test-e.ro

CAIET ELEV

Lecția 18

Eficiența algoritmilor. Măsurarea timpului. Primele ordine de complexitate

*Până acum am scris programe care **funcționează**. De astăzi ne întrebăm **cât costă** un program să funcționeze: cât timp durează și câtă memorie ocupă — și învățăm să argumentăm alegerea unui algoritm cu **numere**, nu cu impresii.*

DISCIPLINA Informatică **CLASA** a IX-a · mate-info **LIMBAJ** Python 3

NUMELE ELEVULUI .....

CUM FOLOSEȘTI CAIETUL

Rezolvă exercițiile pe hârtie: urmărești execuția pas cu pas, numeri pașii unei bucle și scrii singur soluțiile. Noutatea lecției este `import time` cu `time.perf_counter()` — măsurarea timpului. Rămân la dispoziție `while`, `for ... in range()`, `if/elif/else`, `%`, `//` și listele. **Fără** `def`, fără `len` / `min` / `max` / `sum`, fără `import math` — deci **fără** `math.sqrt`: oprirea se scrie `d * d <= n`. Cifrele se parcurg tot cu `while`, prin `% 10` și `// 10`, fără `break`. Regula de aur: **numărul de pași îl calculezi tu, înainte de a rula.**

CUPRINS

1 Ce știm despre bucle, contoare și CMMDC .....
2 Cât costă un program să ruleze .....
3 Numărăm pașii, nu secundele .....
4 O(1), O(n), O(n²) — trei tipare .....
5 Până la n sau până la radical din n? .....
6 Euclid, în două variante .....
7 Cod ghidat · cinci probleme .....

SECȚIUNEA 1 · REACTUALIZARE: BUCLE, CONTOARE, CMMDC

1 Ce știm despre bucle, contoare și CMMDC

Înainte de lecția nouă reîmprospătăm trei unelte vechi: bucla `while`, **contorul** — o variabilă care crește cu 1 de fiecare dată când se întâmplă ceva — și algoritmul lui Euclid cu împărțiri, de la lecția 11.

Un **contor** este o variabilă obișnuită, pornită de la 0, pe care o creștem cu 1 de fiecare dată când se întâmplă ceva care ne interesează. Cu ea am numărat divizori, cifre și numere pare. Iată determinarea numărului de divizori pozitivi ai numărului natural `n`:

```
c = 0
d = 1
while d <= n:
    if n % d == 0:
        c = c + 1
    d = d + 1
```

Algoritmul lui Euclid cu împărțiri: cât timp `b` nu e zero, calculăm restul `r = a % b`, apoi `a` ia valoarea lui `b`, iar `b` ia valoarea lui `r`. Când `b` devine 0, în `a` a rămas cel mai mare divizor comun.

> **ATENȚIE**
>
> Contorul se inițializează **o singură dată, înainte de buclă**. Dacă îl pui din greșeală în interiorul buclei, el se resetează la fiecare pas și la final nu numeri nimic. E o greșeală mică la scris, dar mare ca efect — o revedem în secțiunea 7.

A EXERCIȚIUL A · GRILĂ

Recapitulare: rest, divizori, CMMDC, bucle

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Ce afișează `print(17 % 5)`? (2) (3) (3.4) (12)

2. Câți divizori are numărul 12?

   (6 (1, 2, 3, 4, 6, 12)) (4 (doar 1, 2, 6, 12)) (12 (toate de la 1 la 12)) (2, pentru că 12 e par)

3. Cât este CMMDC(48, 18)? (6) (18) (12) (2)

4. De câte ori se execută corpul buclei `for i in range(1, 8):`?

`de 7 ori` `de 8 ori` `de 6 ori` `o singură dată`

HINT `%` *dă restul, nu câtul zecimal;* `range(1, 8)` *îl include pe 1, dar nu pe 8.*

