INFORMATICĂ · CLASA A XI-A · RECURSIVITATE · LECȚIA 3

Recursivitate (1): mecanism, stivă, funcții recursive simple

Durată: 60 min Nivel: mate-info Înainte: lecțiile xi-02

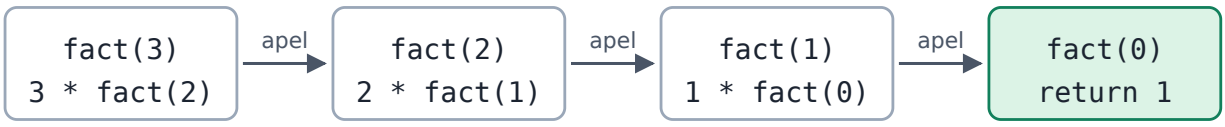
CE VEI ȘTI DUPĂ LECȚIE

- Înțelegi ce este o funcție recursivă: condiția de oprire + apelul pentru o problemă mai mică.
- Urmărești execuția unei funcții recursive pe stivă (arborele de apeluri).
- Scrii recursiv: factorial, putere, cmmdc, suma și numărul cifrelor, cifra maximă.
- Rezolvi grilele de tip „ce valoare are f(...)?" și „ce afișează apelul...?".

1. Ideea: problema mare = problema mică + un pas

O funcție este **recursivă** dacă se apelează pe ea însăși. Gândește-te la factorial: 5! = 5 · 4!, iar 4! = 4 · 3!, și tot așa până la 0! = 1. Soluția pentru n se obține din soluția aceleiași probleme pentru n − 1.

```
long long fact(int n)
{
    if (n == 0) return 1;           // 1) cazul de bază: răspunsul îl știm direct
    return n * fact(n - 1);         // 2) pasul recursiv: o problemă mai mică
}
```



COBORÂRE: apelurile se adună pe stivă până la cazul de bază



URCARE: fiecare apel își primește rezultatul și se termină

Stiva de apeluri

Orice funcție recursivă corectă are două ingrediente:

1. **cazul de bază** (condiția de oprire): situația în care răspunsul e cunoscut și nu mai apelăm;
2. **pasul recursiv**: un apel pe o problemă *strict mai mică*, care se apropie de cazul de bază.

DE CE AȘA?

De ce funcționează? Presupunem că `fact(n - 1)` ne dă corect răspunsul (acesta e „actul de credință" al recursivității) și doar îl completăm cu pasul `n *`. Cazul de bază ne asigură că lanțul de apeluri se termină.

CAPCANĂ

Fără caz de bază (sau dacă apelul nu se apropie de el), funcția se apelează la nesfârșit până se umple memoria alocată stivei: **stack overflow**. Verifică mereu: există un caz de bază? Fiecare apel se apropie de el?

```cpp
#include <iostream>
using namespace std;

long long fact(int n)
{
    if (n == 0) return 1;
    return n * fact(n - 1);
}

int main()
{
    int n;
    cin >> n;
    cout << fact(n);
    return 0;
}
```

Intrare / ieșire

Intrare	Ieșire
`5`	`120`
`0`	`1`
`20`	`2432902008176640000`

2. Ce se întâmplă în memorie: stiva

Fiecare apel primește propriul „cadru" pe **stivă**, cu propriile copii ale parametrilor și variabilelor locale. Apelurile se adâncesc până la cazul de bază (coborâre), apoi se întorc în ordine inversă, fiecare terminându-se cu rezultatul calculat (urcare). Vezi figura de mai sus: `fact(3)` așteaptă `fact(2)`, care așteaptă `fact(1)`, care așteaptă `fact(0)`.

moment	cadre pe stivă (sus = cel mai nou)
după `main` apelează `fact(3)`	`fact(3)`
`fact(3)` apelează `fact(2)`	`fact(3)`, `fact(2)`
`fact(2)` apelează `fact(1)`	`fact(3)`, `fact(2)`, `fact(1)`

moment	cadre pe stivă (sus = cel mai nou)
`fact(1)` apelează `fact(0)`	`fact(3)`, `fact(2)`, `fact(1)`, `fact(0)`
`fact(0)` returnează 1	`fact(3)`, `fact(2)`, `fact(1)`
`fact(1)` returnează 1·1 = 1	`fact(3)`, `fact(2)`
`fact(2)` returnează 2·1 = 2	`fact(3)`
`fact(3)` returnează 3·2 = 6	(gol)

