# Problemes bàsics d'algorísmica

Mireia Ribera, Jordi Vitrià, Pablo Laiz i Pere Gilabert
**Universitat de Barcelona**

Abril de 2021

# 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 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 aquell 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**.

[1] Donald Knuth, un dels pares de l'algorísmica, assumeix que els algorismes són quelcom destinat a ser llegit per humans i només esporàdicament executat per ordinadors.

- 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 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 solucionem 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 es proporciona està escrit en Python, un dels llenguatges que més a prop està del pseudocodi. Tots els programes d'exemple que s'inclouen estan degudament comentats i explicats per entendre'n el funcionament i execució.

```python
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 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 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. En 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 ens faciliten així 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 pel valor 3 és 4, i és correcte, la instrucció `assert` s'executarà i la condició es verificarà.

En canvi, si indiquem 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 de l'ordinador 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 realitzar una operació simple entre dos valors (normalment de 64 bits): $+$, $-$, $/$, etc. Ara bé, si els nombres a operar són més grans de 64 bits i no poden ser operats 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 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 veiem aquestes famílies, i a la figura 1.1 veiem el seu creixement en funció de la mida de l'entrada.

| **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 $c > 1$ | $O(n!)$ |

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

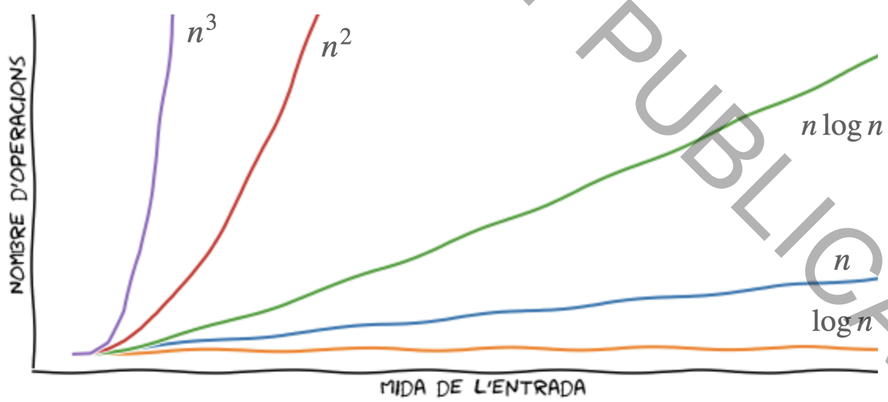



Figura 1.1: Gràfic amb el creixement del cost de les funcions segons l'ordre de complexitat.

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 | 1024 | 3.628.800 |
| 50 | 1,6 | 2500 | 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 en 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 que operem amb les operacions aritmètiques són molt grans.

### Complexitat dels operadors de llistes

**Llistes**
Les llistes en 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 tenir 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:

```
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 d'`op1` és $O(n)$, la complexitat d'`op2` és $O(n^2)$ i la complexitat d'`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 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 la secció 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)
```

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

Així, la complexitat del programa fins que executa `op5` serà de $1+2n+n^2+\log(n)$ 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 d'`op1` és $O(1)$, la complexitat del bloc és $O(n)$, ja que iteren $n$ vegades sobre aquesta operació. En canvi, si la complexitat d'`op1` és $O(\log(n))$, la complexitat total del bloc iteratiu serà de $O(n\log(n))$.

### Complexitat dels blocs iteratius imbricats

El cas dels bucles imbricats (*nested loops*, en anglès) és un cas específic de l'apartat anterior. Aquí 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 el nombre d'operacions següent:

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

**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:

```python
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

### Conversió de 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:

```python
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:

```python
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 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 divideix 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-ho 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 en la qual donat 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 més gran o igual a 7 i menor que 9.
- Excel·lent: nota més gran o igual a 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 no ho has fet ja, 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 retornar en els casos problemàtics.
- Explica els passos que fa l'algorisme quan executem la instrucció `pendent(1, 3, 1, 5)`.

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

```
assert pendent(1,1,4,4) - 1.0 < 0.000001
assert pendent(4,5,2,9) - (-2.0) < 0.000001
```

**2.** Estén el teu programa perquè calculi l'equació de la recta ($y = mx + n$) que passa pels punts amb coordenades $(x_1, y_1)$ i $(x_2, y_2)$. Aprofita el càlcul del pendent de l'exercici anterior. El programa no ha de retornar cap valor, sinó que ha d'imprimir l'equació de la recta en format *string* de la manera següent: si la recta té un pendent $m$ i una ordenada en l'origen $n$, ha d'imprimir: `y = mx + n`.
Fes servir aquesta plantilla per escriure el teu codi:

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

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

- Observa que en aquest cas concret, aquesta funció no retorna cap valor. Només ha d'imprimir l'equació de la recta en el format $y = mx + n$ amb els valors adequats de $m$ i $n$.
- El valor de $m$ ja ve donat per la funcio `pendent`. Com calculem el valor $n$?
- Has considerat el cas $m = 0$? I el cas $n = 0$? I els casos $m = 1$, $m = -1$?

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

```
equacio(1,1,4,4)
>>> y = x + 3

equacio(4,5,2,9)
>>> y = -2x + 13

equacio(-1,-3,3,5)
>>> y = 2x - 1
```

## 2.5 Problema

### Capicua (palíndrom)

Una paraula, frase o grup de paraules són capicua, o el que és el mateix, formen un palíndrom, si les lletres es repeteixen en el mateix ordre quan són llegides en direcció inversa, com ara a `anna` o `rajar`.

---

**1.** Escriu una funció que donada una cadena de caràcters determini si és capicua.
Fes servir aquesta plantilla per escriure el teu codi:

```
def es_capicua(paraula):
    # el teu codi
    return b_es-capicua
```

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

```
assert es_capicua("llull") == True
assert es_capicua("llullu") == False
```

**2.** Escriu una funció que, donada una cadena de caràcters, determini si en aplicar rotacions de lletres, és capicua. Per resoldre aquest problema has d'anar creant les paraules rotades progressivament i validant si són capicua amb l'algorisme anterior.
Una rotació consisteix a situar l'última lletra en la primera posició o viceversa. Per exemple, en la cadena `ABCD` les rotacions serien: `[DABC,CDAB,BCDA,ABCD]`.
Fes servir aquesta plantilla per escriure el teu codi:

```
def es_capicua_rotacions(paraula):
    # el teu codi
    return b_es_capicua
```

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

```
assert es_capicua_rotacions("BAABCC") == True
```

## 2.6 Problema

### Palíndrom

Resol els problemes següents a partir de la definició anterior de palíndrom.

**1.** Donada una cadena de caràcters, troba tots els palíndroms generables a partir de les seves lletres. En aquest exercici pots fer servir la biblioteca de Python `itertools` per crear les possibles combinacions.
Fes servir aquesta plantilla per escriure el teu codi:

```
import itertools

def generar_palindroms(cadena):
    # el teu codi
```

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

- Fixa't en alguns palíndroms i observa:
  - Quantes vegades apareix cada lletra?
  - Quantes lletres amb un nombre senar d'aparicions hi ha?
  - Si hi ha una lletra que apareix un sol cop on és?
  - Si divideixes el palíndrom en 2 com són les parts esquerra i dreta? Hi veus algun patró?
- Pots resoldre el problema pas a pas seguint aquestes indicacions:
  - Crea la funció que només funcioni amb paraules amb lletres que apareguin 2 cops cadascuna.
  - Millora la funció amb paraules que tinguin lletres repetides 2 vegades + 1 lletra que aparegui un sol cop.
  - Millora la funció amb paraules que tinguin lletres repetides 2 vegades + 1 lletra que aparegui tres, cinc o un nombre senar de cops.
  - Millora la funció descartant ràpidament algunes paraules per la freqüència de les lletres que contenen.
- Quina solució hauria de donar per als valors següents: `ab` i `aabtb`?
- Explica els passos que fa l'algorisme en el cas `generar_palindroms('aabtb')`.

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució.

```
generar_palindroms('aann')
 anna
 naan
```

## 2.7 Problema

### Divisors

Siguin $m$ i $n$ dos nombres enters. Es diu que $m|n$ (que es llegeix: $m$ divideix $n$, o $m$ és divisor de $n$) si i només si existeix un element $k$ tal que $n = k \times m$.

---

**1.** Escriu una funció en què, donat un nombre menor que 30, retorni tots els seus divisors.
Fes servir aquesta plantilla per escriure el teu codi:

```python
def divisors(num):
    # El teu codi
    return divisors
```

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

- Com identifiquem els divisors d'un nombre?
- Quins divisors hauria de retornar la funció en els casos següents: 28, 12, 8, 3?
- Explica els passos que fa l'algorisme en el cas que el nombre donat sigui 28.
- Si no ho has fet ja, reescriu el programa amb una *list comprehension*.

***List Comprehension***
Una *list comprehension* permet escriure llistes ràpidament. Per exemple, podem crear una llista amb els múltiples de 3 menors a 30 usant el codi `[m for m in range(1,30,3)` o bé `[3*m for m in range(1,30)]`

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

```python
assert divisors(8) == [1,2,4,8]
assert divisors(5) == [1,5]
assert dividors(210) == [1, 2, 3, 5, 6, 7, 10, 14,
 ↪  15, 21, 30, 35, 42, 70, 105, 210]
```

## 2.8 Problema

### Factorial menor que un nombre donat

El factorial d'un enter no negatiu $n$, denotat per $n!$, és el producte de tots els nombres enters positius inferiors o iguals a $n$.

---

**1.** Escriu una funció, `factorial_menor`, que retorni tots els valors menors que un nombre donat i que són factorials d'algun nombre natural. No es pot usar cap funció específica de Python per calcular els factorials.
Fes servir aquesta plantilla per escriure el teu codi:

```
def factorial_menor(num):
    # El teu codi
    return factorials
```

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

- Quin és el factorial d'1? i de 2? i de 3? i de 4?... Quina és la fórmula general del factorial per a un nombre qualsevol?
- Per calcular el factorial, hauràs d'utilitzar algun tipus d'iteració. Quina utilitzaràs?
- Explica els passos que fa l'algorisme per a `factorial_menor(25)`.

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

```
assert factorial_menor(20) == [1, 2, 6]
assert factorial_menor(50) == [1, 2, 6, 24]
```

## 2.9 Problema

### Mínim i màxim

**1.** Donada una llista de nombres enters, troba el valor mínim i el valor màxim amb un algorisme de complexitat $O(n)$. No utilitzis cap funció específica de Python [com ara les funcions `min()`, `max()`] per resoldre el problema.
Fes servir aquesta plantilla per escriure el teu codi:

```
def minim_maxim(llista):
    # El teu codi
    return _min, _max
```

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

- Utilitza una iteració per recórrer la llista d'entrada. Passant només un cop de principi a fi, has de poder calcular el nombre mínim i el nombre màxim.
- Explica els passos que fa l'algorisme per a `minim_maxim([1, 7, 4, 6, 8, -2, 9, 5])`.
- Com ho faries si saps que la llista està ordenada? Quin cost tindria?

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

```
assert minim_maxim([3, 1, 5, 2, 7, 8]) == (1, 8)
assert minim_maxim([1, 7, 4, 6, 8, -2, 9, 5]) == (-2,
↪  9)
```

## 2.10 Problema

### Sumatori de parelles

---

1. Donada una llista d'enters i un valor de suma, troba, de la manera més eficient possible, totes les parelles de nombres a la llista que sumin aquest valor. L'algorisme pot tenir una complexitat d'ordre $O(n^2)$ però el cost exacte ha de ser menor que $n^2$.
Fes servir aquesta plantilla per escriure el teu codi:

```
def sumatori_parelles(llista, valorSuma):
    # El teu codi
    return parelles
```

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

- Si la llista té 3 elements, quantes comparacions hem de fer? Com podem evitar repeticions?
- Explica els passos que fa l'algorisme en el cas `sumatori_parelles([3, 1, 5, 2, 7, 8], 10)`.
- Quins casos hem de tenir en compte?
- Calcula la complexitat de l'algorisme.

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

```
assert sumatori_parelles([3, 1, 5, 2, 7, 8], 10)  ==
↪  [(2, 8), (3, 7)]
```

## 2.11 Problema

### Sumes i quadrats

La suma dels quadrats dels primers 10 nombres naturals és $1^2+2^2+...+10^2 = 385$. El quadrat de la suma dels primer 10 nombres naturals és $(1+2+...+10)^2 = 55^2 = 3025$. Per tant, la diferència entre la suma d'aquests quadrats i el quadrat de la suma és $3025 - 385 = 2640$. Troba aquesta diferència per als 100 primers nombres naturals.

---

**1.** Resol el problema fent servir *list comprehensions*.
Fes servir aquesta plantilla per escriure el teu codi:

```
def suma_quadrats(n = 100):
    # El teu codi
    return diferencia
```

**Comentari**: amb l'expressió $n = 100$ com a paràmetre de la funció, el que estem fent és donar un valor per defecte a $n$.

**Reflexions prèvies:** Per resoldre el problema pots seguir les indicacions següents pas a pas:

- Escriu la *list comprehension* per calcular una llista amb els quadrats dels 100 primers nombres naturals. Un cop feta la llista pots calcular-ne la suma amb l'operació *sum* de Python.
- Escriu la *list comprehension* per calcular una llista amb els 100 primers nombres naturals. Un cop feta la llista, calcula'n la suma i calcula també el quadrat d'aquest nombre.
- Calcula la diferència entre els dos valors obtinguts.
- Quina és la complexitat de l'algorisme? Tingues en compte la complexitat de la funció *sum*, si l'has utilitzat.

**Banc de proves:** Comprova que has programat l'algorisme correctament utilitzant la instrucció següent

```
assert suma_quadrats() == 25164150
```

**2.** Si no fem servir *list comprehensions* és possible solucionar el problema amb una complexitat $O(n)$. Escriu un programa que ho faci.
Fes servir aquesta plantilla per escriure el teu codi:

```
def suma_quadrats_lineal():
    # el teu codi
    return diferencia
```

**Reflexions prèvies:** Per resoldre el problema pots seguir les indicacions següents pas a pas:

- Utilitza una sola iteració, que recorri tots els nombres menors a 100. Quina informació t'has de guardar de cada un d'ells?

**Banc de proves:** Pots comprovar que el resultat és el mateix que el de l'exercici anterior executant la instrucció següent:

```
assert suma_quadrats() == suma_quadrats_lineal()
```

## 2.12 Problema

### Nombres amics

Diem que dos nombres són amics si els divisors d'un, sumats, són equivalents a l'altre nombre i viceversa. Per exemple:

- Els divisors de 220 són 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 i 110, que sumen 284.
- Els divisors de 284 són 1, 2, 4, 71 i 142, que sumen 220.

Per tant, **220** i **284** són amics.

---

**1.** Escriu una funció que determini si una parella de nombres són amics.
Fes servir aquesta plantilla per escriure el teu codi:

```
def amics(nre1, nre2):
    # El teu codi
    return sonAmics
```

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

- Pensa que s'ha de fer una mateixa operació sobre els dos nombres. Pots fer una funció auxiliar que serveixi per als dos casos.
- Explica els passos que fa l'algorisme en el cas `amics(220, 284)`
- Quins casos especials hem de tenir en compte?

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

```
assert amics(220, 284) == True
assert amics(200, 230) == False
assert amics(9363584, 9437056) == True
```

## 2.13 Problema

### Nombres perfectes

Diem que un nombre enter positiu és **perfecte** si és la suma dels seus divisors positius i menors que ell mateix. Per exemple, el nombre 6 és un nombre perfecte ja que els seus divisors són 1, 2 i 3 i es compleix que $6 = 1 + 2 + 3$.

