test-e.ro

CAIET ELEV

Lecția 27

Eficiența algoritmilor – analiză comparativă. Notația O completă. Alegerea metodei potrivite

De la intuiția „care algoritm e mai rapid?” la instrumentul precis: notația O.

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, completezi, asociezi și scrii singur soluțiile, apoi le verifici pe platformă. Ce ai voie să folosești: secvențial, if/elif/else, while, for cu range, %, //, liste (literal, indexare, len, append), for x in lista, ord()/chr(), def/return, min/max/sum, import math, slicing, măsurarea timpului. Fără sorted() și .sort() (ar goli de sens lecțiile de sortare), fără open/with/class. Atribuirea simultană e permisă de la L20, dar pentru continuitate cu L24–L26 interschimbarea rămâne scrisă cu variabila auxiliară aux. Cifrele se parcurg doar cu while, cu % și //. Paritatea cu n % 2 != 0. Fără diacritice în codul Python. Convenția de citire a listelor rămâne cea de la L15–L26: n pe primul rând, apoi câte o valoare pe rând.

CUPRINS

1 Ce știm deja despre pași, timp și sortare .....

2 Scara ordinelor de complexitate .....

3 Reguli de calcul: cum citim ordinul direct din cod

4 De unde vin O(log n) și O(n log n) .....

5 Cele trei sortări, comparate riguros .....

6 Timp și memorie: cum alegem metoda potrivită

7 Cod ghidat · cinci probleme .....

SECȚIUNEA 1 · CE ȘTIM DEJA DESPRE PAȘI, TIMP ȘI SORTARE

1 Ce știm deja despre pași, timp și *sortare*

Înainte de a măsura riguros eficiența, ne reamintim uneltele: numărarea pașilor unei bucle, împărțirile repetate la 2 și cele trei metode de sortare pe care le-am scris deja. Nimic nou aici — doar terenul pe care vom construi.

La lecția 18 am pus prima dată întrebarea „care algoritm e mai rapid?” și am răspuns în două feluri: cu ceasul (funcția `time()`) și cu creionul (numărând pașii). Am văzut atunci că verificarea primalității cu `while d < n` face aproape n pași, în timp ce varianta cu `while d * d <= n` face doar în jur de rădăcina lui n — pentru n = 10 000, 9998 de pași față de 99.

Apoi am învățat să lucrăm cu liste și am scris trei metode de sortare. **Selecția minimului** (L24) caută minimul din porțiunea nesortată și îl aduce la locul lui, cu o interschimbare făcută prin variabila auxiliară `aux`. **Sortarea cu listă de frecvențe** (L25) numără de câte ori apare fiecare valoare și apoi scrie valorile în ordine, dar funcționează doar pentru numere întregi dintr-un interval mic. **Metoda bulelor** (L26) compară perechi de vecini și îi interschimbă, iar varianta optimizată se oprește mai devreme dacă o trecere completă nu a produs nicio interschimbare.

Mai avem nevoie de o unealtă veche: împărțirea repetată la 2. Am folosit-o la conversia în baza 2 (L12). Un număr împărțit din nou și din nou la 2 ajunge surprinzător de repede la 1 — și tocmai de aceea revine în lecția de azi.

A

EXERCIȚIUL A · GRILĂ

Ce ne amintim din lecțiile 18 și 24–26

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Sortarea prin selecția minimului aplicată unei liste cu 6 elemente face exact câte comparări?

   - `6, câte una pentru fiecare element al listei de sortat`
   - `36, adică 6 × 6, câte o comparare pentru fiecare pereche posibilă`
   - `30, adică 6 × 5, toate perechile ordonate de elemente distincte`
   - `15, adică 6 × 5 ÷ 2`

2. Sortarea cu listă de frecvențe poate fi folosită numai dacă:

   - `valorile sunt numere reale, distribuite uniform pe un interval mare`
   - `lista este deja aproape ordonată crescător, cu foarte puține inversiuni`
   - `valorile sunt numere întregi dintr-un interval mic, cunoscut dinainte`
   - `lista are un număr par de elemente, ca să poată fi împărțită în două`

3. Optimizarea metodei bulelor (oprirea timpurie) se sprijină pe:

- `reducerea cu o unitate a numărului de treceri rămase, la fiecare pas al sortării`
- `un martor care reține dacă în trecerea curentă s-a făcut vreo interschimbare`
- `compararea primului element cu ultimul element înainte de fiecare trecere nouă`
- `numărarea elementelor deja așezate definitiv la capătul din dreapta al listei`

4. La lecția 18 am măsurat timpul de execuție al unui program folosind:

- `funcția len(), care numără câți pași a făcut bucla while până la oprire`
- `funcția input(), care citește de la tastatură durata măsurată cu cronometrul`
- `funcția time() din modulul time, apelată înainte și după algoritm`
- `funcția round(), care rotunjește numărul de operații la secunde întregi`

B

EXERCIȚIUL B · DRY-RUN

*Câți pași face bucla cu d * d <= n?*

DRY-RUN

Urmărește bucla pentru n = 30. Coloana d este dată; tu completezi coloana d * d și coloana pasi (câte iterații s-au făcut până acolo). Primul rând e completat ca exemplu. Ultimul rând arată starea în care condiția devine falsă și bucla se oprește. Toate valorile de completat sunt numerice.

```
n = 30
d = 1
pasi = 0
while d * d <= n:
    pasi = pasi + 1
    d = d + 1
print(pasi)
```

d	d * d	pasi după iterație
1	1	1
2		
3		
4		
5		
6		

HINT *La fiecare iterație calculează întâi d * d, apoi crește pasi cu 1. Pe ultimul rând bucla nu se mai execută, deci pasi rămâne neschimbat.*

SPRIJIN · SCRIE CODUL

Cât de departe merge căutarea?

