INFORMATICĂ · PYTHON · CLASA A IX-A · PRELUCRĂRI ALE DATELOR NUMERICE (INTENSIV) · LECȚIA I1

Ciurul lui Eratostene și exponențierea rapidă

Durată: 150 min Nivel: intensiv · Python Înainte: lecțiile py9-05, py9-06, py9-07, py9-08, py9-09, py9-14, py9-15

CE VEI ȘTI DUPĂ LECȚIE

- Generezi toate numerele prime până la n cu Ciurul lui Eratostene, memorat într-o listă de valori True/False, și explici de ce tăierea pornește de la i · i.
- Compari, cu număr de pași și timp măsurat, ciurul cu testul de primalitate aplicat fiecărui număr (lecția py9-05).
- Calculezi a la puterea b (și a la puterea b modulo m) cu exponențierea rapidă, explicând rolul ridicării la pătrat și legătura cu biții exponentului.
- Aplici cei doi algoritmi într-un sistem RSA simplificat, la codificarea culorilor prin produs de numere prime și la calculul unor coduri de autentificare.

1. Ciurul lui Eratostene: ideea

În lecția py9-05 ai verificat dacă **un** număr este prim. Dacă îți trebuie **toate** primele până la `n` (de exemplu, ca să alegi coduri, chei sau identificatori), nu mai testezi fiecare număr: le scrii pe toate și **tai multiplii**.

Reprezentarea datelor: o listă `e` cu `n + 1` valori, în care **indicele este numărul**, iar valoarea spune dacă numărul este (încă) candidat la prim: `e[7] == True` înseamnă „7 n-a fost tăiat”.

1. **Inițializare:** toate valorile `True`, apoi `e[0] = e[1] = False`.
2. **Eliminare:** iei pe rând `i = 2, 3, 4, …`. Dacă `i` n-a fost tăiat, este prim, și îi tai toți multiplii.
3. **Lista finală:** indicii rămași `True` sunt exact numerele prime.

Evoluția pentru `n = 30` (am afișat doar numerele tăiate **pentru prima dată** la fiecare etapă):

Etapa	Ce se taie nou	Rămân (după etapă)
`i = 2`	4, 6, 8, …, 30 (14 numere)	2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29
`i = 3`	9, 15, 21, 27	2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29
`i = 5`	25	2, 3, 5, 7, 11, 13, 17, 19, 23, 29

La `i = 4` nu se întâmplă nimic (4 e deja tăiat), iar la `i = 6` ciurul se oprește, pentru că `6 · 6 = 36 > 30`.

De ce rămân doar prime? Un număr compus `x` are un divizor prim `p < x`. Când ciurul ajunge la `p`, îl taie pe `x` ca multiplu al lui `p`. Un număr prim nu are divizori mai mici (în afară de 1), deci nu îl taie nimeni.

2. Codul și cele două „de ce” cu i · i

```python
def ciur(n):
    e = [True] * (n + 1)          # e[k] spune dacă k este prim
    e[0] = e[1] = False
    i = 2
    while i * i <= n:
        if e[i]:                  # i a rămas netăiat, deci e prim
            for j in range(i * i, n + 1, i):
                e[j] = False      # tai multiplii lui i
        i += 1
    return e

e = ciur(50)
prime = [k for k in range(len(e)) if e[k]]   # k din 0..50, păstrat dacă e[k] este True
(vezi py9-13, §4)
print(prime)        # [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
print(sum(ciur(100)))   # 25: True contează ca 1, deci sunt 25 de prime ≤ 100
```

De ce tăierea pornește de la `i · i`, nu de la `2 · i`? Multiplii mai mici ai lui `i`, adică `2i, 3i, …, (i − 1)i`, au un factor mai mic decât `i` și au fost deja tăiați la o etapă anterioară. Exemplu: la `i = 5`, numerele 10, 15, 20 au fost tăiate de 2 sau de 3; primul multiplu „nou” este 25.