B

EXERCIȚIUL B · DRY-RUN

Euclid cu împărțiri, pas cu pas

DRY-RUN

Urmărește algoritmul pentru `a = 48` și `b = 18`. Coloanele `a` și `b` îți sunt date; tu completezi restul `r = a % b` pe fiecare rând. Primul rând e completat ca exemplu, ultimul e starea în care bucla se oprește.

```
1  a = 48
2  b = 18
3  while b != 0:
4      r = a % b
5      a = b
6      b = r
7  print(a)
```

a (dat)	b (dat)	r = a % b
48	18	12
18	12	
12	6	
6	0	—

→ Ce afișează programul? _____ Câte împărțiri a făcut bucla? _____

HINT *pe fiecare rând restul se calculează din* `a` *și* `b` **de pe acel rând** *, nu din valorile inițiale.*

✦

SPRIJIN · SCRIE CODUL

Câți divizori pari are numărul?

SCRIERE COD

Se citește un număr natural `n` (`n >= 1`). Afișează câți divizori pari are `n`, fără text explicativ. Ex.: pentru `n = 12` se afișează `4`, pentru că divizorii sunt 1, 2, 3, 4, 6, 12, iar pari sunt 2, 4, 6 și 12.

HINT *încerci pe rând toate valorile de la 1 la* `n`*; contorul crește doar dacă valoarea este divizor al lui* `n` **și** *, în același timp, este pară.*

SOLUȚIA TA · PYTHON

SPRIJIN · SCRIE CODUL

CMMMC-ul a două numere

SCRIERE COD

Se citesc două numere naturale `a` și `b` (`a >= 1`, `b >= 1`), fiecare pe câte o linie. Afișează cel mai mic multiplu comun, folosind formula `cmmmc = a * b // cmmdc`, unde `cmmdc` se calculează cu algoritmul lui Euclid cu împărțiri. Ex.: pentru 48 și 18, cmmdc = 6, deci se afișează `144`.

HINT *algoritmul lui Euclid consumă valorile pe care le primește, iar formula are nevoie de* `a` *și* `b` *nemodificate. Lucrează pe două copii (*`x` *și* `y`*). Fără atribuire simultană.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 2 · TIMP ȘI MEMORIE

2 Cât *costă* un program să ruleze

Un program corect nu e neapărat un program bun. Pe lângă rezultat contează și **prețul**: cât timp durează și câtă memorie ocupă. Orice program consumă două resurse — **spațiul de memorie** (câte valori memorează simultan) și **timpul de executare**. Cele două nu cresc neapărat la fel:

```
# Varianta A - memoreaza toate valorile: spatiu O(n)
numere = []
i = 1
while i <= n:
    numere.append(i)
    i = i + 1
# Varianta B - retine doar suma: spatiu O(1)
s = 0
i = 1
while i <= n:
    s = s + i
    i = i + 1
```

CELE DOUĂ RESURSE SE MĂSOARĂ SEPARAT

Ambele variante fac `n` pași — același cost de **timp**. Dar varianta A ține minte `n` valori deodată, iar B doar una: pentru `n = 1 000 000`, prima ocupă un milion de căsuțe, a doua una singură. Când spunem că un algoritm este „mai bun", trebuie mereu să precizăm **din ce punct de vedere**.

Timpul se măsoară simplu: citim un moment înainte de algoritm, un altul după, și le scădem. Momentul se citește cu `time.perf_counter()`, care întoarce un număr real, măsurat foarte fin. Ceasul stă în **modulul** `time`, iar linia `import time`, scrisă o singură dată la începutul programului, îl aduce în lucru.

```
import time
start = time.perf_counter()
s = 0
i = 1
while i <= n:
    s = s + i
    i = i + 1
final = time.perf_counter()
durata = final - start
```

Durate măsurate real pentru bucla de mai sus, care adună numerele de la 1 la `n`:

N	DURATA MĂSURATĂ
100 000	≈ 0,004 s
1 000 000	≈ 0,043 s
10 000 000	≈ 0,406 s

De fiecare dată când `n` se înmulțește cu 10, durata se înmulțește tot cu aproximativ 10 — semnătura unui algoritm **liniar**.

> **ATENȚIE**
>
> Un singur `perf_counter()` nu măsoară nimic: are sens doar **diferența** dintre două repere. Iar durata măsurată **depinde de calculator** — pentru valori mici de `n` ea e atât de scurtă încât, exprimată în secunde întregi, iese pur și simplu **0**. De aceea, ca să comparăm doi algoritmi, ne vom baza pe **numărul de pași**.