SCRIERE COD

Se citește un număr natural n, apoi n numere întregi, câte unul pe rând, care formează o listă. Afișează numărul de pași făcuți de o parcurgere care se oprește imediat ce a găsit primul număr negativ (pasul în care l-a găsit se numără și el); dacă lista nu conține niciun număr negativ, numărul de pași este n. Ex.: pentru n = 5 și lista 3, 7, −2, 9, 4 se afișează 3: parcurgerea verifică 3, apoi 7, apoi −2, unde se oprește.

HINT *Numărul de pași nu este mereu n — depinde de date. Aceasta este diferența dintre cazul cel mai favorabil și cel mai defavorabil, de care ne vom folosi în toată lecția.*

SOLUȚIA TA · PYTHON

SPRIJIN · SCRIE CODUL

De câte ori se înjumătățește?

SCRIERE COD

Se citește un număr natural n (n ≥ 1). Afișează de câte ori trebuie împărțit n la 2, prin împărțire întreagă, până când ajunge la 1. Ex.: pentru n = 8: 8 → 4 → 2 → 1, adică 3 împărțiri, deci se afișează 3.

HINT **Atenție.** *Împărțirea este întreagă: din 25 se obține 12, nu 12,5. Pentru n = 1 nu se face nicio împărțire, deci se afișează 0.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 2 · SCARA ORDINELOR DE COMPLEXITATE

2 Scara ordinelor de *complexitate*

Notația O nu spune câte secunde durează un algoritm, ci cum crește munca lui atunci când cresc datele. Cinci trepte — O(1), O(log n), O(n), O(n log n) și O(n²) — acoperă aproape tot ce vei întâlni.

Când comparăm doi algoritmi, secundele ne mint. Același program rulează de zece ori mai repede pe un calculator nou și de zece ori mai încet dacă e scris într-un limbaj mai lent. Ce nu se schimbă niciodată este **forma creșterii**: dacă dublăm datele, cât se înmulțește munca?

Notăm cu **n** dimensiunea datelor (numărul de elemente ale listei, valoarea numărului prelucrat) și scriem **O(f(n))** pentru a spune că numărul de operații crește ca `f(n)`. Litera O se citește „ordinul lui” și vine de la cuvântul german *Ordnung*.

Iată scara completă, cu numere reale, calculate:

ORDIN	CE SE ÎNTÂMPLĂ CÂND N SE DUBLEAZĂ	N = 10	N = 100	N = 1000	N = 1 000 000
O(1)	nu se întâmplă nimic, munca e aceeași	1	1	1	1
O(log n)	munca crește cu o unitate	~3	~6	~9	~19
O(n)	munca se dublează	10	100	1000	1 000 000
O(n log n)	munca se dublează și un pic	~30	~600	~9000	~19 000 000
O(n²)	munca se face de 4 ori mai mare	100	10 000	1 000 000	1 000 000 000 000

Uită-te la ultima coloană. Pentru un milion de elemente, un algoritm O(n) termină înainte să clipești, unul O(n log n) durează câteva secunde, iar unul O(n²) are de făcut un milion de milioane de operații — adică peste o zi de calcul neîntrerupt. **Diferența dintre ordine nu e o nuanță, e diferența dintre „merge” și „nu merge”.**

> **ORDINUL DE COMPLEXITATE**
>
> Ordinul O(f(n)) descrie **ritmul de creștere** al numărului de operații, nu numărul lor exact și nici timpul în secunde. Doi algoritmi cu același ordin pot diferi de câteva ori între ei; doi algoritmi cu ordine diferite ajung să difere de mii de ori, dacă n e mare.

> **ATENȚIE**

Un ordin mai mic nu înseamnă automat „mai rapid pentru orice n”. Pentru n = 5, un algoritm $O(n^2)$ cu operații foarte simple poate bate un algoritm $O(n)$ cu operații complicate. Ordinul spune cine câștigă **atunci când n crește**, iar în practică n chiar crește: liste de 10 elemente astăzi, de 100 000 la anul.

UNDE SE FOLOSEȘTE ÎN VIAȚA REALĂ

Când aplicația de mesaje caută un contact printre 5000 de nume și îl găsește instantaneu, în spate e un algoritm de ordin mic; dacă ar fi fost $O(n^2)$, ai fi așteptat vizibil. Motoarele de căutare indexează miliarde de pagini tocmai pentru că interogarea rămâne aproape $O(1)$ — căutarea prin toate paginile la fiecare întrebare ar fi imposibilă. Iar când un joc „se blochează” la încărcare, aproape mereu cineva a lăsat undeva o buclă imbricată acolo unde datele au crescut peste așteptări.

A

EXERCIȚIUL A · GRILĂ

Ce spune și ce nu spune notația O

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Ce descrie notația $O(n^2)$?

   - Faptul că algoritmul face exact n × n operații, nici una în plus, nici una în minus
   - Faptul că algoritmul are nevoie de n^2 celule de memorie pentru datele de lucru
   - Faptul că numărul de operații crește aproximativ cu pătratul dimensiunii datelor
   - Faptul că timpul de execuție este de n^2 secunde, pe orice fel de calculator

2. Pentru n = 1 000 000, care dintre ordinele de mai jos cere cele mai puține operații?

   - O(n log n), adică aproximativ 19 milioane de operații în total
   - O(n), adică un milion de operații, câte una pentru fiecare element
   - O(log n), aproximativ 19 operații
   - $O(n^2)$, adică un milion de milioane de operații, deci zile de calcul

3. Un algoritm de ordin O(1):

   - este întotdeauna mai rapid în secunde decât orice algoritm de ordin O(n)
   - face exact o singură operație, oricare ar fi datele de intrare primite
   - se poate scrie doar fără bucle, folosind numai instrucțiuni de atribuire
   - face același număr de operații, indiferent cât de mari sunt datele

B

EXERCIȚIUL B · COMPLETARE

Trei ordine, trei comportamente

COMPLETARE

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

`se înjumătățește` `imbricate` `consecutive` `nu depinde` `proporțional`

Notația O(1) înseamnă că numărul de operații _____ de dimensiunea datelor, ordinul O(n) înseamnă că numărul de operații crește _____ cu n, iar ordinul O(n²) apare cel mai des la două bucle _____.

