433 TEXTOS DOCENTS

PROBLEMES BÀSICS D'ALGORÍSMICA

Mireia Ribera
Jordi Vitrià
Pablo Laiz
Pere Gilabert

Departament de Matemàtiques i Informàtica

UNIVERSITAT DE BARCELONA

PROBLEMES BÀSICS D'ALGORÍSMICA

Mireia Ribera
Jordi Vitrià
Pablo Laiz
Pere Gilabert

Departament de Matemàtiques i Informàtica

433 TEXTOS DOCENTS

UNIVERSITAT DE BARCELONA
Edicions

Índex

Capítol 1

QUÈ ÉS L'ALGORÍSMICA?

Un **algorisme**, en els seus termes més generals, és una seqüència d'instruccions no ambigües per resoldre un problema en un temps finit.

Amb l'ajuda dels llenguatges de programació, podem implementar algorismes computacionals que més tard podran ser interpretats per un ordinador. Com en qualsevol activitat humana, aquests algorismes computacionals poden estar ben o mal escrits, i poden ser més o menys eficients, en funció de com es dissenyin. Per tant, donat un algorisme computacional que vol resoldre un problema, és interessant que ens formulem un seguit de preguntes, concretament tres, sobre la natura de l'algorisme en relació amb el problema que vol resoldre: l'algorisme és correcte? L'algorisme és eficient? L'algorisme és elegant?

L'algorísmica, com a disciplina, ens dona eines per respondre a aquestes preguntes de manera formal i defineix de forma precisa com les hem d'entendre:

- L'algorisme és correcte si es pot demostrar formalment que resol el problema que volem resoldre.
- L'algorisme és eficient si usa la mínima quantitat de recursos computacionals (memòria, cicles de computació) per resoldre el problema.
- L'algorisme és elegant si és simple i fàcil de llegir.[1]

Definició. L'algorísmica es pot definir de forma genèrica com el conjunt de coneixements que ens ajuden a contestar les tres preguntes bàsiques respecte a un algorisme aplicat a alguna classe de problema.

Aquest llibre no pretén abordar els continguts teòrics d'aquesta matèria, sinó que és un recull de problemes que ha de servir com a suport d'un curs introductori a l'algorísmica. Així doncs, ens limitarem a l'escenari on:

- Els problemes que proposem es poden resoldre amb **variables** de diferents tipus i amb estructures de dades senzilles d'un llenguatge de programació, com ara les **llistes**, els **diccionaris** i els **conjunts**.
- L'eficiència dels algorismes es mesurarà des del punt de vista dels **cicles computacionals (o passos)** que necessiten per arribar a la solució. Només en alguns casos comentarem l'eficiència des del punt de vista de la memòria necessària per arribar a la solució. A fi i efecte d'especificar aquest aspecte, farem servir la notació $O()$, anomenada ***gran O***.

Amb relació a l'**elegància**, és un concepte més difícil de definir i cal haver llegit i entès molts algorismes per arribar a copsar-ne tots els aspectes. Aquí ens limitarem a donar un

1 Donald Knuth, un dels pares de l'algorísmica, assumeix que els algorismes estan destinats a ser llegits per humans i només esporàdicament a ser executats per ordinadors.

seguit de consells per ajudar a escriure algorismes més elegants. Direm que un algorisme és elegant si se segueixen les pautes següents:

- El codi està ben **estructurat** pel que fa a blocs i funcions.
- El codi és **minimal**: no hi sobra ni hi falta res.
- Els **noms** de les variables i funcions són informatius i ajuden a entendre l'algorisme.
- El codi fa servir **el tipus de variables i les col·leccions més adients** al problema.
- És fàcil **comprovar la correcció** del codi de forma manual, fent un seguiment mental del flux només amb l'ajut de llapis i paper.
- El codi implementa **solucions algorísmiques genèriques** i les adapta al problema que resolem, en lloc de fer servir funcions molt específiques que dificulten la comprensió i la generalització.