3. Funcții recursive pe cifre

Ideea: ultima cifră (`n % 10`) + restul numărului (`n / 10`), care este o problemă mai mică.

```cpp
#include <iostream>
using namespace std;

int sumaCif(int n)
{
    if (n == 0) return 0;                 // fără cifre: suma 0
    return n % 10 + sumaCif(n / 10);
}

int nrCif(int n)
{
    if (n < 10) return 1;                 // o singură cifră
    return 1 + nrCif(n / 10);
}

int cifMax(int n)
{
    if (n < 10) return n;
    int m = cifMax(n / 10);               // cea mai mare cifră din restul numărului
    return (n % 10 > m) ? n % 10 : m;
}

int main()
{
    int n;
    cin >> n;
    cout << sumaCif(n) << ' ' << nrCif(n) << ' ' << cifMax(n);
    return 0;
}
```

Urmărire pentru `sumaCif(4075)`:

apel	n % 10	rest n / 10	returnează
`sumaCif(4075)`	5	407	5 + `sumaCif(407)` = 5 + 11 = **16**
`sumaCif(407)`	7	40	7 + `sumaCif(40)` = 7 + 4 = 11
`sumaCif(40)`	0	4	0 + `sumaCif(4)` = 0 + 4 = 4
`sumaCif(4)`	4	0	4 + `sumaCif(0)` = 4 + 0 = 4

apel	n % 10	rest n / 10	returnează
`sumaCif(0)`			0 (caz de bază)

> **DE CE AȘA?**
>
> **De ce cazul de bază diferă la `sumaCif` (n == 0) și `nrCif` (n < 10)?** Suma cifrelor „fără nicio cifră" are sens (0), dar numărul de cifre al lui 0 nu este 0, ci 1 (scrisul „0" are o cifră). Alege cazul de bază care dă o valoare corectă pentru **cea mai mică intrare** a problemei.

Cmmdc și puterea

Algoritmul lui Euclid este recursiv prin natura lui: cmmdc(a, b) = cmmdc(b, a mod b).

```cpp
#include <iostream>
using namespace std;

int cmmdc(int a, int b)
{
    if (b == 0) return a;
    return cmmdc(b, a % b);
}

long long putere(long long a, int n)        // a^n în O(log n) pași
{
    if (n == 0) return 1;
    long long p = putere(a, n / 2);          // calculat O SINGURĂ DATĂ
    if (n % 2 == 0) return p * p;
    return p * p * a;
}

int main()
{
    int a, b, n;
    long long x;
    cin >> a >> b >> x >> n;
    cout << cmmdc(a, b) << ' ' << putere(x, n);
    return 0;
}
```

> **DE CE AȘA?**
>
> **De ce `putere` face doar log n pași?** Folosim 2^10 = $(2^5)^2$ și 2^5 = $(2^2)^2 \cdot 2$: de fiecare dată jumătate din exponent. Dacă am fi scris `putere(a, n/2) * putere(a, n/2)`, am fi calculat de două ori aceeași valoare și am fi ajuns tot la n pași. Variabila `p` păstrează rezultatul.

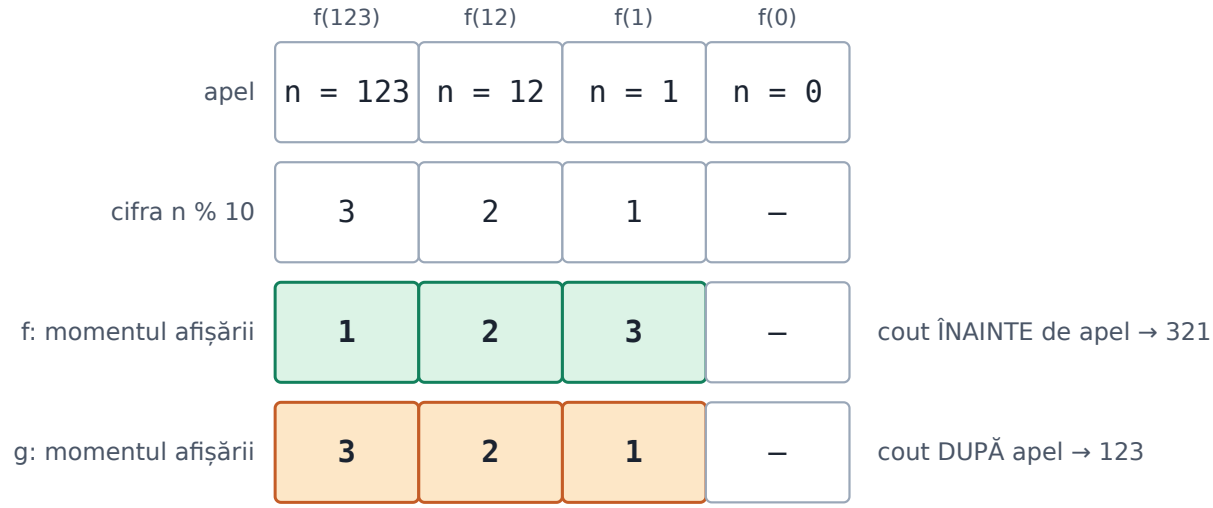
Urmărire pentru `putere(2, 5)`:

apel	n	p (după apel)	returnează
`putere(2, 5)`	5	`putere(2, 2)` = 4	4 · 4 · 2 = **32**
`putere(2, 2)`	2	`putere(2, 1)` = 2	2 · 2 = 4
`putere(2, 1)`	1	`putere(2, 0)` = 1	1 · 1 · 2 = 2

apel	n	p (după apel)	returnează
`putere(2, 0)`	0		1

4. Afișarea „la coborâre” vs. „la urcare”

Locul instrucțiunii `cout` față de apelul recursiv schimbă ordinea!



Ordinea afișării

```cpp
#include <iostream>
using namespace std;

void f(int n)
{
    if (n == 0) return;
    cout << n % 10;      // ÎNAINTE de apel: cifrele de la dreapta la stânga
    f(n / 10);
}

void g(int n)
{
    if (n == 0) return;
    g(n / 10);
    cout << n % 10;      // DUPĂ apel: cifrele de la stânga la dreapta
}

int main()
{
    int n;
    cin >> n;
    f(n); cout << '\n';
    g(n);
    return 0;
}
```

DE CE AȘA?

De ce `g` afișează în ordinea corectă? `g(123)` așteaptă să se termine `g(12)`, care așteaptă `g(1)`. Primul `cout` care se execută este cel din `g(1)` (afișează 1), apoi `g(12)` afișează 2, apoi `g(123)` afișează 3. Stiva răstoarnă ordinea, la fel ca un teanc de farfurii: ultima pusă e prima luată.

Acesta este exact tipul de grilă de la Subiectul I: o funcție cu `cout` înainte și/sau după apel.

```
#include <iostream>
using namespace std;

void h(int n)
{
    if (n > 0)
    {
        cout << n;
        h(n - 2);
        cout << n;
    }
}

int main()
{
    h(5);
    return 0;
}
```

Urmărire pentru `h(5)`:

apel	afișează înainte	apelează	afișează după
`h(5)`	5	`h(3)`	5
`h(3)`	3	`h(1)`	3
`h(1)`	1	`h(-1)`	1
`h(-1)`	(n > 0 fals, nu face nimic)		

Ordinea în timp: `5`, `3`, `1` (la coborâre), apoi `1`, `3`, `5` (la urcare) → **531135**.

LA BAC

La grile desenează arborele (lanțul) de apeluri și notează ce se afișează la coborâre și ce se afișează la urcare. Citirea „pe diagonală” a codului duce la greșeli.