C

EXERCIȚIUL C · SCRIE CODUL

Cât de mare poate fi n?

SCRIERE COD

Se citește un număr natural P — numărul maxim de operații pe care suntem dispuși să le așteptăm. Afișează cel mai mare număr natural n pentru care un algoritm de ordin O(n²), care face exact n × n operații, rămâne sub acest prag, adică n × n ≤ P. Ex.: pentru P = 100 se afișează 10, pentru că 10 × 10 = 100 încape exact în prag, iar 11 × 11 = 121 îl depășește.

HINT *Rezultatul este partea întreagă a rădăcinii pătrate a lui P, dar aici o afli căutând pas cu pas, fără math.sqrt, ca să vezi cum crește n.*

HINT **Atenție.** *Pentru P = 0 niciun n ≥ 1 nu încape, deci se afișează 0.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 3 · REGULI DE CALCUL: CUM CITIM ORDINUL DIRECT DIN COD

3 Reguli de calcul: cum citim ordinul direct din *cod*

Nu e nevoie să numeri operațiile una câte una. Patru reguli simple — atribuiri, bucle una după alta, bucle imbricate, termen dominant — îți dau ordinul dintr-o privire.

O instrucțiune simplă (o atribuire, o comparare, un `print`) costă un pas: **O(1)**. Ordinul apare din bucle, iar buclele se combină în două feluri.

Bucle una după alta — se adună. Fiecare buclă își face pașii ei, apoi începe următoarea:

```
for i in range(n):
    c = c + 1
for i in range(n):
    c = c + 1
```

Aici corpul se execută n + n = 2n ori. Ordinul este **O(n)**: constanta 2 se aruncă, pentru că nu schimbă ritmul de creștere (dublezi n și tot se dublează și munca).

Bucle imbricate — se înmulțesc. Bucla interioară se reia complet la fiecare pas al celei exterioare:

```
for i in range(n):
    for j in range(n):
        c = c + 1
```

Corpul se execută n × n ori, deci ordinul este **O(n²)**. Dublezi n și munca se face de patru ori mai mare.

Un caz special, pe care îl cunoaștem deja de la sortări: bucla interioară nu pornește de la 0, ci de după poziția celei exterioare.

```
for i in range(n):
    for j in range(i + 1, n):
        c = c + 1
```

Pentru n = 5 corpul se execută 4 + 3 + 2 + 1 + 0 = 10 ori, adică n(n−1)/2. Jumătate din n², dar **tot O(n²)** — constanta 1/2 se ignoră.

> **REGULA BUCLELOR**

Bucle **una după alta** → numerele de pași se **adună**. Bucle **una în alta** → numerele de pași se **înmulțesc**.

REGULA TERMENULUI DOMINANT

Din suma obținută se păstrează doar termenul care crește cel mai repede, iar coeficienții se aruncă. Astfel, $5n^2 + 300n + 1000$ devine **$O(n^2)$**. Pare nedrept față de 300n, dar la n = 1000 termenul pătratic valorează 5 000 000, iar ceilalți doi împreună doar 301 000 — și distanța crește la fiecare pas.

ATENȚIE

Regula se aplică termenilor, nu cifrelor. `O(5n²)`, `O(n² + n)` și `O(1000)` nu se scriu niciodată: forma corectă este `O(n²)`, respectiv `O(1)`.

UNDE SE FOLOSEȘTE ÎN VIAȚA REALĂ

Programatorii citesc ordinul unui cod exact așa, din ochi, în timpul unei revizuiri de cod: o buclă în altă buclă peste lista de utilizatori e un semnal de alarmă imediat. La interviurile pentru joburi de programator, „ce complexitate are soluția ta?” e printre primele întrebări puse. Iar când o aplicație merge perfect în teste cu 100 de înregistrări și moare la 100 000, cauza e aproape mereu o pereche de bucle imbricate ascunsă undeva.

A

EXERCIȚIUL A · GRILĂ

Aplicăm regulile

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Un algoritm face $5n^2 + 300n + 1000$ operații. Ordinul lui este:

   - `O(1000), pentru că 1000 este cea mai mare valoare care apare în formulă`
   - `O(n²), pentru că termenul cu n² crește cel mai repede dintre toate`
   - `O(n), pentru că termenul 300n are coeficientul cel mai mare dintre toți`
   - `O(5n²), pentru că trebuie păstrat și coeficientul termenului dominant`

2. Două bucle scrise una după alta, fiecare cu n pași, dau împreună:

   - `n pași, deci ordinul O(n), pentru că se execută doar cea mai lungă dintre ele`
   - `2n pași, deci ordinul O(2n), pentru că se păstrează și constanta 2`
   - `2n pași, deci ordinul O(n)`
   - `n × n pași, deci ordinul O(n²), pentru că se execută amândouă buclele`

3. Bucla `for j in range(i + 1, n)`, aflată în interiorul buclei `for i in range(n)`, execută corpul în total:

n × n ori, deci de exact două ori mai mult decât o singură parcurgere

n − 1 ori, pentru că j pornește de fiecare dată cu o poziție mai departe

n(n + 1)/2 ori, pentru că j pornește chiar de la i și ajunge până la n

n(n − 1)/2 ori, deci ordinul rămâne O(n²)

B

EXERCIȚIUL B · DRY-RUN

Câți pași face bucla triunghiulară?