---

**1.** Escriu una funció que determini si un nombre és perfecte i retorni un valor booleà amb el resultat.
Fes servir aquesta plantilla per escriure el teu codi:

```
def perfecte(nre):
    # El teu codi
    return es_perfecte
```

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

- Repassa manualment i compta els passos que fa l'algorisme per a `perfecte(6)`.
- Quins casos especials hem de tenir en compte?

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

```
assert perfecte(6) == True
assert perfecte(8) == False
assert perfecte(28) == True
```

## 2.14 Problema

### Nombres apocalíptics

Un nombre, $x$, es diu **apocalíptic** si la xifra que representa $2^x$ conté la seqüència 666.

---

**1.** Escriu una funció que determini si un nombre és apocalíptic. Un cop escrit el programa, contesta les preguntes següents:

- Quins dels nombres següents són apocalíptics? 157, 180, 192.
- Dona una llista amb tots els nombres apocalíptics entre 100 i 300.
- Quants nombres apocalíptics hi ha menors que 5000?

Fes servir aquestes plantilles per escriure el teu codi:

```python
def apocaliptic(nre):
    # El teu codi
    return es_apocaliptic

def llista_apocaliptics(minim, maxim):
    # El teu codi
    return llista
```

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

- Com podem saber si la seqüència 666 apareix al nombre? Amb el valor numèric de $2^x$ és difícil de saber. Quin altre tipus de variable podem fer servir?
- Explica els passos que fa l'algorisme en el cas `apocaliptic(157)`.

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

```
assert apocaliptic(157) == True
assert apocaliptic(180) == False
assert apocaliptic(192) == True

assert llista_apocaliptics(100,300) == [157, 192,
↪ 218, 220, 222, 224, 226, 243, 245, 247, 251, 278,
↪ 285, 286, 287]
assert len(llista_apocaliptics(0,5000)) ==  2253
```

## 2.15 Problema

### Nombre feliç

Un nombre es diu **feliç** si en transformar-lo de forma iterativa en la suma dels quadrats dels seus dígits, fins que només quedi una xifra, acabem amb un 1.

Per exemple, si considerem el 203 i apliquem la definició:

- $203 : 2^2 + 0^2 + 3^2 = 13$
- $13 : 1^2 + 3^2 = 10$
- $10 : 1^2 + 0^2 = 1$

podem veure que 203 és un nombre feliç (i que 13 i 10 també ho són!).

---

**1.** Escriu una funció que determini si un nombre és feliç.

Un cop escrita la funció, respon a les preguntes següents:

- Quants nombres feliços hi ha fins a 3000?
- Si el primer nombre feliç és l'1, quin és el nombre feliç número 1234?
- Quins són els primers 10 nombres **infeliços** (nombres no feliços)?

Fes servir aquesta plantilla per escriure el teu codi:

```
def felic(nre):
    # El teu codi
    return es_felic
```

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

- Calcula manualment si el nombre 4 és un nombre feliç i observa que per a certs nombres es pot generar un cicle que farà que el programa no acabi mai. Com es pot evitar que es generin cicles i així fer que el programa no iteri infinitament?
- Quin tipus d'iteració faràs servir? Per què?
- Per agafar els dígits del nombre, tens dues opcions possibles: a partir del text o a partir de la divisió entera mòdul 10. Et recomanem que ho solucionis de les dues maneres.
- Com has de tractar el cas del nombre 4? I el del nombre 16?

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

```
assert felic(203) == True
assert felic(13) == True
assert felic(10) == True
assert felic(96) == False
assert felic(33) == False
```

## 2.16 Problema

### Nombres ambiciosos

Diem que un nombre és **ambiciós** si la suma dels seus factors és, quan apliquem recursivament aquesta definició, un nombre perfecte. Per exemple, 95 és un nombre ambiciós ja que:

- Els seus divisors són 1, 5 i 19, que sumen 25. Com que 25 no és perfecte, tornem a aplicar-ho.

- Els divisors de 25 són 1 i 5 que sumen 6 que és un nombre perfecte. Per tant, ens aturem i diem que 95 és ambiciós.

---

**1.** Escriu una funció que determini si un nombre és **ambiciós**.

Fes servir aquesta plantilla per escriure el teu codi:

```
def ambicios(nre):
    # El teu codi
    return es_ambicios
```

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

- Pensa aquest problema com dos subproblemes concrets. El primer, trobar la suma dels divisors; el segon, comprovar si el nombre és perfecte.
- Quin algorisme hem d'aplicar? Com aturarem aquest algorisme?
- Explica els passos que fa l'algorisme en el cas `ambicios(25)`.
- Hi ha casos extrems, com ara el nombre 276, que encara no se sap si és un nombre ambiciós. Afegeix un mecanisme de control que limiti el nombre de comprovacions, per exemple, a 1000.

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

```
assert ambicios(25) == True
assert ambicios(30) == False
assert ambicios(95) == True
```

## 2.17 Problema

### Avet

---

**1.** Donat un enter senar, escriu una funció que dibuixi un avet amb tants asteriscs en la seva part inferior com el nombre donat. Per exemple, `avet(7)` ha de produir:

```
   *
  ***
 *****
*******
```

Fes servir aquesta plantilla per escriure el teu codi:

```
def avet(nre):
    # el teu codi
```

**Reflexions prèvies:** Fes aquestes reflexions abans de començar a programar:
- Dibuixa diversos avets a mà començant per nombres petits.
- Quants asteriscs hi ha a la fila 1? I a la 2? I a la 3? ...
- Quants espais inicials hi ha a la fila 1? I a la 2? I a la 3? ...
- Quin és l'algorisme per calcular el nombre d'asteriscs de cada fila? I per calcular el nombre d'espais inicials?
- Observa que, en aquest cas, la funció no ha de retornar cap valor, només imprimir els caràcters `'*'` i `' '` en l'ordre apropiat.
- Pots imprimir diversos caràcters en una mateixa línia fent servir el paràmetre `end=''` de la funció `print()`.
- Explica els passos que fa l'algorisme en el cas `avet(7)`.
- Si no te'n surts, comença dibuixant l'avet cap per avall; és més senzill.

# Capítol 3

# Algorismes numèrics

Implementar de forma eficient les operacions aritmètiques ha estat una necessitat des del principi de la informàtica per fer els ordinadors eficients. Algunes de les solucions algorísmiques per a aquest problema són conegudes des de l'antiguitat, però l'ús massiu de tecnologies avançades com la criptografia aplicada a seguretat informàtica ha fet encara més important aquesta necessitat.

En aquest capítol repassarem alguns problemes numèrics simples que ens faran adonar del guany que pot suposar un bon algorisme de càlcul numèric sobre estratègies de resolució més simples com pot ser la força bruta.

## 3.1 Problema

### Divisió entera

**1.** Escriu una funció que faci la divisió entera entre dos nombres enters, positius i més grans que 1. No utilitzis els operadors `*`, `/`, `%` per resoldre el problema.

Fes servir aquesta plantilla per escriure el teu codi:

```
def divisio_entera(dividend, divisor):
    # El teu codi
    return quocient
```

> **Reflexions prèvies:** Quantes operacions fa l'algorisme quan calculem 10/2?

> **Banc de proves:** Defineix diversos exemples per comprovar la teva solució. Entre d'altres, executa les comprovacions següents:
>
> ```
> assert divisio_entera(10,2) == 5
> assert divisio_entera(9,2) == 4
> ```

## 3.2 Problema

## El màxim comú divisor i les seves aplicacions

La forma més òbvia de trobar el **màxim comú divisor** (*mcd*, i en anglès *gcd*) de dos nombres és trobar els factors dels dos nombres i multiplicar els seus factors comuns.

Per exemple, el *mcd* de 1035 i 759 surt en comprovar que

$$1035 = 3^2 \times 5 \times 23$$

i que

$$759 = 3 \times 11 \times 23$$

Per tant, $mcd = 3 \times 23 = 69$.

El problema és que no es coneix cap algorisme eficient per **factoritzar** els nombres, és a dir, no existeix cap algorisme publicat que pugui factoritzar-lo en temps $O(n^k)$ independentment de quina sigui la constant $k$.

El millor algorisme que es coneix té aquesta complexitat:

$$O\left(\exp\left(\left(\frac{64}{9}n\right)^{\frac{1}{3}}(\log n)^{\frac{2}{3}}\right)\right)$$

Fa més de dos mil anys que **Euclides** va enunciar un algorisme alternatiu per trobar el màxim comú divisor de dos nombres $a$ i $b$:

```
def gcd(a,b):
    while a:
        a,b = b%a, a
    return b

gcd(1071, 462)
> 21
```

Sabries dir quina complexitat té aquest algorisme per a nombres grans?

No és un cas simple d'anàlisi, però es pot deduir si t'adones dels següents punts:

- A cada iteració els arguments $(a, b)$ es converteixen a $(b \bmod a, a)$: canviem l'ordre i el més gran queda reduït al mòdul del petit.
- Es pot demostrar que això vol dir que **en dues iteracions successives els dos arguments decreixen almenys a la meitat**, és a dir, perden un bit en la seva representació. Si inicialment eren enters de $n$ bits, en $2n$ iteracions arribarem al final de l'algorisme.
- Com que cada iteració implica una divisió ($a \bmod b$) d'ordre quadràtic $O(n^2)$, el temps total serà $O(n^3)$.

---

**1.** Fent servir l'algorisme anterior, escriu una funció que simplifiqui una fracció a la seva expressió irreductible. Recorda que una fracció irreductible és una fracció en la qual el numerador i el denominador no tenen cap divisor comú. Fes servir aquesta plantilla per escriure el teu codi:

```
def reduir_fraccio(numerador, denominador):
    # El teu codi
    return numReduit, denReduit
```

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

- Explica els passos que fa l'algorisme en el cas `reduir_fraccio(12, 8)`.
- Calcula el cost de l'algorisme per `reduir_fraccio(127,63)`.

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució. Entre d'altres, executa la comprovació següent:

```
assert reduir_fraccio(12, 8) == (3,2)
```



## 3.3 Problema

### Numeració en diferents bases

En un sistema de numeració posicional, anomenem ***base*** el conjunt de símbols emprats per a representar qualsevol nombre. En base 2, anomenat *sistema de numeració **binari***, utilitzem només dos símbols: 0 i 1. En base 10, sistema de numeració **decimal**, s'utilitza els 10 símbols numèrics: 0, 1, 2, 3, 4, 5, 6, 7, 8 i 9. En base **hexadecimal**, és a dir, base 16, s'usen els símbols numèrics 0 a 9 i també les lletres A, B, C, D, E i F.

Tot nombre es pot expressar en qualsevol base, només hem d'utilitzar els símbols corresponents. Vegem un exemple per al nombre $58_{10}$ (que s'ha de llegir com "el nombre 58 en base 10"). Observa el subíndex del nombre per representar la base en què està escrit:

$$58_{10} = 8 \times 10^0 + 5 \times 10^1$$

$$3A_{16} = A \times 16^0 + 3 \times 16^1$$

$$11101_2 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 1 \times 2^4$$

Com que el caràcter $A$ en base 16 representa el nombre 10 decimal, fent les operacions de les igualtats observem que les tres representacions corresponen al mateix nombre.

$$58_{10} = 3A_{16} = 11101_2$$

---

**1.** Escriu una funció que, donat un nombre en hexadecimal (base 16), el converteixi a nombre decimal (base 10). L'entrada de la funció serà de tipus *string* i la sortida serà de tipus enter.

Fes servir aquesta plantilla per escriure el teu codi:

```
def convert_hex(xifra):
    simbols = {'A': 10, 'B': 11, 'C': 12, 'D': 13,
               'E': 14, 'F': 15}
    # El teu codi
    return decimal
```

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

- Com pots convertir fàcilment de base 16 a base 10? Utilitza el diccionari `simbols` per convertir els caràcters a nombres.
- Quin exponent li correspon al primer dígit del nombre d'entrada?
- Calcula manualment alguns nombres, com ara, $3A_{16}$, $B6_{16}$, $123_{16}$ i aplica el mateix procediment a l'hora de programar.

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

```
assert convert_hex('3A') == 58
assert convert_hex('B6') == 182
assert convert_hex('123') == 291
```

**2.** Escriu una funció que compti quants dígits té un nombre natural escrit en base 10 i representat per una variable `int` de Python.

Fes servir aquesta plantilla per escriure el teu codi:

```
def digits(xifra):
    # El teu codi
    return nre_digits
```

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

- Es poden comptar els caràcters d'un nombre? I els d'una cadena de caràcters?
- Quina forma té un nombre decimal?
- Explica els passos que fa l'algorisme en el cas *digits(123456)*.
- Quina és l'operació amb més cost d'aquest algorisme?

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

```
assert digits(123456)== 6
```

**3.** Escriu una funció que donat, un número $N$, retorni tots els nombres binaris entre 1 i $N$ dins d'una llista, ambdós inclosos. Pots usar alguna funció específica de Python, com ara la funció `bin()`.
Fes servir aquesta plantilla per escriure el teu codi:

```
def binari(N):
    # El teu codi
    return llista
```

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

- Quin algorisme hem d'aplicar?
- Com sabem la llargada del nombre binari més gran?
- Quina solució generarà *binari(5)*?
- Explica els passos que farà l'algorisme per a *binari(10)*.
- Quins casos hem de tenir en compte?

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

```
assert binari(10) == ['0b1', '0b10', '0b11',
↪ '0b100', '0b101', '0b110', '0b111', '0b1000',
↪ '0b1001', '0b1010']
```

**4.** Escriu una funció que compti la suma dels dígits d'un nombre elevat a una potència. Per exemple, quina és la suma els dígits de $5^{1000}$?

Fes servir aquesta plantilla per escriure el teu codi:

```
def suma_digits(nre, potencia):
    # El teu codi
    return suma
```

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

- Com es calculen les potències a Python?
- Quin algorisme hem d'aplicar? Revisa la funció *digits* si l'has resolt abans.
- Explica els passos que fa l'algorisme en el cas *sumaDigits(2,10)*.

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

```
assert suma_digits(2,100)== 115
```

## 3.4 Problema

### Resta binària

El **complement a un** i el **complement a dos** són dues eines matemàtiques que faciliten molt les tasques aritmètiques en el sistema binari, sobretot la realització de restes i el treball amb nombres negatius.

El complement a un ($C_1$) d'un nombre binari és el nombre resultant d'invertir els uns i els zeros d'aquest nombre. Per exemple, el complement a un del nombre 1101 és el nombre 0010.

El complement a dos ($C_2$) d'un nombre binari és el nombre resultant de sumar 1 al seu complement a un. És a dir, $C_2 = C_1 + 1$. Generalment es diu que el $C_2$ és la manera de representar el negatiu d'un número binari.

Podem restar dos nombres binaris utilitzant l'operació de complement a dos. Siguin $x$, $y$ els dos nombres binaris que volem restar, podem expressar aquesta operació com:

$$\text{Resta binària } x - y = x + C_2(y)$$

---

**1.** Escriu una funció que resti dos nombres binaris que estan emmagatzemats en dues llistes. Considera que els nombres es poden emmagatzemar en 4 bits i que si l'operació que volem fer és $x - y$, el minuend, $x$, ha de ser més gran que el subtrahend, $y$.
Un cop programat, calcula la complexitat de l'algorisme.
Fes servir aquesta plantilla per escriure el teu codi:

```python
def resta_binaria(op1, op2):
    # El teu codi
    return resta
```

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució. Entre d'altres, executa la comprovació següent:

```python
assert resta_binaria([0,1,0,1], [0,0,0,1]) == [0,1,0,0]
```

## 3.5 Problema

### Operacions amb nombres binaris

Els operadors booleans `and` (`&`) i `or` (`|`) permeten realitzar operacions amb la representació binària, o màscara, dels nombres.

