Recursivitatea: Noțiuni fundamentale

Recursivitatea e un concept fundamental în matematică și informatică.
Un obiect (o noțiune) e recursiv(ă) dacă e folosit în propria sa definiție.

Exemplu din matematică: șiruri recurente
– progresie aritmetică: $x_0 = a$, $x_n = x_{n-1} + p$, pentru $n > 0$
– șirul lui Fibonacci: $F_0 = 1$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ pentru $n > 1$.

E înrudită cu iterația: ambele implică repetiție, dar în mod diferit:
Ex.: descrierea unui obiect compus (un șir)
– iterativ: un *șir* e un *obiect*, urmat de alt *obiect*, urmat ... (*repetiție*)
– recursiv: un *șir* e un *obiect*, sau un *obiect* urmat de un *șir*
(noțiunea definită (șir) apare din nou în corpul definiției)

Exemplu: funcția factorial

Matematic:

$$n! = \begin{cases} 1 & \text{pt. } n = 0 \\ n \cdot (n-1)! & \text{pt. } n > 0 \end{cases}$$

```
unsigned fact(unsigned n)
{
  if (n == 0) return 1;
  else return n * fact(n-1);
}
```

```
unsigned fact_nonrec(unsigned n)
{
  int p = 1;
  while (n > 0) {
    p = p * n;
    n = n - 1;
  }
  return p;
}
```

sau: `unsigned fact(unsigned n) { return n ? n * fact(n-1) : 1; }`
⇒ transcriere practic directă din formularea matematică
valoarea factorialului se acumulează automat în expresia returnată
(în varianta nerecursivă, e necesară o variabilă suplimentară `p`)

Obs.: am ales `unsigned n;` pentru `int` trebuie tratat cazul < 0
Întrebare: pentru ce valoare $n!$ depășește cel mai mare `unsigned` ?

Recursivitate și inducție

O definiție recursivă trebuie să fie *bine formată*
– o noțiune nu se poate defini *doar* în funcție de sine însuși ($x = x$)
– o definiție recursivă se poate folosi *numai* de noțiuni deja definite
NU: $x_n = x_{n+1} - 1$, pentru $n \geq 0$ (x_{n+1} nu e încă definit)
⇒ orice șir de apeluri de funcții recursive trebuie să se oprească
(nu va genera un calcul infinit)

În general, distingem (asemănător ca și la inducția matematică)
– un *caz de bază* (pentru care noțiunea e definită direct) (ex. $a^0 = 1$)
– un *pas inductiv* (recursivitatea propriu-zisă) (ex. $a^{n+1} = a^n * a$)
(noțiunea e definită folosind aceeași noțiune, dar pe un caz mai simplu)

Cum ne asigurăm de oprirea recursivității (la cazul de bază)?
– dacă avem un indice explicit (ex. la șiruri):
când definiția pentru $n + 1$ se folosește doar de valorile pt. indici $\leq n$.
– sau cu altă valoare care scade până ajunge la 0 în cazul de bază

Recursivitatea în limbajul C

– funcții recursive: care se apelează pe sine însuși (ex. factorial)
– sau indirect recursive:
Ex: pentru o expresie aritmetică putem defini următoarea *gramatică*
expresie ::= termen | *expresie* + *termen* | *expresie* - *termen*
termen ::= factor | *termen* * *factor* | *termen* / *factor*
factor ::= număr | *(expresie)*
implementăm *expresie*, *termen*, *factor* cu câte o funcție:
`e` apelează pe `t`, `t` apelează pe `f`, iar `f` pe `e`
– tipuri de date recursive (tip structură cu câmp pointer la acel tip,
de exemplu pentru liste înlănțuite)

```
typedef struct l {
  int info;    /* sau mai multe câmpuri cu diverse date */
  struct l *next;    /* pointer la acelaşi tip */
} list_t;        /* definim list_t ca nume de tip pentru struct l */
```


Recursivitate și iterație

Recursivitatea și iterația sunt noțiuni strâns legate.

Semantica unui ciclu (cu test inițial) poate fi definită recursiv:

```
while ( cond )
  instrucțiune;
```

```
if ( cond ) {
  instrucțiune;
  while ( cond )
    instrucțiune;
}
```

Orice program recursiv poate fi transformat *mecanic* într-unul iterativ:
– în cel recursiv, *stiva* pentru apelurile de funcție și variabilele locale e gestionată explicit de compilator
– în cel iterativ, poate fi necesar să folosim explicit o stivă (sau doar să modificăm variabilele în ciclu, similar ca în apelul recursiv)

– în general, codul scris folosind doar iterația e mai eficient
– dar o soluția recursivă e adesea cea mai simplă, elegantă și naturală