DRY-RUN

Urmărește bucla dublă pentru n = 5. Coloana i este dată; tu completezi câte valori ia j la fiecare trecere și valoarea lui c la finalul trecerii. Primul rând e completat ca exemplu. Ultimul rând este cel în care bucla interioară nu se mai execută deloc. Toate valorile de completat sunt numerice.

```
1  n = 5
2  c = 0
3  for i in range(n):
4      for j in range(i + 1, n):
5          c = c + 1
6  print(c)
```

i	câte valori ia j	c la finalul trecerii
0	4	4
1		
2		
3		
4		

HINT *La fiecare trecere j ia valorile de la i + 1 până la n − 1, deci numărul lor scade cu o unitate de fiecare dată: 4, 3, 2, 1, 0.*

C

EXERCIȚIUL C · SCRIE CODUL

Adunare sau înmulțire?

SCRIERE COD

Se citește un număr natural n. Afișează pe primul rând de câte ori se execută corpul a două bucle scrise una după alta, fiecare parcurgând n valori, iar pe al doilea rând de câte ori se execută corpul a două bucle imbricate, fiecare parcurgând n valori. Ambele numere se obțin numărând efectiv, cu contoare, în bucle `for` reale — nu prin formulă. Ex.: pentru n = 3 se afișează 6 pe primul rând (3 + 3) și 9 pe al doilea (3 × 3).

HINT *Pentru n mic diferența pare mică, dar la n = 1000 devine 2000 față de 1 000 000.*

HINT **Atenție.** *Pentru n = 0 nicio buclă nu se execută, deci ambele numere sunt 0.*

SOLUȚIA TA · PYTHON

★

PERFORMANȚĂ · SCRIE CODUL

Ghicește ordinul din măsurători

SCRIERE COD

Se citesc două numere naturale p1 și p2 (p1 ≥ 1): p1 este numărul de operații măsurat pentru n elemente, iar p2 numărul de operații măsurat pentru 2n elemente, pentru același algoritm. Afișează O1 dacă algoritmul este de ordin O(1), On dacă este de ordin O(n) și On2 dacă este de ordin O(n²).

Regula de decizie: dacă p2 este mai mic decât 1,5 × p1, ordinul este O(1); altfel, dacă p2 este mai mic decât 3 × p1, ordinul este O(n); altfel ordinul este O(n²). Ex.: pentru p1 = 100 și p2 = 400 raportul este 4, deci dublarea datelor a împătrit munca: se afișează On2.

HINT *Aici mergi invers față de tot restul lecției: nu pornești de la cod ca să afli ordinul, ci de la două măsurători ca să deduci ordinul. Așa lucrează cineva care primește un program străin, îl rulează de două ori și trage concluzia.*

HINT **Atenție.** *Comparațiile cu 1,5 se scriu fără numere zecimale: în loc de p2 / p1 < 1.5 folosește 2 * p2 < 3 * p1, ca să rămâi în numere întregi.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 4 · DE UNDE VIN O(LOG N) ȘI O(N LOG N)

4 De unde vin O(log n) și O(n log *n*)

Două ordine noi, dar cu o unealtă veche: împărțirea repetată la 2. Un milion de elemente se reduc la unul singur în doar 19 pași — iar de aici pornesc cei mai rapizi algoritmi din informatică.

Am făcut asta deja, la conversia în baza 2: împărțim un număr la 2, apoi rezultatul la 2, și tot așa. Ce nu am observat atunci este **cât de repede** se termină. Numărul de înjumătățiri până la 1 crește îngrozitor de încet:

N	8	100	1000	1 000 000
înjumătățiri până la 1	3	6	9	19

De la 1000 la un milion, n se face de o mie de ori mai mare, iar numărul de pași crește de la 9 la 19. **Asta este O(log n).** Scheletul, în Python:

```
v = n
c = 0
while v > 1:
    v = v // 2
    c = c + 1
```

ORDINUL LOGARITMIC O(LOG N)

Apare ori de câte ori un algoritm **aruncă jumătate din date la fiecare pas**. Numărul de pași este de aproximativ $\log_2 n$, adică puterea la care trebuie ridicat 2 ca să obții n. Regula practică: fiecare dublare a lui n adaugă **un singur** pas.

Al doilea ordin nou apare când repetăm o operație logaritmică pentru **fiecare** dintre cele n elemente ale unei liste:

```
for i in range(n):
    v = a[i]
    while v > 1:
        v = v // 2
```

Munca totală este n × (numărul de înjumătățiri), adică **O(n log n)**. Pentru o listă cu 1000 de valori, în jur de 9000 de operații — de peste 100 de ori mai puțin decât cele 1 000 000 ale unui algoritm pătratic pe aceleași date.

ORDINUL O(N LOG N)

Este ordinul celor mai rapide metode generale de sortare, pe care le vei studia în clasele următoare. Reține de pe acum reperul: pentru 1000 de elemente, o sortare $O(n^2)$ face aproape 500 000 de comparări, iar una $O(n \log n)$ în jur de 9000.

ATENȚIE

Împărțirea este întreagă, deci din 25 se obține 12, nu 12,5. Numărul de înjumătățiri nu este niciodată un număr zecimal — de aceea scriem „aproximativ $\log_2 n$”, nu „exact”.