Per exemple, l'expressió (`n & 1`) retorna 1 si el nombre és senar i 0 si és parell ja que la màscara corresponent al nombre 1 està actuant sobre l'últim bit de la representació binària del nombre $n$.

Vegem l'aplicació d'aquesta màscara a $n = 20$ i $n = 21$:

```
20 & 1
> 0

21 & 1
> 1
```

En el primer cas, la representació binària del nombre 20 és 00010100. Fent l'operació & amb el nombre 1, en binari de 8 dígits 00000001, obtenim 00010100 & 00000001 = 00000000.

En el segon cas, tenim que 00000001 & 00000001 = 1.

---

**1.** Determina si un nombre és potència de 2 fent ús de les operacions binàries tot seguint les indicacions següents:

- Si ens fixem en la representació binària d'un nombre és fàcil veure que les potències de 2 només tenen un bit a 1.
- Pel que fa a bits, donada una potència de 2 (p. ex. 010000), el nombre anterior té una forma complementària en els bits menys significatius (001111).

Fes servir aquesta plantilla per escriure el teu codi:

```python
def potencia2(nre):
    # El teu codi
    return es_potencia
```

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

- Revisa les operacions amb booleans. Quin algorisme aplicarem?
- Explica els passos que fa `potencia2(8)`.
- Quins casos especials hem de tenir en compte?

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

```python
assert potencia2(1024) == True
assert potencia2(2**2345) == True
assert potencia2(2**2345-1) == False
```

## 3.6 Problema

### Aritmètica modular

En certs aspectes de la informàtica (per exemple, la criptografia) és important una variació de l'aritmètica bàsica sobre els nombres enters: l'**aritmètica modular**.

Definim $x$ **mòdul** $N$, o $x\%N$, com el residu de dividir $x$ entre $N$. Concretament, siguin $x$ i $N$ dos nombres enters, siguin $q$ i $r$ tals que $x = qN + r$ amb $0 \leq r < N$. Aleshores, diem que $x$ mòdul $N$ és $r$.

Per exemple, $12 \% 7 = 5$, i $100 \% 12 = 4$.

Això permet definir una equivalència (congruència) entre nombres enters. Direm que $x$ **és congruent amb** $y$**, mòdul N** ($mod\ N$), si i només si $N$ divideix $x - y$.

Per exemple, 16, 28, 40 i 100 són congruents entre si mòdul 12.

---

**1. Lletra NIF**. L'última lletra del NIF es calcula a partir dels nombres del DNI. Per fer-ho, s'ha de dividir el número entre 23 i quedar-se amb la resta, que és un nombre entre 0 i 22. Llavors s'aplica la taula següent per transformar aquest nombre en una lletra:

| Resta | Lletra | Resta | Lletra |
|---|---|---|---|
| 0 | T | 12 | N |
| 1 | R | 13 | J |
| 2 | W | 14 | Z |
| 3 | A | 15 | S |
| 4 | G | 16 | Q |
| 5 | M | 17 | V |
| 6 | Y | 18 | H |
| 7 | F | 19 | L |
| 8 | P | 20 | C |
| 9 | D | 21 | K |
| 10 | X | 22 | E |
| 11 | B | | |

Escriu una funció, `validar_NIF`, que validi si en un NIF, la lletra es correspon realment al número del DNI. Has de fer servir alguna col·lecció de Python que sigui adequada.

Fes servir aquesta plantilla per escriure el teu codi:

```
def validar_NIF(cadenaNIF):
    # El teu codi
    return es_correcte
```

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

- Com separaràs les dues parts del NIF?
- Quina col·lecció faràs servir? Per què?
- Explica els passos que fa l'algorisme en executar `validar_NIF('33333333T')`.
- Quins casos especials hem de tenir en compte?

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

```
assert validar_NIF('56789123F') == True
assert validar_NIF('56789123H') == False
```

2. **Columnes d'un full de càlcul**. Escriu una funció per convertir un nombre en el nom de la columna del full de càlcul corresponent. Fixa't en els exemples següents:

| | | | | | | |
|---|---|---|---|---|---|---|
| **A**<br>1 | **B**<br>2 | **C**<br>3 | **D**<br>4 | **E**<br>5 | **F**<br>6 | **G**<br>7 |
| **W**<br>23 | **X**<br>24 | **Y**<br>25 | **Z**<br>26 | **AA**<br>27 | **AB**<br>28 | **AC**<br>29 |
| **OP**<br>406 | **OQ**<br>407 | **OR**<br>408 | **OS**<br>409 | **OT**<br>410 | **OU**<br>411 | **OV**<br>412 |
| **ZZ**<br>702 | **AAA**<br>703 | **AAB**<br>704 | **AAC**<br>705 | **AAD**<br>706 | **AAE**<br>707 | **AAF**<br>708 |

Fes servir aquesta plantilla per escriure el teu codi:

```
def conversio_full_calcul(nre):
    # El teu codi
    return columna
```

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

- Quines lletres s'usen per a les columnes Excel?
- Cada quant s'afegeix una nova lletra al nom de la columna?
- Quins casos hem de tenir en compte?

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

```
assert conversio_full_calcul(1000) == 'ALL'
assert conversio_full_calcul(107) == 'DC'
```

## 3.7 Problema

### Descomposició en funcions

Per facilitar la reutilització del codi, i el seu manteniment i llegibilitat, sovint es recomana descompondre'l en parts més petites però amb significat clar mitjançant la definició de múltiples funcions.

Quan t'enfrontes amb un problema, has de mirar d'identificar aquelles parts del codi que resolen un subproblema concret i separar-les del codi principal. Finalment, hi haurà una funció principal, que anirà cridant aquestes funcions més petites de forma ordenada, per resoldre el problema gran.

1. **Calculadora** Escriu una funció que permeti fer operacions de suma, resta i multiplicació amb nombres naturals a partir d'una entrada que sigui una cadena. L'usuari anirà entrant les operacions, i quan vulgui acabar pitjarà retorn i la cadena d'entrada serà buida.
   Exemple de funcionament:

```
Entra operacio: 4+5
>9
```

```
Entra operacio: 7-2
>5
Entra operacio: 1*4
>4
Entra operacio:
```

Fes servir aquesta plantilla per escriure el teu codi:

```python
def calculadora():
    # El teu codi
```

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

- Quin tipus d'iteració cal per anar demanant dades a l'usuari?
- Com separaràs els valors de l'entrada? Quins casos cal tenir en compte?
- Pensa en la descomposició de funcions. Creus que la funció calculadora es pot descompondre en altres? En quines?

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució.

2. **Divisible per tots.** El nombre 2520 és el nombre més petit que es pot dividir de manera exacta (sense decimals) per cada un dels nombres enters entre 1 i 10. Escriu un algorisme que calculi el nombre més petit divisible per tots els nombres menors a un nombre $n$ donat.
Fes servir aquesta plantilla per escriure el teu codi:

```python
def mes_petit_divisible_per(n):
    # El teu codi
    return number
```

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

- Escriu una funció a banda, `divisible`, per comprovar que un nombre pot ser dividit de manera exacta per tots els nombres des de 2 fins al nombre donat. Per exemple `divisible(6,3)` ha de retornar `True` perquè 6 és divisible per 3 i per 2. La funció principal cridarà aquesta funció auxiliar.

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

```python
assert divisible(6, 3) == True
assert divisible(10, 4) == False
assert mes_petit_divisible_per(2) == 2
assert mes_petit_divisible_per(9) == 2520
```

## 3.8 Problema

### Primeritat

Suposem que volem trobar un nombre primer qualsevol de $n$ bits.

El sedàs d'Eratòstenes és un algorisme antic per cercar tots els nombres primers fins a un determinat enter. Va ser creat per Eratòstenes (276-194 aC), un matemàtic de l'Antiga Grècia, i té la forma següent:

1. Escriu una llista *llista1* amb els nombres del 2 fins a l'enter més gran $N$ que vulguis calcular.
2. El primer nombre de la llista és un nombre primer. Anota'l en una llista de nombres primers, $B$.
3. Esborra de la llista *llista1* el primer nombre i els seus múltiples.
4. Si el primer nombre de la llista *llista1* és més petit que l'arrel quadrada de $N$, torna al punt 2.
5. Els nombres de la llista *llista2* i els que queden a la llista *llista1* són tots els nombres primers cercats.

Amb aquest algorisme podem trobar nombres primers petits, però no és gaire útil si estem interessants a trobar nombres primers molt grans a causa de la seva complexitat. En aquest cas, un altre resultat força antic, el teorema dels nombres primers de **Lagrange**, ens pots ajudar. Aquest teorema ens assegura que la probabilitat que un nombre de $n$ bits sigui primer és aproximadament:

$$\frac{1}{ln2^2} \approx \frac{1.44}{n}$$

Aquest algorisme ens dona una via, per tant, per trobar nombres primers fent una cerca aleatòria: si generem al voltant de 1000 nombres aleatoris de 1000 bits tenim una certa garantia que algun d'ells sigui primer.

**Nombres aleatoris**
En Python tenim diversos mòduls que generen nombres (pseudo)aleatoris. Aquí podem fer servir el mòdul `random`.

---

**1. Primer 10001.** Si llistem els primers 6 nombres primers: 2, 3, 5, 7, 11 i 13, podem veure que el 6è primer és el 13. Quin és el primer que ocupa la posició 10001?

Fes servir aquesta plantilla per escriure el teu codi:

```
from math import sqrt

def eratostenes(n):
    # El teu codi
    return llista_primers

def primer(n):
    # El teu codi
    return enessim_primer
```

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

- Quants nombres necessitarem per garantir que hi hagi 10001 primers?
- Explica els passos que fa l'algorisme en el cas `primer(6)`.

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

```
assert primer(6) == 13
assert primer(10001) == 104743
```

2. **Factor primer més gran**. Escriu una funció que calculi el factor primer més gran d'un nombre natural donat.
Per exemple: els factors primers de 13195 són 5, 7, 13 i 29. Per tant, 29 és el seu factor primer més gran (i el que la funció ha de retornar).

Fes servir aquesta plantilla per escriure el teu codi:

```
def factor_primer_mes_gran(n):
    # El teu codi
    return factor
```

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

- Explica els passos que fa l'algorisme en el cas `factor_primer_mes_gran(13195)`.
- Quins casos hem de tenir en compte?
- Quina complexitat té aquesta funció?

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

```
assert factor_primer_mes_gran(600851475143) ==
    ↪ 6857
```

## 3.9 Problema

### Teorema de Fermat

Suposem que volem verificar si un gran nombre és primer, evitant la via de la factorització. La base és un teorema de l'any 1640, el teorema petit de **Fermat**:

**Teorema (Fermat):** Si $p$ és un nombre primer, llavors per a qualsevol enter $a$ tal que $1 <= a < p$, es compleix que $a^{(p-1)} \% p = 1$.

Per tant, podem afirmar que si 41651 és primer, llavors $12^{(41651-1)}\%41651 = 1$.

Això ens suggereix un test de **necessitat**, però no de **suficiència**, per comprovar si un nombre és primer.

Considerem aquest algorisme:

```python
from random import randint

def fermat(nre, k):
    if nre == 1:
        return False
    for x in range(k):
        # genera un nombre aleatori
        val = randint(1, nre-1)
        # la potència a Python es basa en l'algorisme
        # de l'exponenciació modular
        # l'expressió pow equival a val**(nre-1) % nre
        if pow(val, nre-1, nre) != 1:
            return False
    return True

fermat(41651,10)
> True
```

Aquest algorisme genera uns quants nombres aleatoris menors que *nre* (en concret, $k$ nombres) i amb cadascun prova la igualtat del teorema petit de Fermat. Si algun dels nombres no la compleix, vol dir que podem estar segurs que *nre* no és primer.

Però cal tenir en compte que només ha provat uns quants valors: encara que no hagi trobat cap nombre que falli, no garanteix que no existeixi algun nombre pel qual falli. Si volguéssim estar-ne segurs del tot caldria fer la prova amb **tots** els nombres menors al nombre donat.[1]

Per tant, la pregunta que ens podem fer és: tot i no provar tots els nombres, quina seguretat puc tenir si en provo uns quants?

De nou, les matemàtiques ens poden ajudar. Es pot demostrar que si $N$ és un nombre compost llavors com a mínim en la meitat dels casos en què $a < N$ el teorema petit de Fermat fallarà i, per tant, $a^{(N-1)}$ no serà congruent amb 1 mòdul $N$. L'expressió `pow(val, nre-1, nre) != 1` retornarà:

- *True* en tots els casos si $N$ és primer.
- *True* per a la meitat o menys dels casos si $N$ no és primer.

Per tant, amb $k$ nombres escollits aleatòriament, la probabilitat que retorni *True* per a un $N$ no primer és menor que $1/(2^k)$.

Per exemple, si $k = 100$, la probabilitat és menor que $2^{-100}$. Amb un nombre moderat de tests podem determinar si un nombre és primer amb força seguretat!

Podem doncs generar nombres primers amb un algorisme força simple:

[1] Per simplificar s'obvia l'explicació dels nombres de Carmichael, que no compleixen l'algorisme petit de Fermat.

```
def generate_prime(n):
    found_prime = False
    while not found_prime:
        # genera un nombre aleatori de n bits
        p = randint(2**(n-1),2**n)
        if fermat(p,20):
            return p

generate_prime(1024)
> 15641643696359282904940897038528762712259931
34071925206793505472254225987111974690667213
58574751282743844272561807622558840439342350
44581033454641974581500307350891912817610402
15875177815103416345918968223130669630077740
07176931196725045459681402843205694434296976
16217614149253615600012129753218239546176689
3
```

1. **Primer de Weiferich**. Un primer de Wieferich és un nombre primer $p$ tal que $p^2$ divideix $2^{p-1}-1$. Quins primers de Wieferich hi ha menors que 5000?

   Fes servir aquesta plantilla per escriure el teu codi:

   ```
   def primers_Wieferich(n = 5000):
       # El teu codi
       return wieferichs
   ```

   **Reflexions prèvies:** Fes aquestes reflexions abans de començar a programar:
   - Hi ha algun algorisme dels que s'han vist a l'assignatura que et pugui resultar útil? Revisa altres problemes fets també. Descompon la funció si és necessari.

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

   ```
   assert primers_Wieferich(n = 5000) == [1093,
   ↪  3511]
   ```

2. **Test de primeritat**. Escriu una funció $factorp$ que comprovi si un determinat nombre $n$ és primer mitjançant la tècnica de la factorització i que imprimeixi quant temps ha trigat a calcular-ho (pots usar aquest mètode). Escriu una segona funció, $fermatp$, que comprovi si un determinat nombre $n > 10$ és primer mitjançant la tècnica de Fermat fent la comprovació simplement amb els nombres $a = 2, 3, 5$ i que imprimeixi quant temps ha trigat a calcular-ho.
   Fes servir aquesta plantilla per escriure el teu codi:

   ```
   import math
   ```

```
def factorp(N):
    # El teu codi
    return es_primer

def fermatp(N, a = [2, 3, 5]):
    # El teu codi
    return es_primer
```

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

- Raona el cost de les dues opcions.

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

```
assert factorp(12) == False
assert fermatp(12) == False
```

## 3.10 Problema

### Restriccions múltiples o operacions amb condicions

Molts problemes requereixen trobar una solució que compleixi diverses restriccions. Per resoldre'ls podem abordar cadascuna de les restriccions per separat i després buscar la manera ideal d'unir-les per fer les operacions mínimes.

Unes altres vegades establirem una única condició aplicada sobre un càlcul previ. En aquest cas caldrà mirar com optimitzem els càlculs a fer per evitar fer-ne de més.