Exemplu: șirul Fibonacci

```
unsigned fib(unsigned n) {
  if (n <= 1) return 1;
  else return fib(n-1)+fib(n-2);
}
unsigned fib_smart(unsigned n) {
  unsigned f2 = 1, f1 = 1, f;
  if (n <= 1) return 1;
  while (--n) { f = f1 + f2;
    f2 = f1;    f1 = f;
  }
  return f;
}
```

```
unsigned fib_nonrec(unsigned n)
{
  int i, *f, res;
  if (n <= 1) return 1;
  f = malloc(n * sizeof(int));
  f[1] = f[0] = 1;
  for (i = 2; i < n; i++)
    f[i] = f[i-1] + f[i-2];
  res = f[n-1] + f[n-2];
  free(f);
  return res;
}
```

Atenție ! Traducerea directă a formulei rezultă în apeluri redundante !
(se apelează din nou `fib(n-2)` și în `fib(n-1)`, etc.)
⇒ nr. de apeluri (câte pt. fib(5)?) e exponențial în `n` (f. ineficient)

Recursivitate: evitarea redundanței

– prin memorarea valorilor intermediare necesare
  – calcul ordonat ca rezultatele intermediare să fie disponibile când sunt necesare (rezolvare de jos în sus, `fib_nonrec`)
  – calculul valorilor după cum devin necesare (de sus în jos)

```
#define MAX   45
unsigned f[MAX] = {1, 1}; /* restul zero */
unsigned fib_memo(unsigned n)  /* doar pt. n < MAX */
{
  if (f[n]) return f[n]; /* nenul -> deja calculat */
  else return f[n] = fib_memo(n-1) + fib_memo(n-2);
  /* memorăm în f[n] înainte de a returna valoarea */
}
```

Câte apeluri se efectuează pentru `fib_memo(5)` ?

Economisirea de apeluri pt. cazul terminal

Adesea, cazul de bază e f. simplu (ex. test și returnarea unei valori). In comparație, costul unui apel de funcție poate fi semnificativ.
⇒ Putem trata cazul de bază fără a mai face un apel suplimentar.

```
#define MAX   45
unsigned f[MAX+1] = {1, 1}; /* restul zero */
unsigned fib_r(unsigned n)  /* pentru n >= 2 */
{
  return f[n]=(f[n-1]?f[n-1]:fib_r(n-1))+(f[n-2]?f[n-2]:fib_r(n-2));
  /* testăm f[k] pt. a decide dacă să apelăm recursiv sau nu */
}
unsigned fib_main(unsigned n) /* se apelează de utilizator */
{
  if (n <= 1) return 1;
  else if (n > MAX) return UINT_MAX; /* prea mare */
  else return fib_r(n);
}
```


Exemplu: afișarea și citirea unui șir

Un șir privit recursiv: șir vid sau caracter urmat de un șir.

```
void puts(char *s)
{
  if (*s) { putchar(*s); puts(s+1); } /* un caracter + restul */
  else putchar ('\n'); /* terminăm afișarea cu linie nouă */
}
```

Citirea unei linii de text într-o zonă de memorie alocată dinamic

```
char *getline(int n) { /* apelăm inițial cu getline(0) */
  char c, *s;
  if ((c = getchar()) == '\n') {
    if (!(s = malloc(n+2))) return NULL;
    s[n+1] = '\0';
  } else s = getline(n+1);
  if (s) s[n] = c;
  return s;
}
```


Exemplu: inversarea unui șir

Inversare recursivă: invers(caracter + rest) = invers(rest) + caracter. (pasul de prelucrare e *după* apelul recursiv); invers(șir vid) = șir vid;

```
void putrev(char *s)
{
  if (*s) { putrev(s+1); putrev(*s); } /* else nimic */
} /* tipărim \n separat după apel */
```

Obs: apelăm cu `s+1`, dar păstrăm `s` nemodificat în funcție, NU `s++` !

Citirea unei linii de text și afișare în ordine inversă:

```
void readrev(void) {
  char c; /* câte o variabilă distinctă pentru fiecare apel */
  if ((c = getchar()) != '\n') readrev(); /* continuă */
  putchar(c); /* la revenire, tipărește caracterul memorat */
}
```

Obs: avem o instanță diferită a lui `c` (caracterul curent) la fiecare apel!