UNDE SE FOLOSEȘTE ÎN VIAȚA REALĂ

> Când cauți un cuvânt în dicționarul de hârtie deschizându-l la mijloc și eliminând jumătate, faci un algoritm O(log n): 100 000 de cuvinte se rezolvă în 17 deschideri. La fel funcționează căutarea unei melodii într-o listă ordonată sau găsirea unei tranzacții într-un registru bancar sortat după dată. Iar butonul „Sortează după preț” dintr-un magazin online, apăsat peste un milion de produse, se rezolvă în câteva zecimi de secundă tocmai pentru că sortarea folosită acolo e O(n log n), nu O(n²).

A

EXERCIȚIUL A · GRILĂ

Înjumătățiri și ordine

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. De câte ori poate fi înjumătățit 1000 (împărțire întreagă) până ajunge la 1?

   - De 500 de ori, pentru că la fiecare pas se scade jumătate din valoarea inițială
   - De 3 ori, pentru că 1000 se împarte exact la 8, iar 8 este 2 la puterea 3
   - De 9 ori, pentru că 2 la puterea 9 este 512, iar 2 la puterea 10 trece de 1000
   - De 100 de ori, pentru că 1000 împărțit la 10 înseamnă 100 de pași de calcul

2. Un algoritm de ordin O(n log n) pe o listă cu 1000 de elemente face aproximativ:

   - 9 operații, adică numărul de înjumătățiri ale lui 1000 până la 1
   - 1 000 000 de operații, adică 1000 × 1000, ca la ordinul pătratic
   - 1000 de operații, adică o singură trecere prin toate elementele listei
   - 9000 de operații, adică 1000 × 9

3. De unde apare ordinul O(log n)?

   - Din algoritmii care compară fiecare element cu toate celelalte elemente
   - Din algoritmii care parcurg lista o singură dată, element cu element
   - Din algoritmii care nu depind deloc de dimensiunea datelor de intrare
   - Din algoritmii care înjumătățesc la fiecare pas cantitatea de date rămasă

B

EXERCIȚIUL B · DRY-RUN

Înjumătățirea lui 100

DRY-RUN

Urmărește bucla pentru v = 100. Coloana „v la începutul pasului” este dată; tu completezi valoarea lui v după împărțire și contorul c. Primul rând e completat ca exemplu. Ultimul rând arată starea în care condiția v > 1 devine falsă: acolo nu se mai face nicio împărțire, deci se scrie —. Valorile de completat sunt numerice.

```python
v = 100
c = 0
while v > 1:
    v = v // 2
    c = c + 1
print(c)
```

v la începutul pasului	v după împărțire	c
100	50	1
50		
25		
12		
6		
3		
1	—	6

HINT *La fiecare pas v devine v // 2 (împărțire întreagă: din 25 se obține 12), iar c crește cu o unitate. Pe ultimul rând bucla nu se mai execută.*

C

EXERCIȚIUL C · SCRIE CODUL

Câte înjumătățiri în total?

SCRIERE COD

Se citește un număr natural n, apoi n numere naturale nenule, câte unul pe rând, care formează o listă. Pentru fiecare număr din listă se numără câte împărțiri întregi la 2 sunt necesare până când ajunge la 1. Afișează suma tuturor acestor numărări. Ex.: pentru n = 3 și lista 8, 8, 8 se afișează 9, pentru că fiecare 8 cere 3 înjumătățiri.

HINT *Numărul afișat este exact munca unui algoritm de ordin O(n log n): pentru fiecare dintre cele n elemente se face o serie de înjumătățiri.*

HINT **Atenție.** *Numărul 1 nu are nevoie de nicio înjumătățire, deci o listă formată numai din valori 1 dă rezultatul 0. Parcurge lista cu* `for` *și folosește o singură buclă* `while`*, cea a înjumătățirilor.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 5 · CELE TREI SORTĂRI, COMPARATE RIGUROS

5 Cele trei sortări, comparate *riguros*

Selecția minimului, metoda bulelor și lista de frecvențe: acum le putem compara cu cifre exacte, nu din impresie. Două sunt O(n²), una este O(n + k) — și tocmai de aceea nu se poate folosi oricând.

Avem trei metode scrise cu mâna noastră. Le punem față în față pe același criteriu: **câte operații esențiale fac pentru o listă cu n elemente.**

Selecția minimului (L24) caută minimul în porțiunea nesortată, apoi îl aduce la locul lui. Prima căutare compară n−1 elemente, a doua n−2, și tot așa. Suma este n(n−1)/2 comparări și cel mult n−1 interschimbări. Ordin: **O(n²)**.

Metoda bulelor neoptimizată (L26) face n−1 treceri, iar la trecerea k compară n−1−k perechi de vecini. Suma este tot n(n−1)/2 comparări, dar interschimbările pot fi la fel de multe ca și comparările, dacă lista e ordonată invers. Ordin: **O(n²)**. Varianta cu oprire timpurie face o singură trecere pe o listă deja ordonată — un caz favorabil, care însă nu schimbă ordinul în cazul cel mai defavorabil.

Lista de frecvențe (L25) parcurge o dată lista ca să numere aparițiile (n pași) și o dată intervalul de valori ca să le scrie în ordine (k pași, unde k este numărul de valori distincte posibile). Total: n + k. Ordin: **O(n + k)**.

Numerele exacte, calculate, pentru un interval de 10 valori (de exemplu note de la 1 la 10):

N	SELECȚIA MINIMULUI / BULELE: N(N−1)/2	LISTA DE FRECVENȚE: N + K	DE CÂTE ORI MAI PUȚIN
10	45	20	~2 ori
100	4 950	110	~45 de ori
1 000	499 500	1 010	~495 de ori
10 000	49 995 000	10 010	~4 995 de ori

CONDIȚIA DE APLICABILITATE

Lista de frecvențe câștigă strivitor, dar **numai** dacă valorile sunt numere **întregi** dintr-un interval **mic și cunoscut dinainte**. Pentru note (1–10), vârste (0–120) sau zile (1–31) e ideală. Pentru prețuri cu zecimale, pentru coduri numerice de 9 cifre sau pentru valori dintr-un interval de un miliard, k devine uriaș și metoda devine mai scumpă decât cea pătratică.