A

EXERCIȚIUL A · GRILĂ

Timp, memorie, măsurare

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Ce două resurse consumă un algoritm atunci când rulează?

`timpul de executare și spațiul de memorie` `numărul de linii și de variabile` `viteza tastaturii și mărimea ecranului` `numele și extensia fișierului`

2. De ce doi elevi pot obține durate diferite rulând exact același program?

`durata depinde de calculatorul pe care rulează`

`programul dă alt rezultat la fiecare rulare`

`perf_counter() alege un număr la întâmplare` `unul dintre ei a scris greșit programul`

3. Ce obținem din apelul `time.perf_counter()`?

`un moment de timp, din care putem calcula durate`

`durata programului, în secunde întregi` `numărul de pași făcuți de algoritm`

`ora exactă, în format oră:minut:secundă`

HINT *`perf_counter()` nu spune „cât e ceasul", ci dă un reper; singurul lucru cu sens este diferența dintre două repere.*

B

EXERCIȚIUL B · ASOCIERE

Timp sau spațiu?

ASOCIERE

Scrie în căsuță litera resursei despre care vorbește fiecare afirmație. Fiecare literă se folosește o singură dată.

☐ „Programul ține minte, în același timp, toate cele `n` valori citite."

☐ „Bucla se execută de `n` ori până când programul afișează rezultatul."

☐ „Se citește cu `time.perf_counter()`, iar rezultatul diferă de la un calculator la altul."

☐ „Rămâne neschimbat oricât de mare ar fi `n`, pentru că se folosesc doar două variabile."

`A Timp de executare — crește cu n` `B Spațiu de memorie — constant`

`C Spațiu de memorie — crește cu n` `D Timp de executare — măsurat în secunde`

HINT *întreabă-te de fiecare dată dacă afirmația vorbește despre câte valori sunt ținute minte sau despre cât durează.*

C

EXERCIȚIUL C · ORDONARE

Pașii unei măsurători de timp

ORDONARE

Numerotează pașii de la 1 la 4, în ordinea corectă a unei măsurători de timp.

☐ Citim momentul de final cu `time.perf_counter()`.

- [ ] Rulăm algoritmul pe care vrem să-l măsurăm.
- [ ] Scădem startul din final și obținem durata.
- [ ] Citim momentul de start cu `time.perf_counter()`.

HINT *algoritmul măsurat stă mereu* **între** *cele două citiri de ceas; scăderea vine la final.*

D

EXERCIȚIUL D · SCRIE CODUL

Cronometrul: câte împărțiri face Euclid

SCRIERE COD

Se citesc două numere naturale `a` și `b` (`>= 1`), fiecare pe câte o linie. Calculează CMMDC-ul cu algoritmul lui Euclid cu împărțiri, numără câte împărțiri face bucla și măsoară cu `time.perf_counter()` cât durează. Afișează pe primul rând CMMDC-ul, pe al doilea durata transformată în număr întreg de secunde cu `int(...)`, iar pe al treilea numărul de pași. Ex.: pentru 48 și 18 se afișează `6`, apoi `0`, apoi `3`.

HINT *al doilea rând va fi 0 la orice date — durata reală e mult sub o secundă, iar* `int(...)` *taie tot ce e după virgulă. Exact acesta e mesajul secțiunii. Startul se citește înainte de buclă, finalul după; contorul se inițializează o singură dată.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 3 · OPERAȚIA ESENȚIALĂ

3 Numărăm *pașii*, nu secundele

Într-un algoritm nu toate instrucțiunile contează la fel. Cele scrise o singură dată, în afara buclelor, se execută o singură dată. Ceea ce contează este **operația pe care algoritmul o repetă cel mai des** — o numim **operație esențială**. La căutarea divizorilor este testul `n % d == 0`; la Euclid cu scăderi, scăderea.