---

**1. Tres condicions**. Fes una funció que trobi el nombre $n$ més gran tal que compleixi les tres condicions següents:

1. $n < 10000$,
2. $n = m \times m$ per a algun $m$,
3. $n = k! + 1$ per a algun $k$

Fes servir aquesta plantilla per escriure el teu codi:

```
import math
def nombre_3_condicions():
    # El teu codi
    return buscat
```

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

- Planteja l'algorisme per a cadascuna de les condicions per separat de manera eficient. Revisa el significat de factorial si cal.
- Mira si pots ajuntar-les de manera eficient, perquè una condició ajudi a reduir el cost d'alguna altra.

**Banc de proves:** Comprova la teva solució amb el codi següent:

```
assert nombre_3_condicions() == 5041
```

**2. Setmanes i segons**. Fes una funció que trobi tots els nombres de setmanes $n$ que tinguin un nombre de segons $s$ tal que $s = k!$ per a algun $k < 100$. Pots fer servir la funció `math.factorial(x)`, que calcula el factorial de x.

```
import math
math.factorial(4)
> 24
```

Fes servir aquesta plantilla per escriure el teu codi:

```
import math

def setmanes_segons():
    # El teu codi
    return llista_setmanes
```

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

- Quants segons té una setmana?
- Com escriuries la solució amb *list comprehensions*?

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució. Executa la comprovació següent:

```
assert setmanes_segons() == [6, 66, 792]
```

**3. Quadrats que sumen $n$.** Donat un nombre $n$, troba dos nombres positius tals que la suma dels seus quadrats sigui $n$, o si no n'hi ha que retorni $(-1, -1)$.
Fes servir aquesta plantilla per escriure el teu codi:

```
import math
def quadrats(n):
    # El teu codi
    return(nre1, nre2)
```

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

- Quin valor màxim tindran els nombres positius trobats?
- Explica els passos que fa l'algorisme en el cas `quadrats(125)`.

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

```
assert quadrats(125) == (2,11)
assert quadrats(106) == (5,9)
assert quadrats(81) == (-1,-1)
```

**4. Suma de múltiples de 3 i 5**. Els nombres naturals menors que 10 que són múltiples de 3 o 5 són 3, 5, 6 i 9. La suma d'aquests múltiples és 23. Calcula la suma de tots els naturals que són múltiples de 3 o 5 i que són menors que 1000.
Pots usar l'operació `sum(llista)` que suma tots els elements d'una llista; per exemple: `sum([3,7,1])` donarà 11.
Fes servir aquesta plantilla per escriure el teu codi:

```
def multiples():
    # El teu codi
    return resultat
```

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

- Com pots crear els múltiples de 3 o 5 i evitar repeticions?
- Hi ha alguna estructura que usem a les iteracions que et sigui útil en aquest problema?
- Quina complexitat té aquesta funció?

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

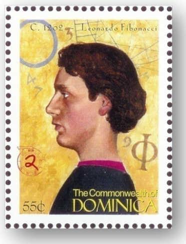
```
assert multiples() == 233168
```

## 3.11 Problema

### La seqüència de Fibonacci

Leonardo Fibonacci (1175 - c. 1250) és avui conegut sobretot per la seva seqüència o sèrie:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34...$$



Segell amb el bust de Leonardo Fibonacci

**Definició (sèrie de Fibonacci):** Siguin $F_0 = 0$ i $F_1 = 1$. Llavors el terme enèsim de la seqüència de Fibonacci es defineix com $F_n = F_{n-1} + F_{n-2}$.

Això encara **no és un algorisme**; és només una definició, atès que hi ha moltes maneres, o algorismes, per implementar aquesta sèrie. Cal observar que la seqüència creix molt ràpid: es pot demostrar (matemàticament) que el terme enèsim de la seqüència és aproximadament:

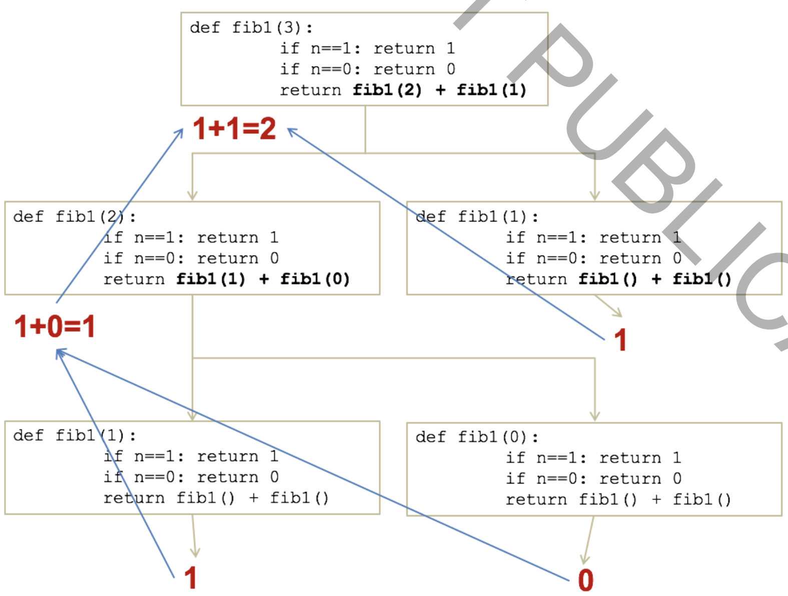
$$F_n \approx 2^{0.694n}$$

Però per calcular **exactament** un terme concret necessitem una fórmula o un algorisme. Una primera possibilitat és mitjançant un algorisme recursiu:

```
def fib1(n):
    if n==0:
        return n
    if n==1:
        return n
    else:
        return fib1(n-1) + fib1(n-2)

fib1(10)
> 55
```

Un algorisme recursiu és un algorisme que es crida a si mateix. Els algorismes recursius, per executar-se, creen automàticament còpies d'ells mateixos (amb paràmetres possiblement diferents) i creen un arbre. Quan la recursivitat s'acaba, reconstrueixen la solució movent-se cap enrere per l'arbre.




Fibonacci

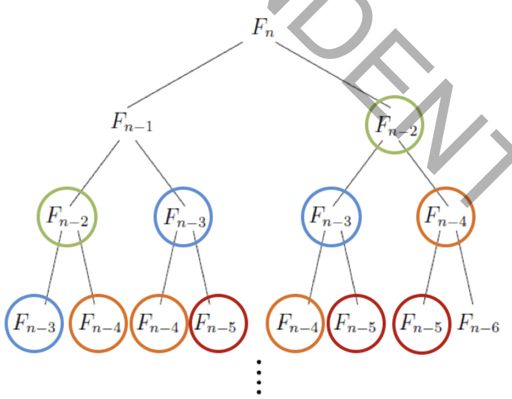
De la mateixa manera que faríem amb qualsevol algorisme, podem fer tres preguntes (**les tres preguntes bàsiques de l'algorísmica**) sobre l'algorisme que hem escrit:

1. És correcte?
2. Quant trigarà? En aquest cas, té sentit preguntar-ho en funció de $n$, la mida del nombre que passem com a paràmetre.
3. Hi ha alguna manera millor de fer-ho?

I les respostes són:

1. **És correcte?** En aquest cas és evident que sí, atès que segueix exactament la definició!
2. **Quant trigarà?** Es pot demostrar que el nombre de passos computacionals que fa és de l'ordre de $F_n$. Per calcular el terme 200 hauria de fer entorn de $2^{138}$ passos. A l'ordinador més ràpid del món, que pot executar al voltant de $4 \times 10^{13}$ passos per segon, necessitaríem més temps que el necessari per al col·lapse del Sol. A la velocitat que els ordinadors augmenten la seva capacitat de càlcul, cada any que passa podríem calcular un nombre de Fibonacci més que l'any anterior!
3. **Hi ha alguna millor manera de fer-ho?** Sí...

Per trobar una manera millor, només cal adonar-se per què és tan lent:




Hi ha molts càlculs (en aquest cas, crides recursives) que es repeteixen! Una possible solució és guardar el resultat de cada crida el primer cop que ho calculem i no tornar a calcular-ho. Tot seguit en farem una versió **iterativa**, basada en **llistes**:

```
def fib2(n):
    if n==0:
        return 0
    ls = [0,1]
    for i in range(2,n+1):
        ls.append(ls[i-1]+ls[i-2])
    return ls[n]
```

Aquest algorisme és correcte perquè implementa directament la definició i executarà $(n-1)$ vegades la iteració. Ara podem calcular fins i tot `fib(100.000.000)`.

Recorda: Amb l'eina Code Skulptor podem visualitzar fàcilment el funcionament de qualsevol algorisme.

L'algorisme `fib2(n)` és **lineal** (o polinòmic) respecte de $n$. Però l'algorisme encara es pot millorar...

```
def fib3(n):
    a,b = 0,1
    for i in range(1,n+1):
        a,b = b, a+b
    return a

fib3(10)
> 55
```



---

1. **Suma de termes parells de Fibonacci**. Calcula la suma dels termes parells (`fib(n) % 2 = 0`) menors de quatre milions de la funció de Fibonacci. Quant temps triga?
Fes servir aquesta plantilla per escriure el teu codi:

```
def fibonacci_termes_parells(n = 4000000):
    # El teu codi
    return suma

# Al principi d'una nova cel·la executa la comanda per saber el
↪  temps que triga
%timeit fibonacci_termes_parells(4000000)
```

**Reflexions prèvies:** Fes aquestes reflexions abans de començar a programar:
- Revisa l'algorisme de Fibonacci de teoria i implementa'n la versió més eficient.
- Compara el temps en el teu ordinador amb el d'una companya o company.

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució.

```
assert fibonacci_termes_parells(4000000) == 4613732
```

2. **Suma de múltiples de 3 de Fibonacci**. Fes una funció que sumi els $n$ primers termes de Fibonacci que siguin múltiples de 3. És a dir, d'aquells termes de Fibonacci que es poden dividir per 3 de manera exacta.
Fes servir aquesta plantilla per escriure el teu codi:

```
def fibonacci_multiples_3(n):
    # El teu codi
    return suma
```

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

- Com sabem si un nombre és divisible per un altre?
- Revisa l'algorisme de Fibonacci vist anteriorment i implementa'n la versió més eficient.

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

```
assert fibonacci_multiples_3(10) == 119814915
```

**3. Els termes de Fibonacci i la multiplicació de matrius**. El terme enèsim de la sèrie de Fibonacci, $F_n$, es pot obtenir aplicant càlculs matricials. És fàcil veure que:

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

Fes una funció que retorni els $n$ primers termes de la sèrie de Fibonacci calculats d'aquesta manera, amb la menor complexitat possible. Codifica en una funció a part la multiplicació de dues matrius sense fer servir cap funció avançada de Python.
Fes servir aquesta plantilla per escriure el teu codi:

```
def fibonacci_matricial(N):
    # El teu codi
    return sequencia
```

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

- Quin és el mínim nombre d'operacions per multiplicar dues matrius de $n$ elements?
- Donada una matriu $A$, es pot calcular $A^n$ com $A \times A \times A \times \cdots \times A$. Creus que és la manera més eficient? Hi ha alguna alternativa més eficient?

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

```
assert fibonacci_matricial(10) == [0, 1, 1, 2, 3,
↪  5, 8, 13, 21, 34]
```

# Capítol 4

# Algorismes i text

De la mateixa forma que les operacions aritmètiques havien d'estar implementades de manera molt eficient en els primers ordinadors, les operacions amb cadenes de caràcters han d'estar implementades amb bons algorismes si es volen aplicar a grans volums de dades.

A Python, i a la majoria de llenguatges, les cadenes de caràcters són **immutables** i no es poden modificar amb cap tipus d'operador. Per tant, els algorismes de text, per ser eficients, han de minimitzar les còpies de cadenes de caràcters.

## 4.1 Problema

### Acrònims

Un **acrònim** és una paraula formada per les primeres lletres, en majúscules, de totes les paraules d'una expressió o frase (per exemple, l'acrònim **RAM** correspon a l'expressió ***random access memory***).

**1.** Escriu una funció, `acro`, que a partir d'una frase sense punts, comes, accents, números ni cap signe de puntuació, imprimeixi l'acrònim corresponent.
Fes servir el patró següent:

```
def acronim(frase):
    # El teu codi
    return acro
```

> **Reflexions prèvies:** Fes aquestes reflexions abans de començar a programar:
> - Recorda que els acrònims, independentment de com l'usuari hagi entrat l'expressió, sempre són en majúscules.
> - Revisa les funcions de les cadenes i mira si n'hi ha alguna que et pugui ser útil. Com ho podem fer per separar les paraules? I per transformar una cadena en majúscules?
> - Explica els passos que fa l'algorisme en el cas acro("Hola que tal").

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució. Entre d'altres, executa la comprovació següent:

```
assert acronim("Hola que tal") == 'HQT'
```

## 4.2 Problema

### Traducció a l'alfabet d'aviació

En el món de l'aviació, els pilots i controladors de trànsit aeri utilitzen una convenció per comunicar-se entre ells. Així, usen l'**alfabet d'aviació**, que utilitza 26 lletres. Cada lletra té una paraula associada i amb ella es comuniquen evitant confusions.

L'alfabet d'aviació és el següent:

| A | B | C | D | E |
|---|---|---|---|---|
| Alpha | Bravo | Charlie | Delta | Echo |
| F | G | H | I | J |
| Foxtrot | Golf | Hotel | India | Juliet |
| K | L | M | N | O |
| Kilo | Lima | Mike | November | Oscar |
| P | Q | R | S | T |
| Papa | Quebec | Romeo | Sierra | Tango |
| U | V | W | X | Y |
| Uniform | Victor | Whiskey | Xray | Yankee |
| Z | | | | |
| Zulu | | | | |

Un controlador de trànsit aeri lletrejaria la paraula *PYTHON* així: Papa Yankee Tango Hotel Oscar November.

**1.** Escriu una funció, `aviacio`, que, usant un diccionari, converteixi una cadena de lletres a l'alfabet d'aviació, generant una llista de sortida.

Fes servir el patró següent:

```
def aviacio(cadena):
    # El teu codi
    return traduccio
```

**Reflexions prèvies:** Fes-te aquestes preguntes abans de començar a programar:

- Has tingut en compte que la paraula d'entrada pot estar en minúscules, majúscules o amb una lletra de cada?
- Quin cost tindrà l'algorisme?

> **Banc de proves:** Defineix diversos exemples per comprovar la teva solució. Entre d'altres, executa les comprovacions següents:
>
> ```
> assert aviacio("Arc") == ['Alpha', 'Romeo',
>  ↪  'Charlie']
> ```

## 4.3 Problema

### Cadenes isomorfes

Diem que dues cadenes de caràcters $a$ i $b$ són **isomorfes** si les ocurrències de cada caràcter de la cadena $a$ es poden reemplaçar per un caràcter de manera que la convertim en la cadena $b$ i viceversa.

Per exemple, les dues cadenes:

```
a = "TEULADES"
b = "PISCINES"
```

són isomorfes perquè podem establir aquestes correspondències:

$$T \leftrightarrow P,\ E \leftrightarrow I,\ U \leftrightarrow S,\ L \leftrightarrow C,\ D \leftrightarrow N$$

Cal tenir en compte que un caràcter pot tenir una correspondència amb si mateix, però dos caràcters diferents no poden correspondre a un mateix caràcter.

1. Escriu una funció anomenada `isomorf` tal que, donades dues cadenes de caràcters d'entrada, retorni `True` si són isomorfes i `False` en cas contrari.