Els algorismes es poden escriure en pseudocodi o directament amb un llenguatge de programació. En el segon cas, el gran avantatge és que els podem executar i provar; l'inconvenient és que hem d'aprendre el llenguatge. Aquest manual no assumeix pràcticament cap coneixement de programació. El codi que s'hi proporciona està escrit en Python, un dels llenguatges que està més a prop del pseudocodi. Tots els programes d'exemple que s'hi inclouen estan degudament comentats i explicats per entendre'n el funcionament i l'execució.

```
def hola():
    """
    Aquesta funció imprimeix la frase
    'Hola, Món!' per pantalla.
    """
    print("Hola, Món!")
```

1.1. Com sabem si un algorisme fa el que ha de fer?

La correcció de l'algorisme només es pot **demostrar** completament usant procediments de raonament matemàtic, però de manera alternativa, tot i que no és una demostració, podem **verificar-ne** el funcionament per a alguns casos provant-lo amb entrades de resultat conegut per comprovar que la sortida és l'esperada.

Per verificar el programa, farem servir **assercions**, instruccions que declaren un fet en un programa. A Python, les assercions es fan amb la instrucció «`assert`» acompanyada d'una expressió booleana (igualtat, desigualtat o comparació). La instrucció verifica si la condició és certa. Si és així, el programa segueix endavant. En canvi, si la condició és falsa, el programa s'atura i retorna un error.

Les assercions són molt útils perquè aturen el programa tan aviat com es produeix l'error i mostren en quin punt del programa s'ha produït, i així ens faciliten depurar el codi. Vegem-ne un exemple.

Si executem el programa:

```
def prog(x):
    return x + 1

assert prog(3) == 4
```

com que hem indicat que el resultat esperat per al valor 3 és 4, i és correcte, la instrucció «`assert`» s'executarà i la condició es verificarà.

En canvi, si haguéssim indicat que el resultat esperat és 9, hauríem usat

```
assert prog(3) == 9
```

i en no complir-se la condició, la instrucció «`assert`» hauria generat un error, en què s'indicaria que el programa no fa el que volem.

1.2. Com sabem si un algorisme és eficient?

Un algorisme és **eficient** si utilitza el nombre mínim de recursos (cicles de càlcul i memòria de l'ordinador) possible. Fer servir algorismes eficients sempre és adient i moltes vegades és una necessitat, ateses les dimensions del problema.

En aquest llibre ens centrem sobretot en els cicles de càlcul computacional dels algorismes, o dit d'una altra manera, en la seva **complexitat de càlcul**.

Cal tenir present que, a cada cicle de càlcul computacional, l'ordinador pot dur a terme una operació simple entre dos valors (normalment de 64 bits): $+$, $-$, $/$, etc. Ara bé, si els nombres per operar són més grans de 64 bits i no es poden operar directament amb el maquinari de l'ordinador, el nombre de cicles necessaris augmentarà i caldrà calcular quin és. Un dels objectius d'aquest llibre és aprendre a calcular aquest cost computacional.

1.3. Notació gran O

Per definir la complexitat computacional d'un algorisme, fem servir la notació gran O, una convenció que ens permet comparar fàcilment dos algorismes entre ells.

La dada més rellevant en el càlcul de la complexitat és el nombre aproximat de passos computacionals que fa l'algorisme **en funció de la mida de l'entrada**.

Quan el nombre de passos que ha de fer l'algorisme no depèn exclusivament de la mida de l'entrada, sinó que també hi influeixen altres factors lligats al problema, tindrem en compte el que s'anomena **el pitjor cas**, és a dir, el nombre de passos que un algorisme pot arribar a fer quan es troba en les pitjors condicions del problema. En general, en la notació gran O ens fixem en aquest cas.

Per simplificar l'expressió de la complexitat mitjançant la notació gran O, farem esment només de la complexitat més gran, obviant els termes més petits. Per exemple, si un algorisme, donada una entrada de mida N, fa $5N^3 + 4N + 3$ passos computacionals, en notació gran O direm que té un ordre de complexitat de $O(N^3)$, ja que és el terme més gran, i per tant més rellevant, de tots els anteriors.