Uneori numărul de pași se vede din felul în care e scrisă bucla: dacă `d` ia toate valorile de la 1 la `n`, testul se face de exact `n` ori. Alteori nu se vede nimic:

```
pasi = 0
while a != b:
    if a > b:
        a = a - b
    else:
        b = b - a
    pasi = pasi + 1
print(pasi)
```

Nimic din `a` și `b` nu ne spune dinainte de câte ori se rotește bucla: pentru `a = 100`, `b = 36` se fac 7 scăderi, iar pentru `a = 1000`, `b = 1` se fac 999. Odată aflat, numărul nu se mai schimbă de la un calculator la altul: **numărul de pași este o proprietate a algoritmului**, nu a mașinii.

> **DE REȚINUT**
>
> Doi algoritmi se compară după numărul de pași făcuți **pentru aceeași valoare de intrare**. „A a durat 0,3 s, iar B 0,2" nu spune nimic dacă A a rulat pe alt `n` sau pe alt calculator. „Pentru `n = 1000`, A face 1000 de pași, iar B face 31" este un argument complet.

A

EXERCIȚIUL A · DRY-RUN

Aceiași divizori, două metode (n = 6)

DRY-RUN

Cele două programe numără divizorii lui `n = 6` și ajung la **același răspuns**, pe drumuri de lungimi diferite. Trasează-le pe rând: coloanele `d` și `pasi` îți sunt date, tu completezi celelalte două. Uită-te apoi la **înălțimea celor două tabele** — acolo se vede diferența.

METODA LENTĂ · WHILE D <= N

```
n = 6
c = 0
pasi = 0
```

METODA RAPIDĂ · WHILE D * D <= N

```
n = 6
c = 0
pasi = 0
```

```python
d = 1
while d <= n:
    pasi = pasi + 1
    if n % d == 0:
        c = c + 1
    d = d + 1
print(pasi, c)
```

d	n % d	c după	pasi
1	0	1	1
2			2
3			3
4			4
5			5
6			6
7	—		6

```python
d = 1
while d * d <= n:
    pasi = pasi + 1
    if n % d == 0:
        c = c + 2
    d = d + 1
print(pasi, c)
```

d	d * d	c după	pasi
1			1
2			2
3		—	2

HINT *la metoda lentă* `c` *se schimbă numai când* `n % d` *este 0; în rest rămâne cu valoarea de pe rândul anterior. La metoda rapidă, de fiecare dată când găsește un divizor* `d`*, se adună* **2** *: pe* `d` *și pe perechea lui,* `n // d`*. Bucla se oprește când* `d * d` *depășește 6.*

B

EXERCIȚIUL B · COMPLETARE

Definiția operației esențiale

COMPLETARE

Completează golurile folosind cuvintele din rezervă (două cuvinte rămân nefolosite).

`pași` `linii` `operație` `secunde` `valoare` `variabile`

Instrucțiunea pe care algoritmul o repetă cel mai des se numește _____ esențială. Ca să comparăm doi algoritmi, numărăm câți _____ fac pentru aceeași _____ de intrare, în loc să măsurăm câte _____ durează.

HINT *cele două cuvinte nefolosite descriu lucruri care* **nu** *spun nimic despre cost: lungimea codului și numărul de nume folosite.*

REMEDIERE · SCRIE CODUL

Câți pași până la prima oprire

SCRIERE COD

Se citește un număr natural `n` (`n >= 1`). Un program caută divizorii lui `n` încercând pe rând valorile de la 1 la `n`, dar se oprește la primul divizor găsit **strict mai mare decât 1**. Afișează câți pași a făcut până s-a oprit; dacă nu găsește niciunul, afișează `n`. Ex.: pentru `n = 12` se afișează `2`; pentru `n = 7` se afișează `7`.

HINT *contorul crește la fiecare valoare încercată, inclusiv la `d = 1`. Oprirea se face doar pentru un divizor strict mai mare decât 1 — și fără `break`, cu o variabilă-semnal în condiția buclei.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 4 · O(1), O(N), O(N²)

4 Trei tipare de *creștere*

Nu ne interesează numărul exact de pași, ci **cum crește** el când datele se măresc. Notația O („O mare") descrie tiparul de creștere, ignorând detaliile mici.