   Per resoldre aquest problema et pot ser útil la funció `zip` de Python. Aquesta funció permet agafar un element de cada objecte d'entrada (llistes, cadenes de caràcters). D'aquesta manera podem fer-ne parelles molt ràpidament. Per exemple, aquest programa:

   ```
   for i, j in zip('hola', 'adeu'):
       print(i, j)
   ```

   imprimeix les parelles formades per les parelles de lletres de les paraules d'entrada

   ```
   h a
   o d
   l e
   a u
   ```

   Fes servir el patró següent:

   ```
   def isomorf(cadX, cadY):
       # El teu codi
       return es_isomorf
   ```

**Reflexions prèvies:** Abans de programar pensa en exemples simples que posin a prova l'algorisme. Pensa també en els punts següents:

- Quins passos fa l'algorisme en els casos que has pensat?
- Quines col·leccions usaries? Per què?
- Quins casos extrems hem de tenir en compte? Per exemple, què passa si les dues paraules tenen longituds diferents? I si una de les cadenes de caràcters és buida, `""`?
- Quantes operacions fa el programa `isomorf("abc","def")`?

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

```
assert isomorf('CBAABC', 'DEFFED') == True
assert isomorf('XXX', 'YYY') == True
assert isomorf('RAMBUNCTIOUSLY',
               'THERMODYNAMICS') == True
assert isomorf('XXY', 'XYY') == False
assert isomorf('ABAB', 'CD') == False
```

**2.** Com a ampliació de l'exercici anterior, escriu una funció que rebi com a entrada una llista i una paraula. Aquesta funció ha de retornar totes les paraules de la llista que siguin isomorfes a la paraula d'entrada.
Fes servir el patró següent:

```
def tria_paraules_isomorfes(llista, paraula):
    # El teu codi
    return llistaIsomorfes
```

**Reflexions prèvies:** Abans de programar fes aquestes reflexions:

- No ho tornis a programar tot, pensa si té sentit cridar `isomorf` dins la nova funció. Pots fer alguna comprovació prèvia que t'estalviï feina?
- Si el cost de cadenes isomorfes és X, quin és el cost de `tria_paraules_isomorfes`?
- Explica els passos que fa l'algorisme en el cas `tria_paraules_isomorfes(['mim', 'gat', 'gos'], 'rar')`.

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

```
assert tria_paraules_isomorfes(['gag', 'sos', 'mim',
↪  'gat', 'gos'], 'rar') == ['gag', 'sos', 'mim']
assert tria_paraules_isomorfes(['gag', 'sos', 'mim',
↪  'gat', 'gos'], 'rap') == ['gat', 'gos']
```

## 4.4 Problema

### Totes les subcadenes

1. Escriu una funció que retorni una llista amb totes les subcadenes d'una cadena donada. Fes servir *list comprehensions*.
   Per exemple: `totes_subcadenes("abcd")` hauria de retornar la llista `['a', 'ab', 'abc', 'abcd', 'b', 'bc', 'bcd', 'c', 'cd', 'd']`
   Fes servir aquesta plantilla per escriure el teu codi:

```
def totes_subcadenes(cadena):
    # El teu codi
    return subcadenes
```

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

- Quines són totes les subcadenes de la paraula *casa*?
- Observa la solució de l'exemple i fixa't en els índexs de cada subcadena. Pots identificar un patró? Quins recorreguts haurem de fer per crear les subcadenes?
- És més senzill si primer escrius la funció amb iteracions i després la converteixes a *list comprehensions*.
- Explica els passos que fa l'algorisme en el cas `totes_subcadenes("abcd")` i calcula'n la complexitat.

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució, i fes també la comprovació següent:

```
assert len(totes_subcadenes('abcd')) ==
↪   int((len('abcd') *(len('abcd') + 1)/2))
assert totes_subcadenes('abcd') == ['a', 'ab', 'abc',
↪   'abcd', 'b', 'bc', 'bcd', 'c', 'cd', 'd']
```

## 4.5 Problema

### Levenshtein

Una seqüència genètica és un una cadena de caràcters (*string*) formada per caràcters d'un alfabet de quatre lletres: `A, T, G, C`, anomenades **bases**, que corresponen a les macromolècules de l'**ADN**. Un **gen** és una seqüència ordenada de bases i el **genoma** és la concatenació de tots els gens.

Cada cèl·lula produïda pel cos rep una còpia del genoma però sovint aquesta còpia és alterada. Les possibles alteracions que es poden produir són, entre d'altres, la substitució d'una base per una altra o la pèrdua d'una base.

**Algorisme de Levenshtein**
El càlcul de la distància d'un patró al *substring* més semblant d'un text es pot fer amb l'algorisme de Levenshtein. L'única diferència és que s'ha d'inicialitzar la primera fila amb zeros i que la distància d'edició serà el valor mínim de l'última fila de la matriu de costos. També has de tenir en compte els costos en la inicialització de la primera columna.

1. Fes una funció, anomenada `dna`, basada en l'algorisme de Levenshtein, que busqui dins d'una seqüència genètica una cadena genètica passada per paràmetre.
   Aquesta funció ha de retornar la línia del fitxer on comença la cadena més semblant i la distància entre la cadena d'entrada i la cadena més semblant.

La seqüència genètica que farem servir és la del cromosoma 2 humà. Seleccionant el vincle, podeu accedir al fitxer.
Les primeres línies d'aquest fitxer tenen aquesta forma:

```
CCCATCTCTTTCTCATTCCTTGGTTGAGAACACGAACTTCAGGACTTGCCTCACACTAGGGCCCATTCTT
TGTTTCCCAGAAAGAAGAGGCTCTCCACACAGAGTCCCATGTACACCAGGCTGTCAACAAACATGAATTG
AATGAAGGAGTGGATGGTTGGGTGGAAGTGATTTAAGAAATCCTAACTGGGGAATTTCACTGGAAACTTA
```

En programar aquesta funció, cal que tinguis en compte que, en aplicacions bioinformàtiques, els costos de les operacions d'edició són lleugerament diferents de les que hem vist fins ara:

- Per a un salt o inserció (al patró o al text) el cost és 2.
- Per a una substitució el cost és 1.
- Quan hi ha correspondència el cost és 0.

Adapta la teva funció anterior, `dna`, a aquests costos. La funció ha de rebre el patró que volem buscar i ha de retornar dos valors: la línia del fitxer on comença la cadena més semblant al patró i la distància mínima entre aquesta cadena i el patró.
Fes servir aquesta plantilla per escriure el teu codi:

```python
def levenshtein(patro, text, dlt = 2, insr = 2, subs = 1):
    # El teu codi
    return distancia_minima

def dna(patro, fitxer = 'HUMAN-DNA.txt'):
    # El teu codi
    return linia, distancia_final
```

**Banc de proves:** Per comprovar la teva funció, pots usar aquestes instruccions:

```python
assert dna('AGATACATTAGACAATAGAGATGTGGTC') == (32,
↪   11)
assert dna('GTCAGTCTGGCCTTGCCATTGGTGCCACCA') == (352,
↪   11)
assert dna('TACCGAGAAGCTGGATTACAGCATGTACCATCAT') ==
↪   (233, 13)
```

## 4.6 Problema

### Run Length Encoding

***Run Length Encoding*** (RLE) és un algorisme de compressió de dades sense pèrdua que agrupa els valors repetits amb el nombre de vegades que es repeteixen per optimitzar la memòria.

Per exemple, suposem que tenim aquesta cadena:

```
BBBBBBBBNBBBBBBBBBBBBBNNNBBBBBBBBBBBBNBBBBBBBBB
```

Aquesta cadena es pot comprimir en la cadena `B7N1B13N3B12N1B9` i s'interpreta de la manera següent: 7 caràcters blancs, 1 de negre, 13 de blancs, 3 de negres, 12 de blancs, 1 de negre i 9 de blancs. D'aquesta manera podem reconstruir la cadena original sense pèrdua d'informació.

**1.** Escriu una funció, `rle`, que donat un text de lletres ASCII (A-Z), el codifiqui usant l'algorisme RLE.

Fes servir el patró següent:

```
def rle(text):
    # El teu codi
    return text_codificat
```

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

- En el millor dels casos, si tenim una cadena de $n$ caràcters, quina és la mida de la cadena resultant?
- Quin seria el pitjor dels casos? En aquest cas, té sentit aquest tipus de compressió?
- Si tenim una cadena de $n$ caràcters, quantes operacions bàsiques cal fer per comprimir-lo?

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

```
assert rle('ABBBBNNNEEEDDDZZAAAAA') ==
↪ 'A2B4N3E3D3Z2A5'
assert rle('BBBBBBBBBBBBBBBBBBWWWWWWZAAA') ==
↪ 'B18W6Z1A3'
```

## 4.7 Problema

### Subcadena més llarga sense cap caràcter repetit

**1.** Escriu una funció anomenada `subcadena_mes_llarga` que donada una cadena de caràcters, identifiqui la subcadena més llarga que no contingui cap caràcter repetit.

Per exemple, per a la cadena `'lacadenamesllarga'`, la subcadena més llarga sense cap caràcter repetit és `'namesl'`.

Fes servir el patró següent:

```
def subcadena_mes_llarga(cadena):
    # El teu codi
```

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

- Quines col·leccions et poden ser útils? Raona la resposta.
- Quina és la complexitat d'aquest algorisme? Mostra els càlculs fets. (Nota: idealment hauries d'aconseguir un algorisme de complexitat $O(n)$.)
- Comprova manualment el resultat de la funció per a la paraula `'algorismica'`. El teu programa retorna el mateix?

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

```
assert subcadena_mes_llarga('lacadenamesllarga') ==
↪  'namesl'
assert subcadena_mes_llarga('mesllarga') == 'mesl'
assert subcadena_mes_llarga('aaa') ==  'a'
```

## 4.8 Problema

### Subseqüència en comú més llarga

**1.** Escriu una funció **basada en força bruta** tal que, donades dues cadenes de caràcters, identifiqui la longitud de la subseqüència compartida més llarga. En aquest problema definim una subseqüència com una seqüència de caràcters en el mateix ordre que a la cadena original però **no necessàriament consecutiva**. Per exemple de la paraula `ACBA`, podem treure'n les subseqüències `AC, AB, AA, ACB, ACA, ABA, ACBA, CB, CA, CBA, BA, A, C, B`, i de la paraula `BRA`, podem treure'n les subseqüències `B, R, A, BR, BA, RA, BRA`, i la subseqüència comuna més llarga entre les dues seria `BA`, amb 2 caràcters. Pots fer servir alguna funció del mòdul `itertools`.
Fes servir aquesta plantilla per escriure el teu codi:

```
import itertools

def subsequencia_comuna_v1(paraula1, paraula2):
    # El teu codi
    return nre_caracters_comuns
```

**Banc de proves:**

```
assert subsequencia_comuna_v1('STUTVST', 'TVUSTS') ==
↪  4
```

**2.** Si ens fixem en l'algorisme, veurem que sovint es repeteixen càlculs. El que podem fer és usar un diccionari que vagi guardant les solucions parcials ja calculades i cada cop que n'hàgim de calcular una, mirar el diccionari per veure si ja s'ha calculat.

Quan hàgim de calcular una solució parcial. si ja hi és, mirem el resultat en comptes de calcular-la de nou; altrament, la calculem i l'emmagatzemem al diccionari.
Fes servir aquesta plantilla per escriure el teu codi:

```
def subsequencia_comuna_v2(paraula1, paraula2):
    # El teu codi
    return nre_caracters_comuns
```

**Banc de proves:**

```
assert subsequencia_comuna_v2('STUTVST', 'TVUSTS') ==
↪ 4
```

**3.** També podem construir la solució usant una taula que va guardant els valors de la subseqüència comuna més llarga.
Per exemple, per a les paraules `SUBCADENA` i `ABECEDARI` la taula seria la següent:

| | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | S | U | B | C | A | D | E | N | A |
| | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | A | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | B | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | E | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 |
| 3 | C | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| 4 | E | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 3 | 3 |
| 5 | D | 0 | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 |
| 6 | A | 0 | 0 | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 4 |
| 7 | R | 0 | 0 | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 4 |
| 8 | I | 0 | 0 | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 4 |

La casella `[i][j]` guarda el valor de longitud de la subseqüència més llarga entre `cadena1[0:j]` i `cadena2[0:i]`.
Per exemple, la casella `[2][6]` guarda la longitud de la subseqüència més llarga entre `cadena1[0:6]` i `cadena2[0:2]`. Efectivament 2 és la longitud de la subseqüència més llarga entre `SUBCADE` i `ABE`, que es correspon a `AE`.
La solució del problema es basa en aquesta observació: per omplir la posició $(i, j)$ de la matriu només cal saber els valors $(i-1, j-1), (i.j-1), (i-1, j)$ i aplicar una regla simple:

```
if paraula1[j-1] == paraula2[i-1]:
    taula[i][j] = taula[i-1][j-1]+1
else:
    taula[i][j] = max(taula[i-1][j],  taula[i][j-1])
```

Fes servir el patró següent:

```
def subsequencia_comuna_v3(cadena1, cadena2):
    # El teu codi
    return nre_caracters_comuns
```

**Banc de proves:**

```
assert subsequencia_comuna_v3('STUTVST', 'TVUSTS') ==
↪ 4
```

# Capítol 5

# Dividir i vèncer

**Dividir i vèncer** és una estratègia de resolució de problemes que consisteix a:

- Dividir un problema en subproblemes que són instàncies més petites (des del punt de vista de la mida de l'entrada) del mateix problema.
- Resoldre recursivament aquests subproblemes.
- Combinar adequadament les solucions dels subproblemes per trobar la solució del problema original.

Les **qüestions** que ens hem de plantejar abans de programar un algorisme seguint aquesta estratègia, són tres:

1. Com anem dividint el problema en subproblemes de manera recursiva?
2. Com aturem la recursió i donem una solució al darrer subproblema?
3. Com combinem les solucions recursives per assolir la solució del problema complet?

L'esquema general d'aquests algorismes és en molts casos el següent:

1. Inicialment, tenim un problema de mida $n$.
2. Reformulem el problema mitjançant la solució d'$a$ problemes de mida $n/b$.
3. Combinem les respostes en un temps $O(n^d)$.

És fàcil deduir que en aquest cas, la complexitat d'aquest esquema té aquesta forma:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

Aquest tipus de recurrència té una **solució tancada**, que està enunciada al **teorema mestre** (*master theorem*). És a dir, podem saber la complexitat dels algorismes recursius coneixent el valor dels paràmetres $a, b$ i $d$.

**Teorema (mestre):** Donada l'expressió

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

amb constants $a > 0$, $b > 1$, i $d >= 0$, la complexitat d'un algorisme recursiu amb aquest tipus de recurrència pot ser de tres tipus:

1. $T(n) = O(n^d)$, si $d > \log_b(a)$
2. $T(n) = O(n^d log(n))$, si $d = \log_b(a)$
3. $T(n) = O(n^{log_b(a)})$, si $d < \log_b(a)$

Observa que, en el cas que $d$ sigui un valor elevat, aquest cost (el de recombinar les solucions) domina la resta d'operacions. En el tercer cas, en canvi, el cost dominant és $\log_b a$ que serà un valor gran quan tinguem molts subproblemes i no es redueixin gaire en cada divisió. En el segon cas, les dues complexitats estan equilibrades.

Per exemple, suposem que tenim un problema amb les característiques següents:

- El problema es pot dividir en dos subproblemes ($a = 2$).
- Cadascun dels problemes processa la meitat de les dades originals, és a dir, el problema es divideix en dos problemes iguals ($b = 2$).
- Suposem que que la reconstrucció es pot fer en temps quadràtic $O(n^2)$ ($d = 2$).

Com que ens trobem en el cas 1, ja que $2 > log_2(2)$, segons el teorema mestre la complexitat d'aquest algorisme és $O(n^2)$.