Per arribar a aquesta simplificació, només cal aplicar les regles següents:

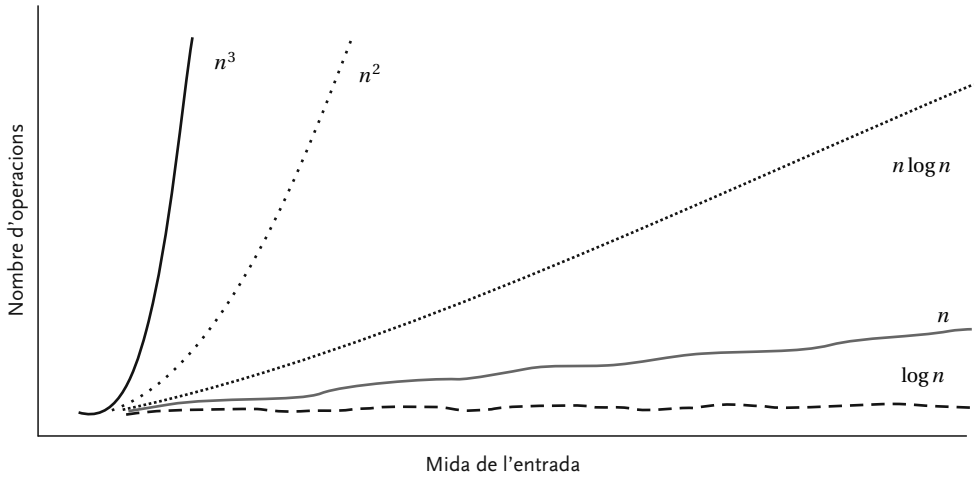
- Ometre les constants multiplicatives: $14N^2$ passos és $O(N^2)$.
- N^a domina sobre N^b si $a > b$: $N^2 + N$ passos és $O(N^2)$.
- Qualsevol exponencial domina sobre un polinomi: $3^N + N^5$ passos és $O(3^N)$.
- Qualsevol polinomi domina sobre un logaritme: $N + \log(N)^3$ passos és $O(N)$ i $N^2 + N\log(N)$ és $O(N^2)$.
- Un terme $N!$ domina sobre qualsevol altre.

Donat l'ordre O d'un algorisme, el podem agrupar en famílies de complexitat. A la taula 1.1 hi veiem aquestes famílies, i a la figura 1.1 hi veiem el seu creixement en funció de la mida de l'entrada.

Taula 1.1. Taula amb les grans famílies d'algorismes segons la complexitat.

Constant	Logarítmic	Lineal	Superlineal	Quadràtic	Cúbic	Exponencial	Factorial
$O(1)$	$O(\log(n))$	$O(n)$	$O(n\log(n))$	$O(n^2)$	$O(n^3)$	$O(c^n)$ amb $c>1$	$O(n!)$

Figura 1.1. Gràfic amb el creixement del cost de les funcions segons l'ordre de complexitat



Si expressem el creixement del cost de les funcions amb valors concrets, el que tenim és això:

N	$\log N$	N^2	2^N	$N!$
5	0,6	25	32	120
6	0,7	36	64	720
7	0,8	49	128	5.040
8	0,9	64	256	40.320
9	0,95	81	512	362.880
10	1	100	1.024	3.628.800
50	1,6	2.500	1,1258E + 15	3,0414E + 64

D'aquests nombres, en podem treure algunes conclusions:

- Qualsevol algorisme factorial, $N!$, és inútil des d'un punt de vista pràctic a partir de $N = 20$.
- Els algorismes exponencials, a^N amb una a petita, són inútils a partir de $N = 40$.
- Els algorismes quadràtics, N^2, comencen a ser costosos a partir de $N = 10.000$ i a ser inútils a partir de $N = 1.000.000$.
- Els algorismes lineals, N, i els logarítmics, $N\log(N)$, poden arribar fins a $N = 1.000.000.000$.
- Els algorismes sublineals, $\log(N)$, són útils per a qualsevol N.

1.4. Càlcul de la complexitat d'un algorisme

A continuació exposem els conceptes necessaris per calcular la complexitat d'un algorisme simple escrit en Python.

Complexitat d'una operació simple

Les operacions simples a Python són:

- Operacions aritmètiques: «`+, -, *, /, %, **`».
- Operacions d'assignació: «`=, +=, -=, *=, /=, %=, //=, **=`».
- Operacions de comparació: «`==, !=, >, <, <=, >=`».
- Operacions lògiques: «`and, or, not`».
- Assignacions de variables: «`variable = 2`».

En general, la complexitat d'aquests casos és d'ordre **constant**, $O(1)$, ja que no depèn de la mida de l'entrada. Aquesta complexitat pot ser superior en alguns casos, com quan els nombres de les operacions aritmètiques són molt grans.