De ce bucla exterioară se oprește când `i · i > n`? Pentru că un `i` mai mare ar începe tăierea de la `i · i > n`, deci n-ar mai tăia nimic. Altfel spus: orice compus `x ≤ n` are un factor prim cel mult `√x ≤ √n` (aceeași idee ca la divizori, în py9-05).

Pentru `n = 10⁶`, varianta „`i` până la `n`, tăiere de la `2i`” face 2 775 210 atribuiri `e[j] = False`, iar varianta de mai sus 2 122 048; ambele găsesc aceleași 78 498 de prime. Economia e mică (cele mai multe tăieri le fac 2 și 3), dar argumentul cu `i · i` este cel care justifică oprirea la √n.

3. Ciur sau test de primalitate?

Varianta „din py9-05”: pentru fiecare `x` de la 2 la `n` apelezi o funcție `prim(x)` care caută un divizor până la `√x`. Am numărat împărțirile, respectiv tăierile, și am măsurat timpul (pe calculatorul de test; la tine valorile vor fi altele, raportul asemănător).

n	Împărțiri (test pentru fiecare x)	Tăieri (ciur)	Timp test	Timp ciur
10^5	2 745 694	193 078	≈ 0,2 s	≈ 0,01 s
10^6	67 740 404	2 122 048	≈ 6,5 s	≈ 0,2 s

Intuitiv: testul face pentru fiecare număr prim până la `√x` pași, deci cam `n · √n` în total. Ciurul trece pe la fiecare număr de câteva ori (o dată pentru fiecare factor prim mic), deci face doar ceva mai mult de `n` pași. **Prețul ciurului este memoria:** o listă cu `n + 1` valori. Pentru `n = 10⁶` e în regulă; pentru `n = 10¹²` nu.

SFAT

Un singur număr mare (ex. 10^{12}) → testul până la √n. **Multe întrebări** până la o limită rezonabilă → ciurul, construit **o singură dată**; apoi `e[x]` răspunde instantaneu.

4. Exponențierea rapidă

Calculul clasic al lui a^b face $b - 1$ înmulțiri. Observația cheie: **la exponent par, ridici la pătrat**.

- b par: $a^b = (a^2)^{(b/2)}$ (exponentul se înjumătățește cu o singură înmulțire, $a \cdot a$);
- b impar: $a^b = a \cdot a^{(b-1)}$, iar $b - 1$ este par.

Varianta iterativă parcurge **biții exponentului** de la dreapta la stânga: a devine pe rând $a, a^2, a^4, a^8, \ldots$, iar în rezultat intră doar puterile pentru care bitul este 1. Pentru $13 = 1101_2$: $3^{13} = 3^8 \cdot 3^4 \cdot 3^1$.

```python
def putere(a, b):
    r = 1
    while b > 0:
        if b % 2 == 1:      # ultimul bit al lui b este 1
            r = r * a       # puterea curentă intră în rezultat
        a = a * a           # a, a², a⁴, a⁸, ...
        b = b // 2          # trec la bitul următor
    return r

print(putere(3, 13), 3 ** 13)   # 1594323 1594323
```

`b` (baza 10)	`b` (baza 2)	`a`	`r` după pas
13	1101	3	3
6	110	9	3
3	11	81	243
1	1	6561	1594323

Numărul de pași = numărul de cifre binare ale lui b (vezi py9-07), adică aproximativ $\log_2 b$.