## 5.1 Problema

### Suma d'una llista

**1.** Escriu una funció recursiva que sumi tots els elements d'una llista. Digues quin és el cas base de l'algorisme i calcula'n la seva complexitat.
Fes servir el patró següent:

```python
def suma_llista(llista):
    # El teu codi
    return suma
```

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

- Tot i que es pot enfocar el problema de diferents maneres, pensa com si ho estiguessis fent amb un bucle `for`, és a dir, agafant un element cada cop.
- Amb quins paràmetres hem de tornar a cridar la funció? Recorda que el problema ha de fer-se cada cop més senzill fins a arribar al cas base.

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

```
assert suma_llista([1,2,3,4]) == 10
assert suma_llista([-1,-2,0,1,2]) == 0
```

## 5.2 Problema

### Nombre no repetit

**1.** Donada una llista de nombres en què cada nombre apareix dues vegades consecutives (una darrera l'altra) i només hi ha un element que surt una sola vegada, escriu una funció que trobi aquest valor desaparellat. Pensa en una solució que resolgui el problema amb una complexitat menor a $O(n)$. Fes servir aquesta plantilla per escriure el teu codi:

```
def busca_unic (llista):
    # El teu codi
    return nre
```

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

- Et recorda algun algorisme d'ordenar i cercar? Com s'ha d'adaptar?
- Pensa exemples senzills i observa els índexs de cada element. Mira si identifiques algun patró que et pugui ajudar a resoldre el problema.

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

```
assert busca_unic([ 1, 1, 2, 4, 4, 5, 5, 6, 6 ]) == 2
```

## 5.3 Problema

### Seqüència capicua més llarga

**1.** Escriu una funció que donada una cadena trobi la llargada de la subseqüència capicua més llarga de manera òptima. Abans, però, observa la solució següent, que usa un algorisme per força bruta:

```
def subs_capicua(cadena):
    # print(cadena)
    i = 0
    j = len(cadena) - 1
    if j == 0:
        return 1
```

```
        if cadena[i] == cadena[j]:
            return subs_capicua(cadena[1:-1]) + 2
        else:
            return max(subs_capicua(cadena[1:]),
                       subs_capicua(cadena[:-1]))

subs_capicua("APXPRABARXP")
>>> 9
```

Descomenta la funció `print()` de la primera línia de la funció i observa quines cadenes avalua la funció. Com pots veure, la solució de força bruta repeteix molts dels càlculs. Implementa un diccionari per guardar les solucions calculades i no les tornis a calcular.
Fes servir el patró següent:

```
def subs_capicua_opt(cadena, solucions = {}):
    # El teu codi
    return subsequencia
```

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

- Observa que es tracta d'un algorisme recursiu. Mostra quin és el cas base i com s'avança en la recursió.
- Quina complexitat té l'algorisme que hem presentat? I quina complexitat tindrà l'algorisme millorat?
- Quina informació hem de guardar al diccionari per evitar repetir execucions?

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

```
assert subs_capicua_opt('APXPRABARXP') == 9
```

## 5.4 Problema

### El valor més petit que falta

**1.** Donada una llista ordenada d'enters no negatius (inclòs el 0), troba el valor més petit que hi falta. En cas que no hi falti cap valor, la teva funció ha de retornar -1.
Fes servir aquesta plantilla per escriure el teu codi:

```
def el_mes_petit(llista):
    # El teu codi
    return valor
```

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

- Quins índexs ocupen els enters següents a la llista `[0, 1, 2, 3, 4, 7, 12]`? Quin és l'index del valor 0? I del 4? I del 7? Quin seria el valor que hauria de retornar la funció en aquest cas? Identifiques algun patró que et pugui ajudar a resoldre el problema?
- Pots relacionar aquest problema amb un algorisme de dividir i vèncer? Quina compexitat té? Compara aquesta complexitat amb la solució per força bruta.

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

```
assert el_mes_petit([0, 1, 2, 3, 4, 7, 12]) == 5
assert el_mes_petit([1, 2, 3, 4, 7, 12]) == 0
assert el_mes_petit([0, 1, 2, 3, 4]) == -1
```

## 5.5 Problema

### Quantitat d'uns a una llista ordenada de nombres binaris

**1.** Donada una llista formada només pels nombres `0` i `1` i ordenada, escriu una funció que calculi el nombre d'uns (nombre `1`) que conté la llista.

Fes servir aquesta plantilla per escriure el teu codi:

```
def quants_1(llistaBinaria):
    # El teu codi
    return quantitat
```

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

- Podem aplicar algun dels algorismes de dividir i vèncer? Com s'ha d'adaptar?
- Hi ha algun cas de solució directa?
- Escriu una versió en la qual llegeixis les llistes d'un fitxer *input.txt*. Aquest fitxer ha de contenir les llistes de nombres, cada una en una fila diferent (utilitza les funcions `open()` i `readlines()` per llegir). El programa ha d'escriure el resultat de cada execució en un fila diferent d'un fitxer diferent, *output.txt*.

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

```
assert quants_1([0, 0, 1, 1, 1, 1, 1]) == 5
assert quants_1([0, 0, 0, 0, 1, 1]) == 2
assert quants_1([0, 0, 0, 0, 0, 0, 0]) == 0
assert quants_1([1, 1, 1]) == 3
```

## 5.6 Problema

### Negatius al davant

**1.** Escriu una funció no recursiva anomenada `negatius` tal que, donada una llista de nombres enters, modifiqui la llista per col·locar els nombres negatius al principi de la llista. Observa que aquesta funció no ha de retornar res, només modifica la llista d'entrada.
Fes servir aquesta plantilla per escriure el teu codi:

```
def negatius(llista):
    # El teu codi
```

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

- Et recorda algun dels algorismes de dividir i vèncer? Com s'ha d'adaptar en aquest cas?
- Quins canvis hauries de fer si no volguéssim modificar la llista inicial?

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

```
llista = [1, -2, 3, -4, -3, 5, 6]
negatius(llista)
llista
>>> [-3, -2, -4, 3, 1, 5, 6]
# Aquest resultat és un d'entre molts de possibles.
# L'única condició és que els negatius
# estiguin al principi de la llista.
```

## 5.7 Problema

### Zeros al final

**1.** Donada una llista d'enters, escriu una funció que desplaci tots els zeros de la llista al final. No alteris la resta d'elements de la llista. Aquesta funció no ha de retornar cap valor; ha de modificar la llista d'entrada.
Fes servir aquesta plantilla per escriure el teu codi:

```
def zeros_final(llista):
    # El teu codi
```

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

- Quin algorisme hem d'aplicar? Com l'hem d'adaptar perquè no ens canviï tots els valors?
- Modifica l'algorisme per moure tots els zeros al principi. Aquest nou algorisme té la mateixa complexitat que el primer?
- Què hauríem de fer si no volguéssim modificar la llista d'entrada?

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució. Entre d'altres, executa la comprovació següent:

```
llista = [3, 1, 0, 5, 0, 0, 2, 7, 8]
zeros_final(llista)
assert llista == [3, 1, 5, 2, 7, 8, 0, 0, 0]
```

## 5.8 Problema

### Rotacions

1. Donada una llista ordenada de nombres enters, definim una **rotació** com l'operació de moure elements del final de la llista al principi. Escriu una funció que, donada una llista, calculi quantes rotacions s'han aplicat prèviament. Podem assumir que la llista no conté cap element duplicat.
Fes servir aquesta plantilla per escriure el teu codi:

```
def rotacions(llista):
    # El teu codi
    return nre_rotacions
```

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

- Analitza l'exemple següent manualment: `rotacions([9, 10, 2, 5, 6, 8]]`. Quin és el resultat esperat de la funció?
- Calcula la complexitat de l'algorisme.

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

```
assert rotacions([9, 10, 2, 5, 6, 8]) == 2
assert rotacions([3, 5, 10, 12, 1, 2]) == 4
assert rotacions([20, 2, 3, 7, 15, 18]) == 1
```

## 5.9 Problema

### Valor igual al seu índex

**1.** Donada una llista $L$ de $n$ nombres naturals ordenats de manera creixent, crea una funció, `existeix`, que, usant l'estratègia de dividir i vèncer comprovi si existeix algun element de la llista que compleixi el següent:

$$L[i] = i, \; i \in \{0, \ldots, n-1\}$$

L'algorisme proposat ha de tenir complexitat $O(log(n))$.
Fes servir aquesta plantilla per escriure el teu codi:

```
def existeix(llista):
    # El teu codi
    return bool_existeix
```

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

- Quin algorisme hem d'aplicar? Quin cost té aquest algorisme? Com s'ha d'adaptar?
- Explica els passos que fa l'algorisme en el cas `existeix([0, 1, 2, 7])`.

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

```
assert existeix([1, 4, 5, 6, 7]) == False
assert existeix([0, 1, 2, 7]) == True
assert existeix([1, 1, 1, 3]) == True
```

## 5.10 Problema

### Comptador d'inversions en una llista

**1.** Volem ordenar una llista $A$ de nombres enters en ordre creixent, és a dir, $A[i] < A[j]$ per a qualsevol $i < j$.
Definim **inversió**, com el procés d'intercanviar dos valors de la llista que no estiguin en el seu ordre natural, és a dir, que el de l'esquerra sigui més gran que el de la dreta. Concretament, diem que $(A[i], A[j])$ és una inversió, si $A[i] > A[j]$ i $i < j$. Escriu una funció que calculi el nombre d'inversions dins una llista i que digui quines són.
Fes servir aquesta plantilla per escriure el teu codi:

```
def comptador_inversions(llista):
    # El teu codi
    return nreInversions
```

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

- Justifica la solució que han de retornar les expressions següents:

```
comptador_inversions([1, 8, 6, 4, 5])
comptador_inversions([4, 6, 1, 3, 9, 4])
comptador_inversions([1, 2])
```

  Quines són les inversions dels exemples anteriors?
- Aquest problema et recorda algun dels algorismes que hem vist de dividir i vèncer? Quin? Com cal adaptar-lo?

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

```
assert comptador_inversions([3, 1, 5, 2, 7, 8, 4]) == 6
```

Les inversions són: (3, 1), (3, 2), (5, 2), (5, 4), (7, 4) i (8, 4)

## 5.11 Problema

### Elements pic

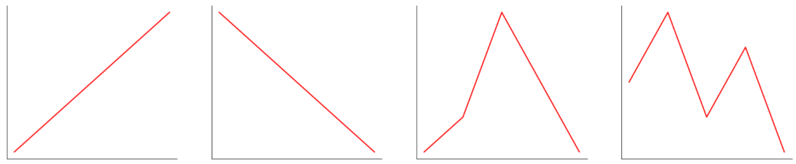
1. Donada una llista d'enters en què mai trobem dos elements consecutius del mateix valor, fes un programa que trobi, si existeix, un dels seus **elements pic**. Diem que un enter de la llista és un element pic si els seus veïns immediats són menors que ell.

   Fes servir aquesta plantilla per escriure el teu codi:

```
def elements_pic(llista):
    # El teu codi
    return pic
```

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

- Pensa en els diferents casos que ens podem trobar i raona quin seria l'element pic en cada un d'ells. Per exemple, troba un element pic per a les funcions següents:



Exemples de seqüències de valors i elements pic.

- Raona si tots els casos possibles tenen, com a mínim, un element pic.
- Imagina't que mires l'element central de la llista, on està el pic? Com ho saps? Quin serà el pas següent?
  - Pensa si hi ha algun algorisme de dividir i vèncer aplicable en aquest cas.
  - Explica els passos que fa l'algorisme en el cas `elements_pic([3, 1, 5, 2, 7, 8])`.
  - Calcula la complexitat de l'algorisme amb el teorema mestre.

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

```
# Observa que aquesta llista conté tres valors pic:
# el 3, el 5 i el 8. L'algorisme només n'ha de
# tornar un. És per això que comprovem que el
# valor retornat estigui dins la llista [3, 5, 8].
assert elements_pic([3, 1, 5, 2, 7, 8]) in [3, 5, 8]
assert elements_pic([9, 5, 2]) == 9
assert elements_pic([1, 2, 7, 8]) == 8
```

## 5.12 Problema

### Divisió d'una llista en tres parts segons un valor donat, $v$

**1.** Escriu un algorisme que a partir d'una llista $A$ i un valor $v$, modifiqui l'ordenació dels elements tal que al principi apareguin tots els elements igual a $v$, a continuació tots els elements més petits que $v$ i finalment tots els elements més grans que $v$. Qualsevol d'aquestes parts pot estar buida.
**Atenció**. No creïs cap llista nova per resoldre el problema; modifica únicament la llista original.
Fes servir aquesta plantilla per escriure el teu codi:

```python
def parts_llista(llista, valor):
    # El teu codi
```

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

- Hi ha algun dels algorismes de dividir i vèncer que s'assembli a aquest problema? En què canvia?

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució. Entre d'altres, executa la comprovació següent:

```python
llista = [6, 2, 9, 8, 7, 4, 3, 7, 1, 9, 7, 1]
parts_llista(llista, 7)
assert(llista) == [7, 7, 7, 6, 2, 4, 3, 1, 1, 9, 8, 9]
```



## 5.13 Problema

### Primera i darrera ocurrència d'un nombre donat en una llista ordenada

**1.** Donada una llista d'enters i un nombre, escriu un algorisme que retorni la posició de la primera i darrera ocurrència d'aquest nombre a la llista. Si el nombre no apareix, el programa ha de retornar -1,-1 com a índexs.
Fes servir aquesta plantilla per escriure el teu codi:

```python
def primera_darrera_ocurrencia(llista, nombre):
    # El teu codi
    return primera_ocurrencia, darrera_ocurrencia
```

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

- Tenint en compte que la llista està ordenada, podem usar algun algorisme de dividir i vèncer? Com cal adaptar-lo?
- Si sabéssim que el nombre apareix menys de 5 vegades, o si, en canvi, sabéssim que el nombre n'apareix més de 30, com podria variar la solució?

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

```python
assert primera_darrera_ocurrencia([3, 5, 5, 5, 6, 6, 8, 8, 9, 9, 9], 3) == (0,0)
assert primera_darrera_ocurrencia([3, 5, 5, 5, 6, 6, 8, 8, 9, 9, 9], 9) == (8,10)
assert primera_darrera_ocurrencia([3, 5, 5, 5, 6, 6, 8, 8, 9, 9, 9], 4) == (-1,-1)
```

## 5.14 Problema

### Parelles que sumen un valor

**1.** Donada una llista ordenada d'enters sense cap repetició, $L$, de longitud $n$, i un valor $s$, troba totes les parelles de nombres a la llista que sumin aquest valor. És a dir, troba el conjunt següent:

$$\{(L[i], L[j]) \mid i < j, L[i] + L[j] = s,\ \ i, j \in \{0, \ldots, n-1\}\}$$

L'algorisme proposat ha de tenir complexitat $O(n)$.
Fes servir aquesta plantilla per escriure el teu codi:

```
def sumatori_parelles_ord(llista, valorSuma):
    # El teu codi
    return llista_tuples
```

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

- Què sabem si la suma del primer i l'últim element de la llista és inferior a $s$? I si sumen més? Pensa en un recorregut que només faci $\frac{n}{2}$ passos.
- Explica els passos que fa l'algorisme en el cas `sumatori_parelles_ord([1, 2, 3, 5, 7, 8], 10)`.
- Calcula la complexitat de l'algorisme. És $O(n)$?

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

```
assert sumatori_parelles_ord([1, 2, 3, 5, 7, 8], 10)
↪  ==  [(2, 8),(3, 7)]
```

## 5.15 Problema

### Menor i major relatius

**1.** Donada una llista de nombres enters positius i ordenats en ordre creixent $L$ i un valor $k$, escriu una funció que trobi dins la llista $L$ el valor immediatament més petit que $k$ i el valor immediatament més gran que $k$. En qualsevol dels dos casos, retorna -1 si aquest valor no existeix.
Fes servir aquesta plantilla per escriure el teu codi:

```
def menor_major_relatius(nres, k):
    # El teu codi
    return menor, major
```

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

- Podem aplicar algun dels algorismes de dividir i vèncer? Quin cost té l'algorisme original? Com s'ha d'adaptar?
- Si no ho has fet ja, prova d'aplicar l'estratègia de dividir i vèncer. Quins són els subproblemes en què podem dividir el problema principal?
- Calcula la complexitat de l'algorisme.