O(1) · O(N) · O(N²)

O(1) — constant. Numărul de pași **nu depinde** de `n`: `print(n % 10)` afișează ultima cifră la fel de repede pentru `n = 7` și pentru un `n` cu douăzeci de cifre. **O(n) — liniar.** Numărul de pași crește **proporțional** cu `n`: dacă `n` se dublează, pașii se dublează. **O(n²) — pătratic.** Pașii cresc ca **pătratul** lui `n`: dublăm `n`, numărul de pași se face de patru ori mai mare. Apare la două bucle una în alta, fiecare de la 1 la `n`.

```
pasi = 0
i = 1
while i <= n:
    j = 1
    while j <= n:
        pasi = pasi + 1
        j = j + 1
    i = i + 1
print(pasi)
```

Diferența dintre cele trei tipare nu se vede la `n` mic, ci la `n` mare:

N	O(1)	O(N)	O(N²)
10	1	10	100
100	1	100	10 000
1000	1	1000	1 000 000

ATENȚIE

Regula se extinde: trei bucle una în alta, fiecare de la 1 la `n`, fac `n³` pași. Pentru `n = 1000` înseamnă un miliard de pași — un program care s-ar termina abia peste câteva minute, deși pare la fel de scurt la scris. **Lungimea codului nu spune nimic despre costul lui.**

A

EXERCIȚIUL A · GRILĂ

Recunoaște ordinul

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Un program care afișează ultima cifră a lui `n`, cu `print(n % 10)`, face…

   `mereu același număr de pași` `atâția pași câte cifre are n` `n pași` `n² pași`

2. Pentru `n = 100` un algoritm face 10 000 de pași, iar pentru `n = 200` face 40 000. Ce ordin are?

`O(n²)` `O(n)` `O(1)` `O(n³)`

3. Ce înseamnă că un algoritm este O(n)?

`numărul lui de pași crește proporțional cu n` | `face exact n pași, nici unul în plus` | `se termină în cel mult o secundă` | `folosește n variabile diferite`

HINT *la întrebarea 2, `n` s-a dublat, iar numărul de pași s-a făcut de patru ori mai mare — exact semnătura tiparului pătratic.*

B

EXERCIȚIUL B · ASOCIERE

Fragment de cod → ordin de complexitate

ASOCIERE

Scrie în căsuță litera ordinului potrivit pentru fiecare fragment. Fiecare literă se folosește o singură dată.

☐	`print(n % 10)` — afișează ultima cifră a lui n
☐	O buclă `while d <= n` care numără divizorii
☐	Două bucle una în alta, fiecare de la 1 la n
☐	Trei bucle una în alta, fiecare de la 1 la n

A $O(n^2)$ | B $O(1)$ | C $O(n^3)$ | D $O(n)$

HINT *numără câte bucle sunt* **una în alta** *: zero bucle → constant, una → liniar, două → pătratic, trei → cubic.*

C

EXERCIȚIUL C · SCRIE CODUL

Câte perechi (i, j) cu j divizor al lui i

SCRIERE COD

Se citește un număr natural `n` (`n >= 0`). Un program parcurge toate perechile `(i, j)`, unde `i` ia valorile de la 1 la `n` și, pentru fiecare `i`, `j` ia valorile de la 1 la `n`. Afișează câte perechi au proprietatea că `j` este divizor al lui `i`. Ex.: pentru `n = 3` se afișează `5`; pentru `n = 0` se afișează `0`.

HINT *variabila `j` trebuie readusă la 1 la începutul fiecărei treceri prin bucla exterioară. Dacă o inițializezi o singură dată, bucla interioară rulează doar la primul `i`.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 5 · O(N) FAȚĂ DE O(√N)

5 Până la n sau până la *radical din n*?

Punem față în față doi algoritmi care dau **exact același rezultat**, dar cu costuri complet diferite.