Complexitat dels operadors de llistes

Llistes

Les llistes a Python són seqüències mutables d'objectes arbitraris. Les llistes es manipulen amb diferents mètodes i s'accedeix als seus elements amb els operadors de *slicing*.

Quan treballem amb llistes, algunes operacions aparentment simples poden adquirir una complexitat ben diferent. Per exemple:

```
llista.append('a')   # Complexitat O(1)
llista.insert(2,'a') # Complexitat O(n)
```

En el primer cas estem afegint un nou element a l'última posició de la llista amb cost constant 1. En canvi, en el segon, inserim un element a la segona posició, que obliga a desplaçar la resta de posicions de la llista (n posicions).

A la documentació de Python es poden trobar les complexitats de les operacions amb llistes i estructures de dades més habituals.[2]

Complexitat d'un bloc condicional

Els blocs condicionals tenen un comportament diferent al de les operacions simples. Considerem el codi de Python següent:

2 https://wiki.python.org/moin/TimeComplexity

```
if cond:
    op1
elif:
    op2
else:
    op3
```

En no saber, *a priori*, quina de les tres operacions «`op1, op2, op3`» s'executarà, hem de considerar el pitjor cas possible. Per exemple, si la complexitat de «`op1`» és $O(n)$, la complexitat de «`op2`» és $O(n^2)$ i la complexitat de «`op3`» és $O(\log(n))$, la complexitat del bloc condicional serà $O(n^2)$, ja que és la màxima complexitat d'entre les tres possibles.

Complexitat d'un conjunt d'instruccions

Per calcular la complexitat d'un conjunt seqüencial d'instruccions, hem de sumar-ne totes les complexitats. Un cop sumades, ens hem de quedar amb els termes dominants seguint les normes de la notació gran O, com ja hem vist a l'apartat 1.3.

Considerem el codi següent, amb indicació de la complexitat de les instruccions:

```
def funcio():
    op1       # Complexitat O(1)
    op2       # Complexitat O(n)
    op3       # Complexitat O(n)
    op4       # Complexitat O(n^2)
    op5       # Complexitat O(log n)
    if cond:
        op6   # Complexitat O(n)
    else:
        op7   # Complexitat O(1)
```

Així, la complexitat del programa fins que executa «`op5`» serà d'$1+2n+n^2+\log(n)$, que sabem que és d'ordre $O(n^2)$, que és el terme dominant. Si ara hi afegim la complexitat del bloc condicional (en què «`op6`» és $O(n)$ i «`op7`» és $O(1)$), la complexitat total és de $O(n^2)$, és a dir, es manté igual, ja que sabem que n^2 domina sobre n en la notació gran O.

Complexitat dels blocs iteratius (bucles)

En el cas dels bucles iteratius (*loops*, en anglès), primer cal calcular la complexitat de les operacions dins el bucle, i després multiplicar aquest nombre pel nombre de vegades que iterem.

```
for i in range(0, n):
    op1
```

A l'exemple, si la complexitat de «`op1`» és $O(1)$, la complexitat del bloc és $O(n)$, ja que iterem n vegades sobre aquesta operació. En canvi, si la complexitat de «`op1`» és $O(\log(n))$, la complexitat total del bloc iteratiu serà de $O(n\log(n))$.

Complexitat dels blocs iteratius imbricats

Els bucles imbricats (*nested loops*, en anglès) són un cas específic de l'apartat anterior. Aquí hi afegim un nivell més d'iteració i, per tant, hem de calcular el nombre de vegades que es repeteix cada bloc per arribar a la complexitat total.

Si, per exemple, tenim el codi següent:

```
for i in range(1, n):
    for j in range(1, n):
        op1
```

la complexitat total serà n^2 multiplicada per la complexitat de la instrucció «`op1`». Suposant, per exemple, que aquesta instrucció té complexitat $O(n)$, la complexitat total d'aquest codi serà de $O(n^3)$.

