigned n) {
  return n == 0 ? 1
   : n % 2 == 0 ? sqr(pow2(x, n/2))
   : x * sqr(pow2(x, n/2));
}
```

– numărul de apeluri necesar e $1 + \lfloor \log_2 n \rfloor$
(exponentul se înjumătățește la fiecare apel recursiv)
de ex.: `pow2(5, 6)` → `pow2(25, 3)` → `pow2(625, 1)` → `pow2(625, 0)`
– evaluarea lui `pow(x, n/2)` se face *o singură dată* ca argument pt. `sqr` care lucrează cu *valoarea* obținută (nu substituie expresia de două ori)

Apeluri și calcule repetate

– dacă înlocuim direct $x^{n/2} \cdot x^{n/2}$ în locul funcției `sqr`
și tipărim exponentul pentru a urmări desfășurarea apelurilor recursive:

obținem pentru exponent $n = 3$:

```
double pow2(double x, unsigned n) {
  printf("exponent %u\n", n);
  return n == 0 ? 1
   : n % 2 == 0 ? pow2(x, n/2) * pow2(x, n/2)
   : x * pow2(x, n/2) * pow2(x, n/2);
}
```

```
exponent 3
exponent 1
exponent 0
exponent 0
exponent 1
exponent 0
exponent 0
```

– cele două expresii `pow(x, n/2)` se evaluează succesiv, independent!
fără optimizări, compilatorul nu caută expresii egale, și recalculează
– nr. de apeluri e mai mare decât la înmulțirea obișnuită $x \cdot \ldots \cdot x$
⇒ Atenție la repetarea ineficientă a rezolvării acelorași subprobleme

Puterea cu înjumătățirea exponentului (cont.)

– dar, putem rescrie definiția chiar mai simplu:

$$x^n = \begin{cases} 1 & n = 0 \\ (x^2)n/2 & n \text{ par} \\ x \cdot (x^2)^{n/2} & n \text{ impar} \end{cases}$$

```
double pow2(double x, unsigned n) {
  return n == 0 ? 1
   : n % 2 == 0 ? pow2(x*x, n/2)
   : x * pow2(x*x, n/2);
}
```

(similar cu prima variantă, dar nu mai e nevoie de `sqr`)
⇒ uneori o schimbare minoră în formularea problemei conduce la o soluție foarte diferită

Exemplu: șirul lui Fibonacci

$F_0 = F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ $(n \geq 2)$

```
unsigned fib(unsigned n) {
  printf("calculez fib(%d)\n",n);
  return n<2 ? 1 : fib(n-1)+fib(n-2);
}
// pentru apelul fib(4) obținem:
```

```
calculez fib(4)
calculez fib(3)
calculez fib(2)
calculez fib(1)
calculez fib(0)
calculez fib(1)
calculez fib(2)
calculez fib(1)
calculez fib(0)
```

Modificăm: transmitem indicele `k` pana la care s-a facut calculul,
și ultimii doi termeni calculați `fk` și `fk_1`

```
unsigned fibc(unsigned n, unsigned k, unsigned fk, unsigned fk_1) {
  return k == n ? fk : fibc(n, k+1, fk+fk_1, fk);
}
// definim o functie cu un parametru, apelata de utilizator
unsigned fib1(unsigned n) {
  return n < 2 ? 1 : fibc(n, 1, 1, 1);
}
```


Recursivitate: probleme cu structură liniară (de șir)

Un șir e un element sau un șir urmat de un element
⇒ multe probleme au această structură secvențială
Exemplu: un număr natural în baza 10 are fie o singură cifră, fie e format din ultima cifră, precedată de alt număr în baza 10.
descompunere cu relația: $n = 10 \cdot (n/10) + n\%10$, ex. $1457 = 10 \cdot 145 + 7$

```
unsigned nrcifre(unsigned n) {
  return n < 10 ? 1 : 1 + nrcifre(n / 10);
}
unsigned nrcif2(unsigned n, unsigned r) {
  return n < 10 ? r : nrcif2(n / 10, r + 1);
}    // r : numarul de cifre acumulate/numarate deja
```

Nu e natural sa-i dăm utilizatorului o funcție cu 2 parametri pentru o problema care o singura dată de intrare. ⇒ Definim o nouă funcție:

```
unsigned nrcif(unsigned n) { return nrcif2(n, 1); }
```

Probleme cu numere ca siruri de cifre (cont.)