ATENȚIE

„Același ordin” nu înseamnă „la fel de bune”. Selecția minimului face cel mult n−1 interschimbări, iar bulele neoptimizate pot face până la n(n−1)/2 — dacă mutarea unui element e costisitoare, diferența se simte, chiar dacă ordinul e același $O(n^2)$ la amândouă.

UNDE SE FOLOSEȘTE ÎN VIAȚA REALĂ

Un catalog electronic care ordonează 1000 de note întregi folosește lista de frecvențe și termină practic instantaneu; aceeași sarcină cu selecția minimului ar cere de aproape 500 de ori mai multe operații. La un concurs de matematică cu 20 000 de participanți și punctaje întregi de la 0 la 100, lista de frecvențe rezolvă clasamentul cu ~20 100 de operații, față de ~200 de milioane. În schimb, pentru ordonarea a 20 000 de prețuri cu bani și zecimale, lista de frecvențe nu se poate folosi deloc — și atunci se caută o metodă O(n log n).

A

EXERCIȚIUL A · GRILĂ

Comparăm cu cifre, nu cu impresii

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1.

Pentru o listă cu 1000 de note întregi din intervalul 1–10, lista de frecvențe face 1010 operații, iar selecția minimului 499 500. Raportul dintre ele este de aproximativ:

`100 de ori mai puține operații, adică n împărțit la k, adică 1000 ÷ 10`

`de două ori mai puține operații, pentru că se face o singură parcurgere`

`5 ori mai puține operații, pentru că intervalul are doar 10 valori posibile`

495 de ori mai puține operații pentru lista de frecvențe

2. De ce metoda bulelor și selecția minimului au același ordin $O(n^2)$, deși se comportă diferit?

Pentru că amândouă se opresc mai devreme dacă lista este deja ordonată crescător

Pentru că amândouă folosesc aceeași variabilă auxiliară la interschimbare

Pentru că amândouă parcurg lista de exact n ori, de la primul la ultimul element

Pentru că amândouă fac aproximativ n(n − 1)/2 comparări, deci cresc la fel

3. În ce situație NU putem folosi sortarea cu listă de frecvențe?

Când lista are mai mult de o mie de elemente care trebuie ordonate

Când valorile se repetă de mai multe ori în interiorul aceleiași liste

Când lista trebuie ordonată descrescător, nu crescător, după sortare

Când valorile sunt reale sau se întind pe un interval foarte mare

B

EXERCIȚIUL B · ASOCIERE

Fiecare metodă cu ordinul ei

ASOCIERE

Potrivește fiecare algoritm din stânga cu descrierea corectă a costului său din dreapta. Opțiunile din dreapta sunt amestecate. Scrie în căsuța din fața fiecărui element din stânga litera variantei potrivite. Fiecare literă se folosește o singură dată.

☐ Selecția minimului

☐ Metoda bulelor, varianta neoptimizată

☐ Sortarea cu listă de frecvențe (valori întregi, interval mic)

☐ Parcurgerea unei liste pentru a afla valoarea maximă

A O(n + k) — o parcurgere a listei și una a intervalului de valori

B O(n) — o singură parcurgere, cu o comparare la fiecare element

C $O(n^2)$ — n(n−1)/2 comparări și, în cazul cel mai rău, tot atâtea interschimbări

D $O(n^2)$ — n(n−1)/2 comparări și cel mult n−1 interschimbări

C

EXERCIȚIUL C · SCRIE CODUL

Care metodă e mai ieftină?

SCRIERE COD

Se citesc două numere naturale: n, numărul de elemente ale listei, și k, numărul de valori distincte posibile. Afișează pe primul rând numărul de operații ale selecției minimului, calculat ca n(n−1)/2, pe al doilea rând numărul de operații ale listei de frecvențe, calculat ca n + k, iar pe al treilea rând SELECTIE, FRECVENTE sau EGAL, după care metodă face mai puține operații. Ex.: pentru n = 10 și k = 5 se afișează 45, apoi 15, apoi FRECVENTE.

HINT *Pentru liste mici, lista de frecvențe poate ieși mai scumpă: la n = 5 și k = 100 se compară 10 operații cu 105.*

HINT **Atenție.** *Pentru n = 0 și k = 0 ambele metode dau 0 operații, deci răspunsul este EGAL.*

SOLUȚIA TA · PYTHON

6 SECȚIUNEA 6 · TIMP ȘI MEMORIE: CUM ALEGEM METODA POTRIVITĂ

Timp și memorie: cum alegem metoda *potrivită*

Timpul nu e singura resursă. Un algoritm poate fi mai rapid tocmai pentru că ocupă mai multă memorie — iar alegerea corectă înseamnă să știi ce ai de dat la schimb.

Până acum am numărat doar operații. Dar orice algoritm consumă și **memorie**: celulele suplimentare de care are nevoie **peste** datele primite. Măsurăm și această resursă cu notația O și îi spunem **complexitate spațială**.

Selecția minimului și metoda bulelor lucrează **în aceeași listă**: mută elementele între ele și au nevoie doar de câteva variabile (`i`, `j`, `aux`), oricât de mare ar fi lista. Complexitate spațială: **O(1)**. Lista de frecvențe are nevoie de o listă nouă, cu câte o celulă pentru fiecare valoare posibilă: **O(k)**.