Exemplu: minimul dintr-un tablou

Ideea: *reducerea* la aceeași problemă cu elemente mai puține (cu unul)
Caz de bază: minimul unui tablou de un element e acel element Recursiv: minimul dintre primul element și minimul celor următoare

```
double min_rec(double tab[], unsigned len)
{
  if (len == 1) return *tab; /* unicul element */
  else {
    double min1 = min_rec(tab + 1, len - 1);
    return *tab < min1 ? *tab : min1;
  }
}
```


Exemplu: scrierea unui număr în baza 10

Cazul de baza: $n < 10$, are o singură cifră
Recursiv: cifrele lui $n/10$, urmate de cifra unităților

```
void dec_print(unsigned n)
{
  if (n > 9) dec_print(n/10);
  putchar(n % 10 + '0');
}
```

Tipărirea cifrei e *după* apelul recursiv, ca în descrierea în cuvinte.
Similar, tipărirea pe biți a unui număr:

```
void bit_print(unsigned n)
{
  if (n >> 1) bit_print(n >> 1);
  putchar(n & 1 ? '1' : '0');
}
```

Exemplu: rădăcina unei funcții crescătoare

f crescătoare cu $f(a) < 0$, $f(b) > 0$ are rădăcină în $[a, b]$
– căutăm pe care jumătate a intervalului schimbă semnul și continuăm
– oprire: când se atinge precizia dorită

```
#define EPS 0.001
double f(double x) { return exp(x) - sin(x) - 1.5; }
/* f(0) < 0, f(1) > 0 */
double root(double a, double b)
{
  double m = (a+b)/2, z;
  if (b - a < EPS) return m;
  z = f(m);
  if (z == 0) return m;
  else if (z < 0) return root(m, b); /* f(b > 0 */
  else return root(a, m); /* f(a) < 0 */
}
```


Descompunerea în subprobleme

Una din principalele aplicații ale recursivității: rezolvarea unei probleme prin descompunerea în subprobleme mai mici.

Exemplu: Să se genereze toate șirurile de n cifre binare (în total 2^n)
Definiție recursivă: $n-1$ cifre binare, urmate de 0 sau 1
(descompunere în două subprobleme mai mici)

```
#define N 10
char s[N+1];
s[N] = '\0';
bitstrings(s, 0);
```

```
void bitstrings(char *s, int n) {
  if (n == N) puts(s);
  else {
    s[n] = '0'; bitstrings(s, n+1);
    s[n] = '1'; bitstrings(s, n+1);
  } /* completează de la început */
}
```


Eliminarea recursivității

Exemplu: `puts`: tipărirea unui șir de caractere, urmat de '\n'
– pentru șirul vid ('\0'), tipărește '\n'
– altfel, tipărește primul caracter, tipărește restul șirului

```
void puts(char *s)
{
  if (*s) {
    putchar(*s);
    puts(s+1);
  } else putchar('\n');
}
```

```
void puts(char *s)
{
  while (*s) {
    putchar(*s);
    s = s+1;
  }
  putchar('\n');
}
```

În cazul recursivității la dreapta (apelul recursiv este ultimul lucru)
– transformăm în buclă, cu aceeași condiție de continuare
– actualizăm variabilele cu valorile argumentelor din apelul recursiv

Eliminarea recursivității

Poate fi necesară rescrierea, pentru a aduce apelul recursiv la sfârșit
Ex. factorialul, cu parametru suplimentar produsul acumulat deja

```
int fact_prod(int n, int p)
{
  if (n > 0) {
    p = p * n;
    return fact_prod(n-1, p);
  } else return p;
}
/* apelat cu fact_prod(n, 1) */
```

```
int fact_nonrec(int n)
{
  int p = 1;
  while (n > 0) {
    p = p * n;
    n = n - 1;
  }
  return p;
}
```

Pentru mai mult de un apel recursiv, e necesară folosirea unei *stive*

Căutare: ghicire din nr. minim de întrebări

```
#include <stdio.h>
void main(void)
{
  unsigned m, lo = 0, hi = (1 << 10) - 1;
  printf("Gândiţi-vă la un număr întreg între 0 şi %d\n", hi);
  do {
    m = (lo + hi) >> 1;
    printf("Numărul e mai mare decât %d ? (d/n) ", m);
    if (tolower(getchar()) == 'd') lo = m+1; else hi = m;
    while (getchar() != '\n');
  } while (lo < hi);
  printf("Numărul este %d !\n", lo);
}
```

Invariant: `lo` $\leq$ nr_căutat $\leq$ `hi`
Altfel spus: găsim pe rând biții din scrierea binară a numărului căutat

Căutare binară: recursiv și nerecursiv

Căutăm cheia `v` în tablou sortat `int a[N]`; returnăm indicele

```
int bsrch_nr(int v, int *a,
             int l, int r) {
  while(l<r) {
    int m = (l+r)/2;
    if (v>a[m]) l=m+1;
    else r=m;
  }
  if (v==a[l]) return l;
  else return -1;
}
```

```
int bsrch(int v, int *a,
          int l, int r) {
  int m;
  if (l<r) {
    int m = (l+r)/2;
    if (v>a[m])
      return bsrch(v, a, m+1, r);
    else
      return bsrch(v, a, l, m);
  } else if (v==a[l]) return l;
  else return -1;
}
```

```
void *bsearch(const void *key, const void *base, size_t nmemb,
       size_t size, int (*compar)(const void *, const void *));
```