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

```
assert menor_major_relatius([1, 4, 6, 8, 9], 3) == (1,4)
assert menor_major_relatius([1, 4, 6, 8, 9], 4) == (4,4)
assert menor_major_relatius([1, 4, 6, 8, 9], 5) == (4,6)
assert menor_major_relatius([1, 4, 6, 8, 9], 10) ==
↪  (9,-1)
```

## 5.16 Problema

### Multiplicació de polinomis

**1.** Escriu una funció, `multiplicacio_polinomis`, que calculi la multiplicació de dos polinomis. Cada polinomi es descriurà com una tupla amb els coeficients corresponents ordenats de major a menor grau. Per exemple, la tupla $(3, 1, 4)$ correspon al polinomi $3x^2 + x + 4$.
La funció que escriguis rebrà dues tuples com a entrada i retornarà una tupla corresponent al polinomi resultant després de fer la multiplicació.
Fes servir aquesta plantilla per escriure el teu codi:

```
def multiplicacio_polinomis(polinomi1, polinomi2):
    # El teu codi
    return mult_polinomis
```

**Reflexions prèvies:** Fes aquesta reflexió abans de començar a programar:

- Posa un exemple senzill i fes l'operació manualment.

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

```
assert multiplicacio_polinomis((1, 2, 3, 4), (4, 3,
↪  2, 1)) == [4, 11, 20, 30, 20, 11, 4]
```

## 5.17 Problema

### Patró binari

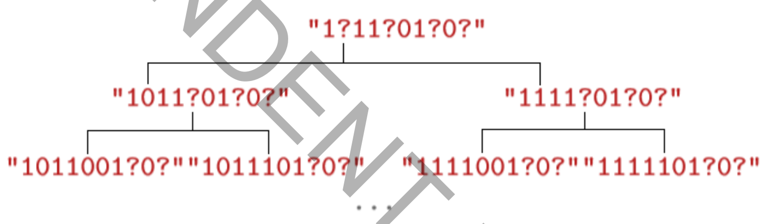
**1.** Donat un patró binari (format per zeros i uns) en què alguns dígits s'han substituït pel caràcter `'?'`, troba totes les possibles combinacions de nombres binaris que es poden obtenir reemplaçant el caràcter `'?'` amb 0 o 1.

Fes servir aquesta plantilla per escriure el teu codi:

```python
def patro_binari(patro):
    # El teu codi
    return nova_llista
```

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

- Si el patró és un *string* podrem manipular els caràcters? Quin altre tipus de dades podem usar per manipular el patró?
- Enfoca el problema com un algorisme recursiu. Identifica el cas base. Com podem recuperar totes les possibles combinacions quan la recursió ha acabat?
- Intenta resoldre manualment l'exemple següent, dibuixant la solució com un arbre. A cada nivell, només es modifica un dígit.




Arbre binari

**Banc de proves:** Entre altres comprovacions, pots provar d'executar el codi següent. Fixa't que les combinacions no tenen per què estar en el mateix ordre que les del teu programa. És per això que fem l'equivalència amb la funció `set()`, que ens permet definir un conjunt sense cap ordre concret.

```python
assert set(patro_binari('1?11?01?0?')) ==
↪   set(['1011001000', '1011001001', '1011001100',
↪   '1011001101', '1011101000', '1011101001',
↪   '1011101100', '1011101101', '1111001000',
↪   '1111001001', '1111001100', '1111001101',
↪   '1111101000', '1111101001', '1111101100',
↪   '1111101101'])
```

## 5.18 Problema

### Implementació eficient de la potència

**1.** Donats dos nombres enters $x$ i $y$, amb $y > 0$, escriu un algorisme recursiu que calculi $x^y$ de manera eficient. No utilitzis cap funció específica de Python per fer aquesta operació.
Fes servir aquesta plantilla per escriure el teu codi:

```
def potencia_eficient(base, exponent):
    # El teu codi
    return potencia
```

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

- Utilitza les propietats dels exponents per afrontar el problema. Pots distingir els casos d'exponent parell i exponent senar.
- Tenint en compte que aquest problema es resol per recursivitat, indica quin és el cas base i com avancem en la recursió.

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

```
assert potencia_eficient(2, 10) == 1024
assert potencia_eficient(3, 4) == 81
assert potencia_eficient(5, 0) == 1
assert potencia_eficient(-2, 3) == -8
```

## 5.19 Problema

### Karatsuba

**1.** Escriu una funció recursiva, `karatsuba`, que calculi la multiplicació pel mètode de Karatsuba en base 10 de dos nombres. Apunta la complexitat de les operacions involucrades i explica quantes vegades es crida en executar `karatsuba(1000, 1050)`.

Fes servir aquesta plantilla per escriure el teu codi:

```
def karatsuba(x,y):
    # El teu codi
    return prod
```

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

- En què es basa aquest algorisme per dividir cada nombre en dos termes?
- Com separaries un nombre de quatre dígits en base 10 en el valor de centenes i en el de desenes usant la divisió i/o el mòdul? Per exemple, 1050 es divideix en 10 centenes i 50 desenes.

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució. Entre d'altres, executa la comprovació següent:

```
assert karatsuba(1100,4050) == 4455000
```

## 5.20 Problema

### Cerca binària

La **cerca binària** és el procediment de buscar un element $K$ en una llista ordenada. Aquest algorisme té una complexitat $O(log_2(n))$. Per a una llista de 1.000.000 elements, només cal fer 20 comparacions!

**Nota:** *Si tenim una llista desordenada de mida $n$ i només hem de buscar uns quants elements, apliquem una cerca exhaustiva. Però si hem de fer moltes cerques (de l'ordre de $n$), serà més eficient ordenar-la primer i fer la cerca binària dels elements després.*

---

**1. Cerca binària**. Donada una llista d'enters ordenada i un valor $K$, escriu una funció recursiva de complexitat $O(log_2(n))$ que retorni si el valor $K$ pertany a la llista.
Fes servir aquesta plantilla per escriure el teu codi:

```
def cerca_binaria(llista, K):
    # El teu codi
    return existeix
```

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

- Quin és el cas base de l'algorisme? És a dir, quin és l'últim cas que s'executarà?
- Pensa que la llista ha d'estar ordenada i que, per tant, només cal comprovar el valor central i una de les dues bandes de la llista.
- Quins casos has de tenir en compte? Què passa si la llista és buida?
- Quantes operacions fa l'algorisme per al cas `cerca_binaria([2, 4, 8, 13, 21, 57], 21)`?

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

```
assert cerca_binaria([2, 4, 8, 13, 21, 57], 10) == False
assert cerca_binaria([2, 4, 8, 13, 21, 57], 21) == True
```

2. **Cerca en una llista quasiordenada**. Donada una llista d'enters quasiordenada, en la qual un element pot estar posicionat un índex per sota o per sobre de la seva posició correcta, i un valor, troba la posició d'aquest valor.

   Fes servir aquesta plantilla per escriure el teu codi:

```python
def cerca_quasiord(x, nres, low, high):
    # El teu codi
    return posicio
```

> **Reflexions prèvies:** Fes aquestes reflexions abans de començar a programar:
> - En què es diferencia aquest algorisme del bàsic de cerca binària?
> - Quantes comparacions es fan en el pitjor cas?
> - Calcula la complexitat de l'algorisme.

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

```python
assert cerca_quasiord(x = 5,
                      nres = [2, 1, 3, 5, 4, 7, 6, 8, 9],
                      low = 0, high = 9) == 3
```

## 5.21 Problema

### Nombres estrictament creixents

1. Donat un nombre enter entre 1 i 9, $N$, escriu una funció que trobi tots els nombres amb $N$ dígits, en els quals els dígits apareguin en ordre creixent. Per exemple 246 seria un nombre de 3 dígits que compliria aquesta condició, però 436 no.

> **Nota:** *Pots pensar en una solució recursiva que vagi generant els nombres de manera incremental, és a dir, començar per pocs números fins a arribar a $N$.*

   Fes servir aquesta plantilla per escriure el teu codi:

```python
def estrictament_creixents(N):
    # El teu codi
    return llista
```

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

- Quina solució retornarà el programa per a $N = 9$? I per a $N = 2$?
- Imaginem que volem construir un nombre de ($N = 7$) dígits i que ja tenim escrit el nombre fins al dígit 5è (per exemple, 12345). Quins passos hem de fer a continuació?
- Quantes combinacions hi ha per a cada valor de $N$ diferent? Revisa si cal la teoria de combinatòria.

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

```
assert estrictament_creixents(8) == [12345678,
↪ 12345679, 12345689, 12346789, 12356789, 12456789,
↪ 13456789, 23456789]
```

## 5.22 Problema

### Ordenar una llista aparellada

**1.** Suposem que necessitem ordenar llistes els nombres de les quals apareixen duplicats, un darrere l'altre. Per exemple, la llista `[1, 1, 4, 4, 3, 3, 5, 5]`. Prova els algorismes d'ordenació `quicksort` i `mergesort`. Calcula la complexitat d'aquests dos algorismes a nivell de comparacions i d'intercanvis d'elements. Mesura el temps d'execució del teu programa per a cadascun.

Observa que en aquest cas, no es demana que el programa retorni res, només ha de reordenar la llista d'entrada.

No utilitzis funcions d'ordenació específiques de Python.

Fes servir aquesta plantilla per escriure el teu codi:

```
def ordenar_parelles(llista):
    # El teu codi
```

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

- Quin dels dos algorismes proposats és, en teoria, més eficient?
- Comenta els resultats del càlcul de temps i de costos amb llistes molt llargues. El resultat obtingut és coherent amb el punt anterior?
- En el cas que no volguéssim modificar la llista original, com ho faríem per retornar una llista nova ordenada?

**Banc de proves:** Defineix diversos exemples per comprovar la teva solució. Entre d'altres, executa la comprovació següent:

```
llista = [3,3,1,1,5,5,0,0]
ordenar_parelles(llista)
assert llista == [0,0,1,1,3,3,5,5]
```



## 5.23 Problema

### Sumatori parcial màxim

L'objectiu d'aquest problema és trobar la subseqüència de suma màxima dins una llista. Aquest problema es pot resoldre de maneres molt diverses. A continuació mostrem tres aproximacions amb complexitats $O(n^2)$, $O(n\log(n))$ i $O(n)$, respectivament.

**1.** Resol el problema del **sumatori parcial màxim** usant un algorisme de força bruta, és a dir, en aquest cas, de complexitat $O(n^2)$.
Fes servir aquesta plantilla per escriure el teu codi:

```
def sumatori_parcial_maxim_forca_bruta(llista):
    # El teu codi
    return max_sum
```

**2.** Resol el problema del **sumatori parcial màxim** usant un algorisme que implementi l'estratègia de dividir i vèncer. Demostra que la complexitat d'aquest algorisme és $O(n\log(n))$.
Fes servir aquesta plantilla per escriure el teu codi:

```
def sumatori_parcial_dividir_vencer(llista):
    # El teu codi
    return max_sum
```

**3.** Resol el problema del **sumatori parcial màxim** usant l'algorisme que es detalla a continuació.
L'**algorisme Kadane** consisteix a recórrer la llista una única vegada. Durant aquesta iteració necessitem emmagatzemar dos valors. Per una banda, la suma parcial de tots els elements anteriors (acum) i, per l'altra, la suma màxima que hem trobat fins al moment (màxim). Si en un moment determinat la suma parcial esdevé negativa, descartem la suma parcial i reiniciem el valor de la suma parcial.
Vegem-ne un exemple. Considerem la llista $[1, 2, -6, 4, -1, 2, 1, -5]$.

- Inicialitzem les dues variables `acum=0, maxim=0`
- Posició 0: `acum=1, maxim=1`
- Posició 1: `acum=3, maxim=3`. Hem sumat el segon element, 2, i hem actualitzat el màxim.
- Posició 2: `acum=0, maxim=3`. Observa que $3-6 = -3 < 0$. L'algorisme de Kadane ens indica que, en el moment que el valor acumulat és negatiu, es descarta la variable que acumula la suma parcial. Observa que tampoc actualitzem la variable del màxim.

- Posició 3: `acum=4, maxim=4`
- Posició 4: `acum=3, maxim=4`. No actualitzem el màxim ja que el valor és inferior.
- Posició 5: `acum=5, maxim=5`
- Posició 6: `acum=6, maxim=6`
- Posició 7: `acum=1, maxim=6`

Observa que amb un sol recorregut de la llista, hem trobat la suma de la subseqüència de suma màxima. Revisa la lògica de l'exemple i de l'algorisme a fons abans de començar a programar.
Fes servir aquesta plantilla per escriure el teu codi:

```
def sumatori_parcial_kadane(llista):
    # El teu codi
    return max_sum
```

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

```
assert sumatori_parcial_maxim_forca_bruta([1, 2, -6,
↪  4, -1, 2, 1, -5]) == 6
assert sumatori_parcial_dividir_vencer([-3, 1, -5, 2,
↪  7, 8]) == 17
assert sumatori_parcial_kadane([1, 2, -5, 3, 6, -2,
↪  4]) == 11
```

## 5.24 Problema

### Xifres i lletres

Considera una llista d'*strings* amb dos tipus de dades diferents: cadenes de lletres i cadenes de nombres, com ara la llista `['aa', '123', 'bc', '98']`. L'objectiu d'aquest problema és ordenar aquesta llista de manera que les cadenes formades per nombres quedin al capdavant. Aquest algorisme no ha de retornar res, només ha de modificar la llista d'entrada.

**1.** Escriu una funció `xifres_lletres_ord_sel` que resolgui el problema usant el mètode d'**ordenació per selecció**. Demostra que la complexitat de l'algorisme que has escrit és $O(n^2)$.
Fes servir aquesta plantilla per escriure el teu codi:

```
def xifres_lletres_ord_sel(llista):
    # El teu codi
```

**2.** Escriu una funció `xifres_lletres_ord_QS` que resolgui el problema usant el mètode de ***quicksort***. Demostra que la complexitat de l'algorisme que has escrit és $O(n \log(n))$.
Fes servir aquesta plantilla per escriure el teu codi:

```
def xifres_lletres_ord_QS(llista):
    # El teu codi
```

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

- Ens cal diferenciar entre els dos tipus de cadenes d'entrada? Executa la instrucció `"A" < "3"` en pPython. Quin sentit té que retorni aquest resultat?
- Quin dels dos algorismes és més eficient? Calcula'n el temps d'execució per a llistes molt grans.