1.1. Problema. Notació gran *O*

1. Segons les indicacions donades anteriorment, calcula l'ordre de complexitat segons la notació gran O d'uns algorismes que fan els nombres d'operacions següents:

- $n+2n$
- $3n^2+\log n$
- 2^n+n^5
- $n^3+n^2\log(n)+n+5$

2. Té sentit un algorisme de complexitat $O(n!)$ per a $n=50$? Per què?

3. Calcula la complexitat total d'aquests codis d'exemple:

```
def codi1():
    op1      # Complexitat O(1)
    op2      # Complexitat O(n)
    for i in range(0, n):
        op3  # Complexitat O(n)
    if cond:
        op4  # Complexitat O(n)
    else:
        op5  # Complexitat O(n^2)


def codi2():
    for i in range(0,2*n):
        for j in range(0, n*n):
            op1     # Complexitat O(n)
```

Capítol 2

ALGORISMES BÀSICS I PYTHON

La millor manera de començar a treballar en algorísmica és familiaritzar-se amb la resolució de problemes simples i concrets. En aquest primer capítol s'introdueixen algorismes molt senzills per treballar la gramàtica i la sintaxi de Python, i també per començar a practicar l'art de calcular-ne la complexitat computacional.

2.1. Problema. Celsius a Fahrenheit

Donada una temperatura en graus Celsius, x, podem convertir-la fàcilment a graus Fahrenheit, y, usant la fórmula:

$$y = \frac{9}{5}x + 32$$

1. Escriu una funció **convert_temp** que converteixi els graus Celsius en graus Fahrenheit. Fes servir aquesta plantilla per escriure el teu codi:

```
def convert_temp(temp_Celsius):
    # El teu codi
    return temp_Fahrenheit
```

2. Utilitzant la funció anterior, escriu una nova funció **convert0a50** que calculi i imprimeixi una llista de temperatures Celsius i dels seus equivalents Fahrenheit cada 5 graus, de 0 °C a 50 °C.

   Fes servir aquesta plantilla per escriure el teu codi:

```
def convert0a50():
    # El teu codi
    return list_temp_Fahrenheit

print(convert0a50())
```

Reflexions prèvies. Fes aquestes reflexions abans de començar a programar:

- La funció «`convert0a50()`» pot cridar la funció «`convertTemp`» cada cop que vulgui fer una conversió de temperatura.
- Quin tipus d'iteració pots usar per crear els diferents valors de Celsius? Per què?
- Quina col·lecció faràs servir per guardar els valors de les temperatures Fahrenheit? Com hi afegiràs nous valors quan els calculis?
- Com pots validar el resultat de la funció?

2.2. Problema. Inversió econòmica

Dins de les matemàtiques financeres, l'interès compost és un concepte important fins i tot per a les nostres finances personals.

Suposem que fem una inversió determinada amb l'objectiu que ens retorni uns interessos durant un període més o menys llarg de temps. El més normal és que aquest període es divideixi en una sèrie de subperíodes de longitud fixa (mesos o anys). En el cas de l'interès compost, els interessos que es produeixen en cada subperíode es van sumant al capital inicial al final de cada subperíode. D'aquesta manera es generen nous interessos. El capital va creixent al final de cada subperíode, així que els interessos es calculen cada vegada sobre un capital més gran.

1. La funció «`futval`» calcula els diners que tindrem en un compte del banc al cap de 10 anys en funció de la inversió inicial (en euros) i de l'interès que ens dona el banc (en %) suposant un subperíode anual:

```
def futval(inversio, interes):
    principal = inversio
    for i in range(10):
        principal = principal * (1 + interes)
    return principal

# Invertim 100 euros al 10%
futval(100, 0.1)
>>> 259.37424601000026
# Al cap de 10 anys disposem de 259 euros i escaig
```

   Escriu, a partir d'aquesta funció, una segona funció amb alguns canvis. En aquest cas, el banc cobrarà una comissió d'un percentatge de l'import final. El programa ha de rebre la inversió inicial, l'interès, el nombre d'anys i també la comissió del banc (per exemple, si la comissió és del 3%, haurem d'entrar un 0.03).

   Fes servir aquesta plantilla per escriure el teu codi:

```
def futval2(inicial, anys, interesPerCent,
↪  comissioPerCent):
    # El teu codi
return diners
```

   Reflexions prèvies. Fes aquestes reflexions abans de començar a programar:

   - Quin càlcul hem d'aplicar? Prova'l amb alguns valors bàsics manualment.
   - Explica els passos que fa l'algorisme en el cas de «`futval2(100,10,0.8,0.03)`».