Iată schimbul, pe cazul din S5 (n = 1000, valori de la 1 la 10):

METODĂ	TIMP (OPERAȚII)	MEMORIE SUPLIMENTARĂ
selecția minimului	499 500	câteva variabile — O(1)
metoda bulelor	499 500	câteva variabile — O(1)
lista de frecvențe	1 010	10 celule — O(k)

Lista de frecvențe cumpără timp cu memorie. Când k e mic, târgul e excelent; când k e uriaș (prețuri, coduri de bare), memoria cerută devine imposibilă.

Ordinul se vede și cu ceasul, nu doar pe hârtie. Măsurăm o buclă dublă, cu funcția `time()` învățată la L18:

```
from time import time
n = 2000
t1 = time()
c = 0
for i in range(n):
    for j in range(n):
        c = c + 1
t2 = time()
print(c)
print(round(t2 - t1, 3))
```

Rulat pe același calculator pentru trei valori, a dat: n = 500 → 0,017 s; n = 1000 → 0,070 s; n = 2000 → 0,279 s. **La fiecare dublare a lui n, timpul s-a făcut de aproximativ 4 ori mai mare** — exact ce prezice O(n²). Valorile absolute diferă de la un calculator la altul; raportul, nu.

> **CRITERIILE DE ALEGERE**
>
> Întreabă, în ordine: (1) Valorile sunt **întregi** dintr-un interval **mic**? Dacă da, lista de frecvențe câștigă detașat. (2) Am **memorie** pentru cele k celule? Dacă nu, rămâi la o metodă în aceeași listă. (3) Lista e deja **aproape ordonată**? Atunci metoda bulelor cu oprire timpurie termină foarte repede. (4) Dacă niciuna nu se potrivește, alegi o metodă generală și accepți ordinul ei.

ATENȚIE

Secundele măsurate cu `time()` nu sunt un argument în sine: același program dă alt număr pe alt calculator, sau chiar pe același calculator la o a doua rulare. Argumentul e **raportul** dintre măsurători atunci când dublezi n — el arată ordinul.

UNDE SE FOLOSEȘTE ÎN VIAȚA REALĂ

Un telefon cu memorie puțină preferă algoritmi care lucrează pe loc, chiar dacă sunt mai lenți; un server cu multă memorie face invers și ține tabele mari ca să răspundă instantaneu. Aplicațiile de navigație păstrează hărți precalculate (multă memorie) ca să găsească traseul în milisecunde. Iar jocurile țin în memorie liste de frecvențe pentru scoruri, tocmai pentru că 10 000 de jucători cu punctaje întregi se ordonează într-o singură trecere.

A

EXERCIȚIUL A · GRILĂ

A doua resursă

GRILĂ

Încercuiește răspunsul corect la fiecare întrebare.

1. Complexitatea spațială a unui algoritm măsoară:

   `numărul de operații pe care le face algoritmul până la terminare`

   `memoria suplimentară cerută, peste datele de intrare`

   `numărul de linii de cod din care este alcătuit programul, adică lungimea lui`

   `spațiul ocupat pe disc de fișierul în care este salvat programul scris`

2. Sortarea prin selecția minimului are complexitate spațială:

   `O(k), pentru că are nevoie de câte o celulă pentru fiecare valoare distinctă`

   `O(1), pentru că lucrează în aceeași listă și cere doar câteva variabile`

   `O(n²), la fel ca timpul, pentru că memoria crește odată cu numărul de pași`

   `O(n), pentru că are nevoie de o a doua listă în care mută elementele`

3. Pentru 200 de elevi, cu note întregi de la 1 la 10, ce metodă alegem?

   `Oricare dintre ele, pentru că toate au același ordin de complexitate O(n²)`

   `Selecția minimului: 19 900 de comparări, dar fără memorie suplimentară`

   `Lista de frecvențe: 210 operații și doar 10 celule suplimentare`

   `Metoda bulelor neoptimizată: 19 900 de comparări și multe interschimbări`

B

EXERCIȚIUL B · COMPLETARE

Cât spațiu cere fiecare metodă

COMPLETARE

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

`constantă` `spațială` `k` `n²` `temporală` `aceeași`

Complexitatea _____ arată câtă memorie suplimentară cere un algoritm. Sortarea prin selecția minimului lucrează în _____ listă, deci cere memorie suplimentară _____, în timp ce lista de frecvențe are nevoie de aproximativ _____ celule în plus.

C

EXERCIȚIUL C · SCRIE CODUL

Alege metoda potrivită

SCRIERE COD

Se citesc, câte unul pe rând, patru numere întregi: n (numărul de elemente), vmin și vmax (cea mai mică și cea mai mare valoare posibilă) și m (numărul de celule suplimentare de memorie disponibile). Afișează pe primul rând k, numărul de valori distincte posibile, calculat ca vmax − vmin + 1. Pe al doilea rând afișează FRECVENTE dacă lista de frecvențe încape în memorie (k ≤ m) și este mai ieftină ca timp (n + k < n(n−1)/2), iar în caz contrar SELECTIE. Ex.: pentru n = 1000, vmin = 1, vmax = 10, m = 100 se afișează 10, apoi FRECVENTE: cele 10 celule încap în cele 100 disponibile, iar 1010 operații sunt mult sub 499 500.

HINT *Ambele condiții trebuie îndeplinite. Pentru n = 5, vmin = 1, vmax = 5, m = 10 memoria ajunge, dar timpul nu: 5 + 5 = 10 operații față de 10, deci nu e mai ieftină și se alege SELECTIE.*

HINT **Atenție.** *Valorile vmin și vmax pot fi și negative — pentru vmin = −5 și vmax = 5 avem k = 11, nu 10.*

SOLUȚIA TA · PYTHON

SECȚIUNEA 7 · COD GHIDAT

7 Scrie cod care *trece testele*

Cinci probleme. 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.