METODA 1 · TOȚI CANDIDAȚII — METODA 2 · PÂNĂ LA RADICAL DIN N

Metoda 1. Încercăm fiecare valoare `d` de la 1 la `n` și numărăm de câte ori `n % d == 0`. Bucla face **n pași**, deci metoda este O(n). Un număr este prim dacă are exact 2 divizori. **Metoda 2.** Divizorii vin în perechi: dacă `d` îl împarte pe `n`, atunci și `n // d` îl împarte, iar unul dintre cei doi este întotdeauna cel mult √n. Deci e destul să încercăm valorile cât timp `d * d <= n` — restul se deduc. Bucla face doar [√n] pași, deci metoda este O(√n).

```
d = 1
rapid = 0
while d * d <= n:
    rapid = rapid + 1
    d = d + 1
print(rapid)
```

N	METODA 1 (PÂNĂ LA N)	METODA 2 (D * D <= N)
25	25	5
100	100	10
1000	1000	31
1 000 000	1 000 000	1000

În secunde, măsurat real pe numărul prim `n = 1 000 003`: metoda 1 a durat ≈ 0,048 s, metoda 2 ≈ 0,000059 s — de aproape 800 de ori mai rapidă, cu același rezultat. La `n = 10 000 019` raportul crește la aproape 2800.

> **ACELAȘI REZULTAT**
>
> Ambele metode răspund identic la întrebarea „este `n` prim?". Metoda 2 nu ghicește și nu aproximează nimic — doar renunță să mai verifice ceva ce se poate deduce.

A

EXERCIȚIUL A · PREDICȚIE

Cursa: metoda lentă contra metodei rapide

PREDICȚIE

Pentru fiecare valoare a lui `n`, prezice câte iterații face fiecare buclă: cea lentă (`while d <= n`) și cea rapidă (`while d * d <= n`), ambele pornind cu `d = 1`. Completează și raportul, rotunjit la un număr întreg.

n (dat)	iterații metoda lentă	iterații metoda rapidă	de câte ori mai puține
1			
25			
36			
97			
400			

HINT *bucla lentă face exact `n` iterații. Pentru cea rapidă, caută cel mai mare `d` cu `d * d <= n` și numără de la 1 până la el — la `n = 97` nu iese un număr rotund, pentru că 97 nu e pătrat perfect.*

B

EXERCIȚIUL B · SCRIE CODUL

Câți pași face fiecare metodă?

SCRIERE COD

Se citește un număr natural `n` (`n >= 0`). Afișează pe același rând, despărțite printr-un spațiu, numărul de pași făcuți de metoda lentă și numărul de pași făcuți de metoda rapidă. Ex.: pentru `n = 25` se afișează `25 5`; pentru `n = 0` se afișează `0 0`.

HINT *după prima buclă, `d` a rămas cu o valoare mare. Adu-l înapoi la 1 înainte de a doua buclă, altfel a doua nu pornește deloc.*

SOLUȚIA TA · PYTHON

★

PERFORMANȚĂ · SCRIE CODUL

Câte numere prime până la n și cu ce cost

SCRIERE COD

Se citește un număr natural `n` (`n >= 2`). Folosind metoda rapidă, verifică pe rând fiecare număr `k` de la 2 la `n`: el este prim dacă nu are niciun divizor `d` cu `2 <= d` și `d * d <= k`. Afișează pe același rând câte numere prime ai găsit și câți pași a făcut în total bucla interioară. Ex.: pentru `n = 10` se afișează `4 9`; pentru `n = 2` se afișează `1 0`.

HINT *bucla interioară parcurge* **toate** *valorile lui `d` cu `d * d <= k`, chiar dacă a găsit deja un divizor — numărăm pașii verificării complete. Contorul total se inițializează o singură dată, înainte de bucla exterioară.*

SOLUȚIA TA · PYTHON

PERFORMANȚĂ · SCRIE CODUL

Unde se întâlnesc cele două curbe

SCRIERE COD

Se citește un număr natural `n` (`n >= 1`). Pentru fiecare `k` de la 1 la `n`, numără câți pași face metoda rapidă. Afișează pentru câte valori ale lui `k` acest număr de pași **rămâne același ca la k − 1**, adică pentru câte valori metoda rapidă nu urcă o treaptă. Prin convenție, `k = 1` nu se numără. Ex.: pentru `n = 4` se afișează `2`.