   Banc de proves. Defineix diversos exemples per comprovar la teva solució. Observa que, com que en aquest cas la funció retorna un valor real, no podem usar l'operador «==», sinó que farem el test per aproximació:

```
assert abs(futval2(100,10,0.8,0.3) - 24993.270) < 0.1
```

No s'ha de fer mai un test del tipus «`(a == 2.3)`», perquè la representació dels nombres decimals ens pot jugar alguna mala passada. Sempre hem de convertir aquesta comparació en un càlcul aproximat amb la precisió que desitgem: «`(a - 2.3 < 0.00001)`».

2.3. Problema. Condicionals

1. **Prou cerveses**. Quan uns estudiants organitzen una festa els agrada beure cerveses. Perquè una festa tingui èxit cal que hi hagi entre 50 i 100 cerveses, excepte durant el cap de setmana, en què no hi ha un límit superior de cerveses.

   Escriu una funció que, donats un nombre de cerveses i un booleà que indiqui si és cap de setmana o no, retorni una cadena de caràcters que expliquin si la festa serà un èxit o no. Proposa uns valors donats a la funció que provoquin un èxit de festa entre setmana i un èxit de festa el cap de setmana.

   Fes servir aquesta plantilla per escriure el teu codi:

   ```
   def prouCerveses(nre_cerveses, es_cap_de_setmana):
       # El teu codi
   ```

   Reflexions prèvies. Fes aquestes reflexions abans de començar a programar:

   - Detalla les condicions que fan que una festa sigui un èxit i les que fan que una festa sigui un fracàs.
   - Explica els passos que fa l'algorisme per a «`prou_cerveses(75, False)`».
   - Pensa en les diferents combinacions que es poden donar i comprova que el teu codi les tingui en compte totes.

   Banc de proves. Defineix diversos exemples per comprovar la teva solució. Entre d'altres, executa les comprovacions següents:

   ```
   assert prouCerveses(75, False) == 'Èxit'
   assert prouCerveses(125, False) == 'Fracàs'
   assert prouCerveses(125, True) == 'Èxit'
   ```

2. **Qualificacions**. Escriu una funció que, donat un nombre amb la qualificació numèrica d'un examen, proporcioni la qualificació qualitativa corresponent al nombre donat:

   - Suspès: nota menor que 5.
   - Aprovat: nota igual o més gran que 5 i menor que 7.
   - Notable: nota igual o més gran que 7 i menor que 9.
   - Excel·lent: nota igual o més gran que 9 i menor que 10.
   - Matrícula d'honor: nota igual a 10.

Fes servir aquesta plantilla per escriure el teu codi:

```
def nota(nre):
    # El teu codi
    return qualificacio
```

Reflexions prèvies. Fes aquestes reflexions abans de començar a programar:

- Quines comparacions has de fer per saber si una nota és un suspès?
- Si ja saps que una nota no és un suspès, quines comparacions has de fer per saber si és un aprovat?
- Si ja saps que una nota no és un suspès ni un aprovat, quines comparacions has de fer per saber si és un notable?
- Explica els passos que fa l'algorisme per a «`nota(9.3)`».
- Si encara no ho has fet, se t'acudeix reformular l'algorisme usant una col·lecció? Quina? Pensa que potser has de fer alguna petita manipulació de la nota que et donen.

Banc de proves. Defineix diversos exemples per comprovar la teva solució. Entre d'altres, executa les comprovacions següents:

```
assert nota(8.9) == 'Notable'
```

2.4. Problema. Pendent d'una recta

1. Considera dos punts en un pla segons les seves coordenades (x_1, y_1) i (x_2, y_2). Escriu una funció que calculi el pendent de la recta per a aquests dos punts. Pots calcular el pendent fàcilment usant l'expressió:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

Fes servir aquesta plantilla per escriure el teu codi:

```
def pendent(x1,y1,x2,y2):
    # El teu codi
    return m
```

Reflexions prèvies. Fes aquestes reflexions abans de començar a programar:

- Hi ha alguna combinació dels valors dels paràmetres que podria provocar un error d'execució? Afegeix algun mecanisme de control perquè el programa no retorni un error en cap cas i decideix què es retornarà en els casos problemàtics.
- Explica els passos que fa l'algorisme quan executem la instrucció «`pendent(1, 3, 1, 5)`».