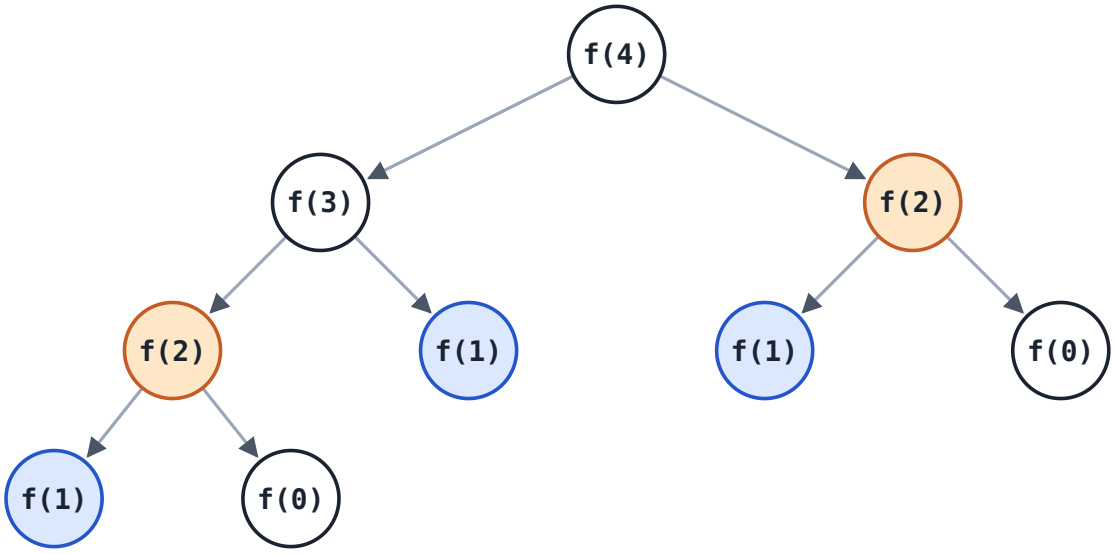
5. Funcții care întorc o valoare: „cât este f(...)”

```cpp
#include <iostream>
using namespace std;

int calls = 0;

int f(int x)
{
    calls++;                            // numărăm câte apeluri se fac
    if (x <= 1) return x;
    return f(x - 1) + f(x - 2);
}

int main()
{
    int n;
    cin >> n;
    cout << f(n) << ' ' << calls;
    return 0;
}
```



f(2) se calculează de 2 ori, f(1) de 3 ori: același lucru refăcut

Arborele de apeluri

`f(4) = f(3) + f(2)`; arborele arată că `f(2)` se calculează de **două ori**, iar `f(1)` de trei ori. Pentru `n = 10` programul face 177 de apeluri ca să afle 55; pentru `n = 40` ar face sute de milioane.

DE CE AȘA?

De ce e lent Fibonacci recursiv? Fiecare apel dă naștere la alte două, și aceleași valori se recalculează în ramuri diferite. Numărul de apeluri crește aproximativ ca $1{,}6^n$. Soluțiile: calculul iterativ (cu doi indici, ca la clasa a IX-a) sau programarea dinamică (lecția XI.16), care „ține minte” rezultatele.

SFAT

Pentru „cât este f(n)" calculează de jos în sus: f(0), f(1), f(2), ... într-un tabel, până la n. Este mult mai rapid decât să desfaci tot arborele.

6. Recursiv sau iterativ?

Orice funcție recursivă se poate scrie și iterativ. Recursivitatea e naturală pentru probleme care „se definesc prin ele însele" (backtracking, divide et impera, arbori), dar consumă memorie pe stivă și, ca la Fibonacci, poate repeta calcule.

	Recursiv	**Iterativ**
claritate	foarte clar pentru probleme recursive	clar pentru bucle simple
memorie	O(adâncimea apelurilor) pe stivă	O(1) de obicei
viteză	apelurile costă puțin	de obicei mai rapid
risc	stack overflow la adâncimi mari	nu există

7. Greșeli frecvente

Greșeala	**Efect**	**Corect**
lipsește cazul de bază	recursie infinită, stack overflow	`if (...) return …;` la început
apelul nu se apropie de cazul de bază (`f(n)` în loc de `f(n - 1)`)	recursie infinită	micșorează problema
cazul de bază greșit (`nrCif(0)` = 0)	răspunsuri greșite	verifică pe cea mai mică intrare
`cout` la locul greșit	ordine inversată	pune-l înainte (coborâre) sau după (urcare)
dublezi apelul (`putere(a, n/2) * putere(a, n/2)`)	timp O(n)	salvează într-o variabilă
ai `return` doar într-o ramură	valoare necunoscută	`return` pe toate căile

8. La BAC

LA BAC

La Subiectul I apar, aproape mereu, una–două grile cu subprograme recursive: „ce se afișează la apelul `f(…)` ?" sau „care este valoarea lui `f(…)` ?". Metoda sigură: tabel cu coloane `apel / afișează înainte / apelează / afișează după / returnează` și se completează de sus în jos, apoi de jos în sus.

9. Exersează pe hârtie

Răspunsurile sunt la final: încearcă întâi singur.