Exponent `b`	Înmulțiri, metoda clasică	Pași (biți), metoda rapidă	Înmulțiri executate de `putere`
13	12	4	7
1 000	999	10	16
10^6	999 999	20	27
10^{18}	aproape 10^{18}	60	84

5. Puteri modulo m: a ** b % m sau pow(a, b, m)

În criptare și autentificare nu interesează numărul uriaș a^b, ci doar **restul** lui la un m. Regula care ne salvează: restul unui produs se poate calcula din resturile factorilor, deci aplici `% m` **după fiecare**

înmulțire și numerele rămân mici.

```python
def putere_mod(a, b, m):
    r = 1
    a = a % m
    while b > 0:
        if b % 2 == 1:
            r = r * a % m
        a = a * a % m
        b = b // 2
    return r

M = 1000000007
print(putere_mod(7, 10**6, M), pow(7, 10**6, M), 7 ** 10**6 % M)
# 880007888 880007888 880007888
```

Metodă (`7` la puterea `b`, modulo `M`)	`b = 10⁶`	`b = 10⁷`
buclă clasică cu `r = r * a % M`	≈ 0,1 s	≈ 0,9 s
`7 ** b % M`	≈ 0,2 s	≈ 7,3 s
`putere_mod(7, b, M)`	sub 0,001 s	sub 0,001 s
`pow(7, b, M)` (predefinită)	sub 0,001 s	sub 0,001 s

`7 ** b % M` este lent nu pentru că `**` ar fi naiv (Python ridică și el rapid la putere), ci pentru că **întâi construiește numărul întreg**: pentru `b = 10⁶` el are 845 099 de cifre. `pow(a, b, m)` cu trei argumente face exact ce face `putere_mod`: exponențiere rapidă cu rest la fiecare pas.

> **DE ȘTIUT**
>
> Ce trebuie să știi să explici: (1) la exponent par se ridică la pătrat, deci exponentul se înjumătățește; (2) numărul de pași crește cu `log₂ b`, nu cu `b`; (3) restul se ia după fiecare înmulțire, ca valorile intermediare să rămână mai mici decât `m²`.

Context: autentificarea. Banca și aplicația ta cunosc un secret `k`. Banca trimite un număr aleator `c` (provocarea), aplicația răspunde cu `pow(c, k, m)`, iar banca verifică același calcul. Secretul nu circulă niciodată prin rețea, iar calculul durează microsecunde chiar pentru exponenți uriași.

6. Aplicație: un sistem RSA simplificat

RSA folosește exact cei doi algoritmi: **numere prime** pentru chei și **exponențiere rapidă modulo** pentru criptare. Iată versiunea „de jucărie", cu numere mici.

1. Alegi două prime (din ciur): `p = 61`, `q = 53`. Calculezi `n = p · q = 3233` și `φ = (p − 1)(q − 1) = 3120`.
2. Alegi `e = 17` (prim cu 3120: `gcd(17, 3120) = 1`, vezi py9-06). **Cheia publică** este `(e, n)`.
3. Cauți **cheia privată** `d` astfel încât `d · e % φ == 1`. Pentru numere mici, o căutare simplă ajunge.
4. **Criptare:** `c = pow(x, e, n)`. **Decriptare:** `x = pow(c, d, n)`.

```python
p, q = 61, 53
n, phi = p * q, (p - 1) * (q - 1)
e = 17
d = 1
while d * e % phi != 1:
    d += 1
print(n, phi, e, d)                       # 3233 3120 17 2753
mesaj = "SALUT"
c = [pow(ord(ch), e, n) for ch in mesaj]  # codul fiecărei litere, criptat
print(c)                                  # [2680, 2790, 2726, 2310, 2159]
print("".join(chr(pow(x, d, n)) for x in c))   # SALUT
# `for x in c` fără paranteze pătrate: aceeași idee ca la liste, dar `join` consumă
elementele pe rând
```

De ce e greu de spart? Ca să afle `d` din cheia publică `(17, 3233)`, atacatorul are nevoie de `p` și `q`. Pe 3233 îl descompune într-o clipă (py9-05), dar în RSA-ul real `n` are sute de cifre: înmulțirea e ușoară, descompunerea practic imposibilă.

> **ATENȚIE — GREȘELI FRECVENTE**
>
> Codul fiecărei litere trebuie să fie **mai mic decât** `n` (aici 65–90 < 3233), altfel decriptarea dă alt rest. Criptarea literă cu literă e doar un exercițiu: aceeași literă dă mereu același număr.