HINT *vei avea o buclă în interiorul alteia; contorul interior se readuce la 0 pentru fiecare `k`. Curiozitate: numărul de „salturi" pe intervalul 1…n este exact numărul de pași al metodei rapide pentru `n` — restul valorilor sunt „paliere".*

SOLUȚIA TA · PYTHON

SECȚIUNEA 6 · SCĂDERI CONTRA ÎMPĂRȚIRI

6 Euclid, în *două variante*

Algoritmul lui Euclid de la lecția 11 are două forme. **Varianta cu scăderi:** cât timp cele două numere sunt diferite, îl scădem pe cel mic din cel mare; când devin egale, valoarea comună este CMMDC-ul. **Varianta cu împărțiri:** cât timp `b` nu e zero, calculăm restul `r = a % b`, apoi `a` ia valoarea lui `b` și `b` ia valoarea lui `r`. O împărțire face dintr-o dată munca mai multor scăderi consecutive.

```
scaderi = 0
while x != y:
    if x > y:
        x = x - y
    else:
        y = y - x
    scaderi = scaderi + 1
```

A ȘI B	SCĂDERI	ÎMPĂRȚIRI
48 și 18	4	3
100 și 40	3	2
270 și 192	10	4
1000 și 1	**999**	**1**

ATENȚIE · CAZUL CEL MAI RĂU

Pe date „obișnuite" cele două variante par la fel de bune — 4 pași față de 3. Diferența apare când un număr e mult mai mare decât celălalt: pentru 1000 și 1, scăderile fac 999 de pași, împărțirile unul singur. Un algoritm nu se judecă după exemplul din manual, ci după **cel mai neprietenos caz posibil**. Mai mult, varianta cu scăderi cere ca ambele numere să fie **strict pozitive**: dacă unul este 0, bucla nu se mai oprește niciodată.

A

EXERCIȚIUL A · DRY-RUN

Euclid cu scăderi, pentru a = 91 și b = 26

DRY-RUN

Coloanele `a` și `b` de la începutul pasului îți sunt date; tu completezi valorile după scădere și numărul de pași făcuți până acolo. Primul rând e completat ca exemplu, ultimul e starea în care bucla se oprește.

```
a = 91
b = 26
pasi = 0
while a != b:
    if a > b:
        a = a - b
    else:
        b = b - a
    pasi = pasi + 1
print(a, pasi)
```

a (dat)	b (dat)	a după	b după	pasi
91	26	65	26	1
65	26			
39	26			
13	26			
13	13	–	–	

→ Ce afișează programul? _____

HINT *la fiecare pas se schimbă* **un singur** *număr: cel mai mare scade, cel mic rămâne neatins. Atenție la rândul în care* `a` *a devenit mai mic decât* `b`*.*

B

EXERCIȚIUL B · GRILĂ

Cazul cel mai rău

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Pentru `a = 1000` și `b = 1`, câți pași face varianta cu scăderi? `999` `1000` `1` `500`

2. De ce varianta cu împărțiri e preferată în practică?

   `restul scade mult mai repede decât prin scăderi pe rând` `dă un rezultat mai exact`

   `merge și cu numere negative` `folosește mai puțină memorie`

3. Ce au în comun cele două variante?

   `dau același rezultat, cu număr diferit de pași` `fac exact același număr de pași`

   `doar cea cu împărțiri e corectă` `se opresc când un număr devine 1`

HINT *uită-te în tabelul comparativ: ultima linie (1000 și 1) răspunde singură la prima întrebare.*

C

EXERCIȚIUL C · SCRIE CODUL

Scăderi contra împărțiri: numără pașii

SCRIERE COD

Se citesc două numere naturale `a` și `b` (`>= 1`), fiecare pe câte o linie. Rezolvă CMMDC-ul în ambele variante și afișează pe același rând numărul de pași ai variantei cu scăderi și numărul de pași ai variantei cu împărțiri. Ex.: pentru 45 și 12 se afișează `6 3`.