1. Ce afișează `g(4)`?

```
void g(int n) { if (n) { g(n - 1); cout << n; g(n - 1); } }
```

2. Scrie o funcție recursivă care returnează numărul de cifre pare ale lui `n` (pentru `n > 0`).

3. Scrie o funcție recursivă care afișează reprezentarea în baza 2 a lui `n`.

4. Dacă `int f(int x) { if (x <= 1) return x; return f(x - 1) + f(x - 2); }`, cât este `f(6)`?

5. Ce se întâmplă la apelul `int f(int n) { return n * f(n - 1); }`?

6. Ce afișează `h(4)`, cu `h` din secțiunea 4?

7. Calculează `cmmdc(48, 18)` scriind apelurile.

8. Câte apeluri face `putere(2, 5)`? Dar `putere(2, 16)`?

9. Cât returnează `nrCif(0)`? Justifică.

10. (BAC) Ce valoare are `f(5)`?

```
int f(int x) { if (x == 0) return 0; if (x % 2 == 1) return x + f(x - 1); return f(x - 1);
}
```

10. Mai departe (pentru cine vrea mai mult)

Citire și afișare inversă fără vector. Stiva de apeluri poate „ține minte” valorile citite:

```
#include <iostream>
using namespace std;

void citeste(int k)
{
    if (k == 0) return;
    int x;
    cin >> x;               // citim și păstrăm x în cadrul acestui apel
    citeste(k - 1);         // citim restul
    cout << x << ' ';       // la urcare afișăm: ordine inversă
}

int main()
{
    int n;
    cin >> n;
    citeste(n);
    return 0;
}
```

Recursivitate de coadă (tail recursion). Dacă apelul recursiv este ultima operație (`return cmmdc(b, a % b);`), compilatorul poate transforma automat funcția într-o buclă, fără consum de stivă.

Funcția lui Ackermann. Un exemplu celebru de funcție recursivă care crește uimitor de repede: `A(3, 3) = 61`, dar `A(4, 2)` are zeci de mii de cifre. O verifici ca exercițiu, dar nu cu valori mari!

Fractali. Triunghiul lui Sierpinski, fulgul lui Koch și alte desene se definesc recursiv: o figură este făcută din copii mai mici ale ei înșăși. Caută „recursion fractal C++" pentru exemple.

Rezumat

- Recursivitate = caz de bază + apel pe o problemă mai mică.
- Fiecare apel are propriile variabile, pe stivă; apelurile se termină în ordine inversă.
- `cout` înainte de apel → la coborâre; după apel → la urcare (ordine inversă).
- Fibonacci recursiv simplu refăcut același calcul de multe ori; folosește iterativ sau programare dinamică.
- Verifică mereu: cazul de bază corect, apel care se apropie de el.

Probleme de lucru (pbinfo)

Începe cu acestea, în ordine: sunt alese să urmeze pașii din lecție.

- [ ] `#820` **factorialrec** — ușoară
  https://www.pbinfo.ro/probleme/820/factorialrec
- [ ] `#821` **cmmdcrec** — ușoară
  https://www.pbinfo.ro/probleme/821/cmmdcrec
- [ ] `#823` **sumcifrec** — ușoară
  https://www.pbinfo.ro/probleme/823/sumcifrec

Temă (pbinfo)

Trimite soluțiile pe pbinfo și notează punctajul obținut. Bifează ce ai terminat.

- [ ] `#822` **nrcifrezerorec** — ușoară
  https://www.pbinfo.ro/probleme/822/nrcifrezerorec
- [ ] `#824` **cifmaxrec** — ușoară
  https://www.pbinfo.ro/probleme/824/cifmaxrec
- [ ] `#825` **cifminrec** — ușoară
  https://www.pbinfo.ro/probleme/825/cifminrec
- [ ] `#829` **afisarerec** — ușoară
  Afișare în ordine inversă a valorilor citite — fără vector!
  https://www.pbinfo.ro/probleme/829/afisarerec
- [ ] `#918` **sumcifrec1** — ușoară
  https://www.pbinfo.ro/probleme/918/sumcifrec1

- [ ] #920 **cifmaxminrec** — ușoară
  https://www.pbinfo.ro/probleme/920/cifmaxminrec
- [ ] #4209 **difparimpar** — ușoară
  https://www.pbinfo.ro/probleme/4209/difparimpar

Pentru cei care vor mai mult

Probleme mai grele sau de concurs, pentru cine vrea să meargă dincolo de BAC.