7. Aplicație: culorile unui tablou, codificate prin produs de prime

Un muzeu vrea să memoreze **ce culori** apar într-un tablou printr-un singur număr. Dă fiecărei culori un prim diferit (primele prime din ciur), iar tabloul primește **produsul** culorilor lui. Decodificarea se face prin împărțiri: culoarea cu primul `p` apare exact când `p` divide codul.

```python
culori = ["roșu", "galben", "albastru", "verde", "alb", "negru"]
e = ciur(20)
prime = [k for k in range(len(e)) if e[k]][:len(culori)]    # [2, 3, 5, 7, 11, 13]

def codifica(lista):
    c = 1
    for nume in lista:
        c *= prime[culori.index(nume)]
    return c

def decodifica(c):
    rez = []
    for k in range(len(culori)):
        if c % prime[k] == 0:
            rez.append(culori[k])
            c //= prime[k]
    return rez

t1 = codifica(["roșu", "albastru", "alb"])
t2 = codifica(["galben", "albastru", "alb", "negru"])
print(t1, decodifica(t1))     # 110 ['roșu', 'albastru', 'alb']
print(t2, decodifica(t2))     # 2145 ['galben', 'albastru', 'alb', 'negru']
```

De ce merge? Descompunerea în factori primi este **unică**, deci două mulțimi diferite de culori dau mereu produse diferite. Bonus: `gcd(110, 2145) = 55 = 5 · 11` dă culorile **comune** (albastru, alb), iar cel mai mic multiplu comun `4290` dă **toate** culorile celor două tablouri.

8. Verifică-te

1. În ciurul pentru `n = 50`, ce număr se taie primul la etapa `i = 7`? Se mai execută etapa `i = 8`?
2. Ce valoare are `e[1]` după ciur și de ce trebuie pusă explicit?
3. Câți pași face bucla din `putere(a, b)` pentru `b = 64`? Dar pentru `b = 63`?
4. Scrie 2^{20} ca produs de pătrate succesive, așa cum îl calculează `putere`.
5. De ce `pow(a, b, m)` este de preferat lui `a ** b % m` când `b` este foarte mare?
6. În RSA-ul din secțiunea 6, cineva primește numărul `2680`. Ce calcul face ca să afle litera, dacă știe cheia privată?
7. Un tablou are codul `78`. Ce culori conține?

Răspunsuri

1. `49 = 7 · 7`. Nu: `8 · 8 = 64 > 50`, bucla se oprește.
2. `False`. Inițializarea pune `True` peste tot, iar 1 nu are niciun divizor prim care să-l taie; la fel 0.
3. 7 pași pentru $64 = 1000000_2$ (7 biți) și 6 pași pentru $63 = 111111_2$ (6 biți). Deși 63 < 64, la 63 se fac mai multe înmulțiri (12 față de 8), pentru că fiecare bit este 1.
4. $20 = 10100_2$, deci $2^{20} = 2^{16} \cdot 2^4$; `a` trece prin $2, 2^2, 2^4, 2^8, 2^{16}$.
5. Pentru că `pow` ia restul după fiecare înmulțire și face doar ~$\log_2 b$ pași, pe când `a ** b` construiește întâi un număr cu milioane de cifre.
6. `chr(pow(2680, 2753, 3233))`, care dă `S`.
7. `78 = 2 · 3 · 13`: roșu, galben, negru.

Probleme de lucru în clasă (pbinfo)

Le rezolvăm împreună în ședință, în ordinea de mai jos.

- [ ] **#303** **eratostene** — medie
  https://www.pbinfo.ro/probleme/303/eratostene
- [ ] **#3977** **expon** — ușoară
  https://www.pbinfo.ro/probleme/3977/expon

Temă pentru acasă (pbinfo)

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

- [ ] **#302** **cautaprim** — medie
  https://www.pbinfo.ro/probleme/302/cautaprim
- [ ] **#510** **sortprime** — ușoară
  https://www.pbinfo.ro/probleme/510/sortprime