HINT *a doua variantă are nevoie de valorile* **inițiale** *, dar prima le-a modificat deja: copiază-le în două variabile de lucru înainte de fiecare buclă. Când `a` și `b` sunt egale, scăderile fac 0 pași, iar împărțirile 1.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 7 · COD GHIDAT

7 Scrie cod care *trece testele*

Cinci probleme, în ordine crescătoare de dificultate. Scrie soluția pe hârtie, urmărește-o mental pe exemplul dat și abia apoi tastează-o pe platformă, unde e verificată automat pe 5 cazuri de test.

PROBLEMA 7.1 · COD GHIDAT

1 *Suma cifrelor și numărul de pași*

SCRIERE COD

Se citește un număr natural `n` (`n >= 0`). Afișează pe același rând, despărțite printr-un spațiu, suma cifrelor lui `n` și numărul de pași făcuți de bucla care le extrage. Ex.: pentru `n = 1234` se afișează `10 4`; pentru `n = 0` se afișează `0 0`.

HINT *cifrele se extrag doar cu `% 10` și `// 10`, într-o buclă `while` — nu prin transformarea numărului în șir de caractere.*

SOLUȚIA TA · PYTHON

2

PROBLEMA 7.2 · COD GHIDAT

Câte perechi (i, j) cu i mai mic decât j

SCRIERE COD

Se citește un număr natural `n` (`n >= 0`). Afișează câte perechi `(i, j)` există cu `i` și `j` între 1 și `n` și `i` strict mai mic decât `j`. Ex.: pentru `n = 4` se afișează `6`; pentru `n = 0` și `n = 1` se afișează `0`.

HINT *bucla aceasta face aproximativ jumătate din pașii buclei duble complete, dar ordinul ei rămâne tot O(n²): tiparul de creștere e același, doar constanta e mai mică.*

SOLUȚIA TA · PYTHON

3

PROBLEMA 7.3 · COD GHIDAT

Câți pași economisește metoda rapidă

SCRIERE COD

Se citește un număr natural `n` (`n >= 0`). Afișează cu câți pași mai puțini lucrează metoda rapidă (`while d * d <= n`) față de metoda lentă (`while d <= n`), ambele pornind cu `d = 1`. Ex.: pentru `n = 100` se afișează `90`; pentru `n = 0` se afișează `0`.

HINT *nu ai voie să folosești `math.sqrt`. Numărul de pași ai metodei rapide se află numărând, cu bucla `while d * d <= n`.*

SOLUȚIA TA · PYTHON

4

PROBLEMA 7.4 · REPARĂ CODUL

Contorul care se resetează

SCRIERE COD

Programul de mai jos ar trebui să afișeze câți pași face o buclă dublă în care `i` și `j` iau, fiecare, valorile de la 1 la `n` — adică `n²`. Pentru `n = 4` el afișează însă 4, deși răspunsul corect este 16. Găsește greșeala, corecteaz-o și afișează numărul corect de pași.

```
n = int(input())
i = 1
while i <= n:
    pasi = 0
    j = 1
    while j <= n:
        pasi = pasi + 1
        j = j + 1
    i = i + 1
print(pasi)
```

HINT *greșeala nu e în bucle, ci în* **locul** *unde este pusă o singură linie — compară cu regula din secțiunea 1: unde se inițializează un contor? Programul stricat se oprește cu eroare pentru `n = 0`; varianta ta corectă trebuie să afișeze `0`.*

PROGRAMUL CORECTAT · PYTHON

PROBLEMA 7.5 · PROVOCARE

De la ce n merită metoda rapidă?

SCRIERE COD

Se citește un număr natural `k` (`k >= 1`). Afișează cel mai mic număr natural `n >= 1` pentru care metoda lentă face de cel puțin `k` ori mai mulți pași decât metoda rapidă. Ex.: pentru `k = 10` se afișează `80` — la `n = 80` metoda lentă face 80 de pași, cea rapidă 8, iar 80 ≥ 10 × 8.

HINT *vei avea o buclă în interiorul alteia: cea exterioară încearcă valori pentru `n`, cea interioară numără pașii metodei rapide. Contorul interior trebuie readus la 0 pentru fiecare `n` încercat.*

SOLUȚIA TA · PYTHON