- [ ] #1809 **powerrec** — ușoară
  https://www.pbinfo.ro/probleme/1809/powerrec
- [ ] #4042 **b16-rec** — ușoară
  https://www.pbinfo.ro/probleme/4042/b16-rec

Grile din subiectele de BAC (pbinfo)

Exerciții de tip grilă din variantele de Bacalaureat, rezolvate online pe pbinfo, cu verificare automată a răspunsului.

- [ ] #585 **Exercițiul #585**
  funcția F: ce valoare are F(3)?
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=585
- [ ] #586 **Exercițiul #586**
  subprogramul f: ce se afișează în urma apelului dat
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=586
- [ ] #590 **Exercițiul #590**
  subprogram recursiv cu return: valoarea unui apel
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=590
- [ ] #592 **Exercițiul #592**
  subprogramul recursiv suma(a, b): valoarea apelului
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=592
- [ ] #596 **Exercițiul #596**
  subprogramul afis(n): ce se afișează
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=596
- [ ] #597 **Exercițiul #597**
  subprogramul scrie(x, y): ce se afișează
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=597
- [ ] #611 **Exercițiul #611**
  f recursivă cu condiție de oprire pe x < 1: valoarea apelului
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=611

- [ ] #618 **Exercițiul #618**
  f(n) recursivă: valoarea pentru un n dat
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=618

Citește și exersează în plus

- Recursivitate https://www.pbinfo.ro/articole/3873/recursivitate
- Funcții recursive pentru prelucrarea cifrelor unui număr https://www.pbinfo.ro/articole/33003/functii-recursive-pentru-prelucrarea-cifrelor-unui-numar
- VisuAlgo — Recursion Tree / Call Stack — animație a arborelui de apeluri și a stivei (engleză) https://visualgo.net/en/recursion
- Python Tutor pentru C++ — urmărește fiecare apel recursiv și valorile din cadrele stivei https://pythontutor.com/cpp.html
- LearnCpp — Recursion — explicații cu exemple, inclusiv pericolul depășirii stivei (engleză) https://www.learncpp.com/cpp-tutorial/recursion/

Răspunsuri

Exersează pe hârtie

1. `121312141213121`. `g(1)` afișează `1`; `g(2)` = `g(1)`, `2`, `g(1)` = `121`; `g(3)` = `121`, `3`, `121` = `1213121`; `g(4)` = `1213121`, `4`, `1213121`.

2.

```cpp
#include <iostream>
using namespace std;

int nrPare(int n)
{
    if (n == 0) return 0;
    return (n % 10 % 2 == 0) + nrPare(n / 10);     // expresia booleană valorează 0 sau 1
}

int main()
{
    int n;
    cin >> n;
    cout << nrPare(n);
    return 0;
}
```

3.

```cpp
#include <iostream>
using namespace std;

void bin(int n)
{
    if (n > 1) bin(n / 2);      // mai întâi biții mai semnificativi (la urcare îi scriem)
    cout << n % 2;
}

int main()
{
    int n;
    cin >> n;
    bin(n);
    return 0;
}
```

4. `f(6) = 8` (șirul 0, 1, 1, 2, 3, 5, 8).

5. Nu există caz de bază: funcția se apelează la infinit (`f(n-1)`, `f(n-2)`, ... trece de 0 în numere negative), stiva se umple și programul se oprește cu *stack overflow*.

6. `4224` : `h(4)` afișează 4, apelează `h(2)` (afișează 2, apelează `h(0)` care nu face nimic, apoi afișează 2) și apoi afișează 4.

7. `cmmdc(48, 18)` → `cmmdc(18, 12)` → `cmmdc(12, 6)` → `cmmdc(6, 0)` = **6**.

8. `putere(2, 5)`: apeluri pentru n = 5, 2, 1, 0: **4**. `putere(2, 16)`: n = 16, 8, 4, 2, 1, 0: **6**. (Crește cu 1 la fiecare dublare a exponentului.)

9. `1`. Cazul de bază `n < 10` prinde și `0`, iar numărul 0 se scrie cu o cifră.

10. `9`: `f(5) = 5 + f(4)`, `f(4) = f(3)`, `f(3) = 3 + f(2)`, `f(2) = f(1)`, `f(1) = 1 + f(0) = 1`. Deci `f(5) = 5 + 3 + 1 = 9` (suma numerelor impare până la 5).