1

PROBLEMA 7.1 · COD GHIDAT

Euclid: scăderi sau împărțiri?

SCRIERE COD

Se citesc două numere naturale nenule a și b, câte unul pe rând. Afișează pe primul rând numărul de pași făcuți de algoritmul lui Euclid cu scăderi repetate, iar pe al doilea rând numărul de pași făcuți de algoritmul lui Euclid cu împărțiri (cu restul). Un pas înseamnă o scădere, respectiv o împărțire. Ex.: pentru a = 100 și b = 1 se afișează 99 pe primul rând (se scade 1 din 100 de nouăzeci și nouă de ori) și 1 pe al doilea (o singură împărțire).

HINT *Amândoi algoritmii dau același CMMDC — ceea ce diferă este drumul până acolo. Cazul a = 100, b = 1 arată de ce varianta cu împărțiri e preferată în practică.*

HINT **Atenție.** *Fiecare variantă trebuie pornită de la valorile citite inițial, deci păstrează a și b în variabile separate înainte de prima buclă, altfel a doua buclă pornește din valorile deja modificate. La varianta cu împărțiri numeri fiecare împărțire cu rest efectuată, inclusiv prima — nu numărul de repetări ale buclei.*

SOLUȚIA TA · PYTHON

2

PROBLEMA 7.2 · COD GHIDAT

Primalitate: O(n) față de O(√n)

SCRIERE COD

Se citește un număr natural n (n ≥ 2), scris în baza 10. Afișează pe primul rând numărul de pași făcuți de bucla while d < n, care pornește cu d = 2 și crește d cu 1 la fiecare pas, iar pe al doilea rând numărul de pași făcuți de bucla while d * d <= n, care pornește tot cu d = 2 și crește la fel. Se numără pașii în cazul cel mai defavorabil, adică fără oprire la primul divizor găsit. Ex.: pentru n = 100 se afișează 98 pe primul rând (d ia valorile de la 2 la 99) și 9 pe al doilea (d ia valorile de la 2 la 10, pentru că 10 × 10 = 100 încă respectă condiția).

HINT *Raportul dintre cele două numere crește odată cu n: la n = 100 e de aproximativ 11 ori, la n = 10 000 de aproximativ 100 de ori. Prima buclă e O(n), a doua O(√n).*

HINT **Atenție.** *A doua buclă trebuie repornită de la d = 2, altfel continuă din valoarea rămasă după prima buclă și afișează 0.*

SOLUȚIA TA · PYTHON

3

PROBLEMA 7.3 · COD GHIDAT

Termenul dominant

SCRIERE COD

Un algoritm face a + b × n + c × n × n operații, unde a, b și c sunt numere naturale. Se citesc, câte unul pe rând, numerele a, b, c și n. Afișează pe primul rând numărul total de operații pentru acest n, iar pe al doilea rând ordinul de complexitate al algoritmului, sub forma O1, On sau On2, determinat de termenul dominant: dacă c este nenul ordinul este On2, altfel, dacă b este nenul, ordinul este On, altfel O1. Ex.: pentru a = 1, b = 1, c = 1 și n = 10 totalul este 1 + 10 + 100 = 111, iar ordinul este On2.

HINT *Ordinul depinde numai de coeficienți, nu de valoarea lui n: un algoritm cu b nenul rămâne de ordin On chiar dacă îl rulăm pentru n = 0, când face doar a operații.*

HINT **Atenție.** *Ordinul nu se scrie niciodată cu coeficient: pentru a = 0, b = 5, c = 0 răspunsul este On, nu O5n.*

SOLUȚIA TA · PYTHON

4

PROBLEMA 7.4 · REPARĂ CODUL

Verifică afirmația din teorie

SCRIERE COD

În secțiunea 5 am afirmat că sortarea prin selecția minimului face exact n(n−1)/2 comparări, adică 28 pentru o listă cu 8 elemente. Programul de mai jos ar trebui să confirme afirmația numărând comparările cu bucle reale, dar pentru n = 8 afișează 36, nu 28. Găsește greșeala, corecteaz-o și afișează numărul corect de comparări pentru n citit de la tastatură.

Cod stricat:

```
n = int(input())
c = 0
for i in range(n):
    for j in range(i, n):
        c = c + 1
print(c)
```

Ex.: pentru n = 5 codul stricat afișează 15, în timp ce răspunsul corect este 10.

HINT *Valorile greșite au fost obținute rulând efectiv codul stricat: 36 pentru n = 8, 15 pentru n = 5, 5050 pentru n = 100.*

HINT **Atenție.** *Comparările se numără între perechi de poziții diferite: un element nu se compară cu el însuși. Uită-te cu atenție de unde pornește bucla interioară.*

PROGRAMUL CORECTAT · PYTHON

PROBLEMA 7.5 · PROVOCARE

De la ce n merită lista de frecvențe?

SCRIERE COD

Se citește un număr natural k — numărul de valori distincte posibile. Afișează cel mai mic număr natural n, mai mare sau egal cu 1, pentru care sortarea cu listă de frecvențe devine strict mai ieftină decât selecția minimului, adică n + k < n(n−1)/2. Ex.: pentru k = 10 se afișează 7: la n = 6 avem 16 față de 15, deci frecvențele sunt încă mai scumpe, iar la n = 7 avem 17 față de 21.

HINT *Pragul crește foarte încet cu k: de la k = 0 la k = 100, pragul urcă doar de la 4 la 16. Cu alte cuvinte, aproape întotdeauna merită — dacă valorile permit metoda.*

HINT **Atenție.** *Compararea este strictă: la egalitate metoda nu este încă mai ieftină, deci n crește mai departe.*

SOLUȚIA TA · PYTHON