```
unsigned max(unsigned a, unsigned b) { return a > b ? a : b; }
unsigned maxcifra(unsigned n) {
  return n < 10 ? n : max(n%10, maxcifra(n/10));
}
unsigned maxcif2(unsigned n, unsigned mc) {
  return n == 0 ? mc : maxcif2(n/10, max(mc, n%10));
}
unsigned maxcif(unsigned n) { return maxcif2(n/10,n%10); }
```

număr cu cifrele în ordine inversă: ultima cifră din n se adaugă la coada parții deja inversate r: $1472 \to 147, 2 \to 14, 27 \to 1, 274 \to 2741$

```
unsigned revnum_r(unsigned n, unsigned r) {
  return n == 0 ? r : revnum_r(n / 10, 10 * r + n % 10);
}
unsigned revnum(unsigned n) { return revnum_r(n, 0); }
```


Caractere. Codul ASCII

ASCII = American Standard Code for Information Interchange
Caracterele sunt memorate ca și cod numeric = indicele în acest tabel
ex. '0' == 48, 'A' = 65, 'a' = 97, etc.

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
-------------------------------------------------------
0x0    \0                    \a \b \t \n \v \f \r
0x10:
0x20:        !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
0x30:   0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
0x40:   @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
0x50:   P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
0x60:   `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
0x70:   p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

Am scris cu prefixul `0x` constante hexazecimale (în baza 16)
– caracterele $<$ `0x20` (spațiu): caractere de control
– cifrele; literele mari; literele mici: în secvențe contigue
– caracterele cu cod $>$ `0x7f` (127): nu fac parte din setul ASCII (diacritice, etc. – diverse variante standardizate de ISO)

Tipul caracter în C

Tipul standard `char` reprezintă caractere (codul lor ASCII – un întreg) ⇒ în C, tipul `char` e un *tip întreg*, dar cu domeniu de valori mai restrâns decât `int` sau `unsigned` ⇒ poate fi memorat pe *un octet* (8 *biți*)

Cf. standardului, `char` poate fi `signed char`, cu valori de la `-128` la `127`, sau `unsigned char`, cu valori de la `0` la `255`. Ambele sunt incluse în `int`.

În program, *constantele caracter* se scriu între apostroafe (simple) ' '
Au valori întregi: codul ASCII. În calcul se convertesc automat la `int`.
Cifrele, literele mici și literele mari sunt dispuse consecutiv ⇒ avem:
`'7' == '0' + 7` `'5' - '0' == 5` `'E' - 'A' == 4` `'f' == 'a' + 5`

Reprezentări pentru caractere speciale:

`'\0'`	null	`'\n'`	linie nouă
`'\a'`	alarm	`'\r'`	carriage return
`'\b'`	backspace	`'\f'`	form feed
`'\t'`	tab	`'\''`	apostrof
`'\v'`	vertical tab	`'\\'`	backslash

Citirea și scrierea unui caracter

`int getchar(void)` declarată în `stdio.h`
– funcție fără parametri, returnează valoarea caracterului (codul ASCII) ca și `unsigned char` convertit la `int`
– sau returnează valoarea specială `EOF` (end-of-file) (-1, diferită de orice `unsigned char` dacă nu s-a putut citi un caracter (la sfârșit de fișier)
La tastatură, caracterele sunt introduse *cu ecou*, și devin disponibile pentru citire din program doar după ce se tastează *Enter*.

ATENȚIE! Programul nu are control asupra datelor introduse la citire ⇒ trebuie verificat întotdeauna ce s-a citit / testate erorile

`int putchar(int c)`
– scrie un `unsigned char` dat ca și `int`; returnează valoarea scrisă

```
#include <stdio.h>
int main(void) {
  putchar(':');         // scrie caracterul :
  putchar(getchar());   // citeste si scrie un caracter
  return 0;
}
```


Exemplu: Citirea unui număr întreg

```
#include <ctype.h>      // pt. functia isdigit (caract. e cifra ?)
#include <stdio.h>
// citeşte nr. natural, până la primul caracter diferit de cifră
unsigned readnat_rc(unsigned r, int c) {
// r: rezultatul parţial acumulat; c: următorul caracter citit
  return isdigit(c) ? readnat_rc(10*r + (c-'0'), getchar()) : r;
}
int readnat(void) { return readnat_rc(0, getchar()); }
int readint_c(int c) {  // tine cont de semn; c: primul caracter
  return c == '-' ? - readnat() :
         c == '+' ? readnat() : readnat_rc(0, c);
}
int readint(void) { return readint_c(getchar()); }  // fara param.
int main(void) {
  printf("numarul citit este: %d\n", readint());
  return 0;
}
```


Noțiunea de efect lateral

Un *calcul* pur nu are alte efecte: următorul program nu *scrie* nimic!
`int sqr(int x) { return x * x; }` `int main(void) { sqr(2); }`
Apelul repetat al unei funcții (în matematică, sau din cele scrise până acum: `sqr`, `fact`, etc.) cu aceiași parametri produce același rezultat (repetiția poate fi ineficientă, dar rezultatul e același, ex. `pow2`, `fib`)

În contrast, tipărirea (`printf`) produce un efect vizibil (și ireversibil).
Citirea cu `getchar()` returnează *alt* caracter din intrare la fiecare apel; caracterul e consumat.

O modificare în starea mediului de execuție a programului se numește *efect lateral* (ex. citire, scriere, atribuire – v. ulterior).
Uneori e necesar să *memorăm* o valoare (caracter citit de la intrare, pentru a nu se pierde sau rezultat de funcție, pentru a nu-l recalcula).

Vom discuta cum se face aceasta prin *atribuire* la o *variabilă*.