**Banc de proves:** Pots provar l'exemple següent. Recorda, però, que l'únic requeriment és que les cadenes numèriques estiguin al davant de la llista, així que existeixen moltes combinacions correctes del resultat.

```
llista = ["456","789","abc","123","tzu","870","zzz"]
xifres_lletres_ord_QS(llista)
llista
>>> ['456', '789', '870', '123', 'tzu', 'abc', 'zzz']
```

# Apèndix A

# Solucions de problemes seleccionats



**Solució del problema 2.2, pàgina 14**

1. **Solució futval2**:

```python
def futval2(inicial, anys, interesPerCent, comissioPerCent):
    """
    Aquest programa calcula el valor d'una inversió

    Parameters
    ----------
    inicial: float
        Inversio inicial
    anys: int
        Nombre d'anys
    interesPerCent: float
        Interes anual en decimal
    comissioPerCent: float
        Comissio del banc

    Returns
    -------
    diners: float
    """

    for i in range(anys):
        inicial = inicial * (1 + interesPerCent)

    return inicial - inicial * comissioPerCent
```

**Banc de proves:**

```
assert abs(futval2(200,10,0.8,0.3) - 49986.54) < 0.1
assert abs(futval2(100,20,0.8,0.3) - 8923765.35) < 0.1
assert abs(futval2(100,10,0.6,0.3) -  7696.58) < 0.1
assert abs(futval2(100,10,0.8,0.5) - 17852.33) < 0.1
```

## Solució del problema 2.3, pàgina 15

1. **Solució prou_cerveses**:

```
def prou_cerveses(nreCerveses, esCapDeSetmana):
    """
    Aquest programa prediu com anirà la festa en funció de les cerveses.

    Parameters
    ----------
    nreCerveses: int
    esCapDeSetmana: bool

    Returns
    -------
    None
    """

    if nreCerveses < 50:
        return 'Fracàs'
    elif esCapDeSetmana or nreCerveses <= 100:
        return 'Èxit'
    else:
        return 'Fracàs'
```

**Comentaris**:

- La combinació d'if-elses minimitza les comparacions a fer.

**Banc de proves:**

```
assert prou_cerveses(75, False) == 'Èxit'
assert prou_cerveses(75, False) == 'Èxit'
assert prou_cerveses(125, False) == 'Fracàs'
assert prou_cerveses(125, True) == 'Èxit'
assert prou_cerveses(25, False) == 'Fracàs'
assert prou_cerveses(25, True) == 'Fracàs'
```

2. **Solució nota**:

```
def nota(nre):
    """
```

```
    Aquesta funció retorna la qualificació d'un alumne.

    Parameters
    ----------
    nre: float

    Returns
    -------
    qualificació: string
    """
    if nre < 5: qualificacio = 'Suspès'

    elif nre < 7: qualificacio = 'Aprovat'

    elif nre < 9: qualificacio = 'Notable'

    elif nre < 10: qualificacio = 'Excel·lent'

    else: qualificacio = 'Matrícula d\'honor'

    return qualificacio
```

**Banc de proves:**

```
assert nota(0.9) == 'Suspès'
assert nota(5.9) == 'Aprovat'
assert nota(7.9) == 'Notable'
assert nota(9.9) == 'Excel·lent'
assert nota(10) == 'Matrícula d\'honor'
```

## Solució del problema 2.11, pàgina 22

1. **Solució suma_quadrats**:

```
def suma_quadrats(n = 100):
    """
    Aquesta funció calcula la diferència entre la suma dels
    ↪ quadrats i els quadrats de la suma.

    Parameters
    ----------
    n: int

    Returns
    -------
    diff: int
    """

    suma_quadrats = 0
    for i in range(1, n + 1):
```

```python
        quadrats_suma += i

    suma_quadrats = quadrats_suma ** 2

    quadrats_suma = 0
    for i  in range(1, n+1):
        quadrats_suma += i ** 2

    return (suma_quadrats - quadrats_suma)
```

**Solució suma_quadrats (list comprehension)**:

```python
def suma_quadrats(n = 100):
    """
    Aquesta funció calcula la diferència entre la suma dels quadrats i els quadrats de la suma.

    Parameters
    ----------
    n: int

    Returns
    -------
    diff: int
    """

    suma_quadrats = sum([i for i in range(1, n+1)]) ** 2
    quadrats_suma = sum([i ** 2 for i  in range(1, n+1)])
    return (suma_quadrats - quadrats_suma)
```

2. **Solució suma_quadrats**:

```python
def suma_quadrats_lineal(n = 100):
    """
    Aquesta funció calcula la diferència entre la suma dels quadrats i els quadrats de la suma.

    Parameters
    ----------
    n: int

    Returns
    -------
    diff: int
    """
    suma_quadrats = 0
    quadrats_suma = 0

    for i in range(1, n+1):
        suma_quadrats += i**2
        quadrats_suma += i
```

```
    return quadrats_suma**2 - suma_quadrats
```

## Solució del problema 2.13, pàgina 24

1. **Solució perfecte**:

```
def perfecte(nre):
    """
    Aquesta funció comprova si un nombre és perfecte.

    Parameters
    ----------
    nre: int
        El nombre que es vol comprovar.

    Returns
    -------
    b: bool
        Si el nombre d'entrada és un nombre perfecte.

    """
    # Aquesta variable 'suma' ens serveix per emmagatzemar la
    ↪ suma de tots els divisors del nombre d'entrada 'nre'.
    suma = 0
    for div in range(1,nre):  # Recorrem els possibles
    ↪ candidats a ser divisors.
        if nre%div == 0:      # Comprovem si el nombre 'div'
        ↪ divideix el nombre 'nre'.
            suma = suma + div # En cas afirmatiu, l'afegim a
            ↪ la suma.
    b = (suma == nre)         # 'b' és una variable de tipus
    ↪ booleà (True/False).
    return b
```

## Solució del problema 3.3, pàgina 33

1. **Solució hexadecimal**:

```
def convert_hex(xifra):
    """
    Aquesta funció converteix un nombre HEX a DEC.

    Parameters
    ----------
    xifra: string

    Returns
    -------
    x: int
    """
    simbols = {'A': 10, 'B': 11, 'C': 12, 'D': 13, 'E': 14,
    ↪ 'F': 15}
```

```
    potencia = 1
    decimal = 0
    for i in range(len(xifra)-1,-1,-1):
        if xifra[i].isalpha():
            mult = simbols[xifra[i]]
        else:
            mult = int(xifra[i])
        decimal = mult * potencia + decimal
        potencia = potencia * 16
    return decimal
```

2. **Solució digits**:

```
def digits(xifra):
    """
    Aquesta funció retorna el nombre de dígits d'un natural.

    Parameters
    ----------
    xifra: int
        Un nombre natural

    Returns
    -------
    nreDigits: int
        Quantitat de dígits que té la xifra
    """
    nreDigits = len(str(xifra))
    return nreDigits
```

3. **Solució suma_digits**:

```
def suma_digits(nre, potencia):
    """
    Aquesta funció calcula la suma dels dígits d'un nombre
↪   elevat a una potència.

    Parameters
    ----------
    nre: int
        Base
    potencia: int
        Exponent

    Returns
    -------
    suma: int
        Suma dels dígits
    """

    valor = str(nre ** potencia)
```

```
    digits = []
    for digit in valor:
        digits.append(int(digit))

    return sum(digits)
```

**Solució suma_digits (list comprehension)**:

```
def suma_digits(nre, potencia):
    """
    Aquesta funció calcula la suma dels dígits d'un nombre
 ↪  elevat a una potència.

    Parameters
    ----------
    nre: int
        Base
    potencia: int
        Exponent

    Returns
    -------
    suma: int
        Suma dels dígits
    """

    valor = str(nre**potencia)

    return sum([int(digit) for digit in valor])
```

## Solució del problema 3.4, pàgina 35

1. **Solució resta_binaria**:

```
def resta_binaria(op1, op2):
    """
    Aquesta funció calcula la resta binària donats dos
 ↪  nombres.

    Parameters
    ----------
    op1: llista d'enters
    op2: llista d'enters

    Returns
    -------
    resultat: llista d'enters
    """
    def sumaBinaria(nre1, nre2):
        result = []
        add = 0
        for n1, n2 in zip(nre1[::-1], nre2[::-1]):
```

```
            aux = n1 + n2 + add
            add, aux = aux//2, aux %2
            result.insert(0, aux)
        if aux > 1:
            result.insert(0, add)
        return result

    def c1(nombreBinari):
        return [1 if nre == 0 else 0 for nre in nombreBinari]

    def c2(nombreBinari):
        return sumaBinaria(c1(nombreBinari), [0,0,0,1])

    return sumaBinaria(op1, c2(op2))
```



**Solució del problema 3.8, pàgina 41**

1. **Solució eratostenes i primer**:

```
from math import sqrt

def eratostenes(n) :
    """
    Aquesta funció implementa l'algorisme d'Eratòstenes per
    cercar tots els nombres primers fins a n.

    Parameters
    ----------
    n: int

    Returns
    -------
    llista_primers: list
    """
    sieve = [True for j in range(2,n+1)]
    for j in range(2,int(sqrt(n))+1) :
        i = j-2
        if sieve[i]:
            for k in range(j*j,n+1,j) :
                sieve[k-2] = False
    llista_primers = [j for j in range(2,n+1) if sieve[j-2]]
    return llista_primers

def primer(n):
    """
    Aquesta funció retorna l'enèsim primer.

    Parameters
    ----------
    n: int
```

```
    Returns
    -------
    primer: int
    """
    x = n * 50
    sol = eratostenes(x)
    return(sol[n-1])
```

**Comentaris**:

- Hem usat descomposició de funcions.
- En primer lloc creem la llista de primers amb l'algorisme d'Eratòstenes.
- Després agafem el membre $n$.

## Solució del problema 3.11, pàgina 47

1. **Solució fibonacci_matricial**:

```
def multiplicacio(A, B):
    a, b, c, d = A
    x, y, z, w = B
    return (
        a*x + b*z,
        a*y + b*w,
        c*x + d*z,
        c*y + d*w,
    )

def exponenciacio(A, m):
    B = A
    for _ in range(m-1):
        B = multiplicacio(B, A)
    return B

def fibonacci_matricial(n):
    """
    Aquesta calcula els termes de la sèrie de Fibonacci.

    Parameters
    ----------
    n: int

    Returns
    -------
    int
    """
    return exponenciacio([1,1,1,0], n)[1]
```

## Solució del problema 4.2, pàgina 54

1. **Solució aviació**:

```
def aviacio(cadena):
    """
    Aquesta funció converteix una cadena d'entrada a l'alfabet
↪   fonètic.

    Parameters
    ----------
    cadena: string

    Returns
    -------
    traduccio: string
    """
    argot = {
        'A': 'Alpha', 'B': 'Bravo', 'C': 'Charlie', 'D':
        ↪  'Delta', 'E': 'Echo',
        'F': 'Foxtrot', 'G': 'Golf', 'H': 'Hotel', 'I':
        ↪  'India', 'J': 'Juliet',
        'K': 'Kilo', 'L': 'Lima', 'M': 'Mike', 'N':
        ↪  'November', 'O': 'Oscar',
        'P': 'Papa', 'Q': 'Quebec', 'R': 'Romeo', 'S':
        ↪  'Sierra', 'T': 'Tango',
        'U': 'Uniform', 'V': 'Victor', 'W': 'Whiskey', 'X':
        ↪  'Xray', 'Y': 'Yankee',
    }
    c = cadena.upper()
    return [argot[caracter] for caracter in c]
```

## Solució del problema 4.4, pàgina 57

1. **Solució totes_subcadenes**:

```
def totes_subcadenes(cadena):
    """
    Aquesta funció retorna totes les subcadenes de la cadena
↪   donada.

    Parameters
    ----------
    cadena: string

    Returns
    -------
    subcadenes: list
    """
    return [cadena[init : end+1] for init in
    ↪  range(len(cadena)) for end in range(init,
    ↪  len(cadena))]
```

## Solució del problema 4.8, pàgina 60

1. **Solució subcadena_mes_llarga**:

```
def subcadena_mes_llarga(cadena):
    """
    Aquesta funció identifica la subcadena més llarga
    sense cap caràcter repetit.

    Parameters
    ----------
    cadena: string
        Cadena donada

    Returns
    -------
    subcadena: string
        Subcadena més llarga sense caràcters repetits
    """
    n = len(cadena)

    # Fem un diccionari amb els caràcters.
    diccCaracters = {caracter: False for caracter in cadena}

    # La solució es basa en una finestra que es redefineix
    ↪ cada cop que troba un caràcter repetit.
    iniciFinestra = 0

    # Inici i final de la finestra. El final és el darrer
    ↪ caràcter vist; el principi és el primer caràcter no
    ↪ repetit des del final de la finestra.
    iniciSubcadena = 0
    finalSubcadena = 0

    # Inici i final de la subcadena més llarga trobada fins al
    ↪ moment
    for finalFinestra in range(0, n):

        if diccCaracters[cadena[finalFinestra]]:
            # Si el caràcter ja hi era.
            while (cadena[iniciFinestra] !=
            ↪ cadena[finalFinestra]):
                diccCaracters[cadena[iniciFinestra]] = False
                iniciFinestra += 1
                # Desplacem la finestra a partir de la primera
                ↪ aparició del caràcter, sense incloure'l.
            iniciFinestra += 1

        else:
            diccCaracters[cadena[finalFinestra]] = True
            # Anotem el caràcter com a existent.
            finalFinestra += 1
```

```
                # Com que no hi ha repeticions augmentem la
                ↪  finestra.
                if finalSubcadena - iniciSubcadena < finalFinestra
                ↪  - iniciFinestra:
                    # Revisem que la nova subcadena no sigui més
                    ↪  llarga que la que teníem guardada.
                    iniciSubcadena = iniciFinestra
                    finalSubcadena = finalFinestra

        return cadena[iniciSubcadena:finalSubcadena]
```

## Solució del problema 5.2, pàgina 65

1. **Solució busca_unic**:

```
def busca_unic(llista):
    """
    Aquesta funció busca l'element que només surt una vegada.

    Parameters
    ----------
    llista: list

    Returns
    -------
    nre: int
    """

    def busca_unic_rec (llista, inici, fi):
        if inici == fi: return llista [inici]

        mid = inici + (fi - inici) // 2

        # Posició parell
        if mid % 2 == 0:
            # A mid hauria d'estar el primer element de la
            ↪  parella.
            if llista [mid] == llista [mid + 1]:
                # Fins aquí tot ha anat bé; busquem a la
                ↪  segona meitat.
                return busca_unic_rec (llista, mid + 2, fi)
            else:
                # No som on hauríem de ser! Cerca a la primera
                ↪  meitat.
                # Important no fer mid-1, ja que podríem estar
                ↪  en el nombre que busquem.
                return busca_unic_rec (llista, inici, mid)

        # Posició imparell
        else:
```

```
            # A mid hauria d'estar el segon element de la
            ↪ parella.
            if llista [mid-1] == llista [mid]:
                # Fins aquí tot ha anat bé; busquem cap
                ↪ endavant.
                return busca_unic_rec (llista, mid + 1, fi)
            else:
                # No som on hauríem de ser! Cerca a la primera
                ↪ meitat.
                return busca_unic_rec (llista, inici, mid-1)

    assert len(llista) % 2 != 0, 'La longitud de la llista ha
    ↪ de ser senar.'
    return busca_unic_rec (llista, 0, len (llista) -1)
```



**Solució del problema 5.4, pàgina 66**

1. **Solució el_mes_petit**:

```
def bin_falta(llista, low, high):

    mid = (low+high) // 2

    if high-low < 1:
        return low if(llista[low] != low) else -1

    #Observo que els índexs coincideixen amb el dígit fins que
    ↪ arribo al que falta.
    elif llista[mid] == mid :
        return bin_falta(llista, mid + 1, high)

    elif llista[mid-1] == mid-1:
        return mid

    else:
        return bin_falta(llista, low, mid-1)

def el_mes_petit(llista):
    """
    Aquesta funció troba el valor més petit que falta.

    Parameters
    ----------
    llista: list

    Returns
    -------
    valor: int
    """
    if llista == []:
        return ("Entrada no vàlida")
```

```
        else:
            return bin_falta(llista, 0, len(llista) - 1)
```

## Solució del problema 5.5, pàgina 67

1. **Solució quants1**:

```
def quants1(llistaBinaria):
    """
    Aquesta funció troba de manera eficient el nombre d'uns que conté una llista.

    Parameters
    ----------
    llistaBinaria: list
        La llista de nombres binaris ordenada

    Returns
    -------
    num1s: int
        El nombre d'uns que conté
    """

    def rec_nombre_1_binsearch(llista, low, high):
        if low > high:
            return 0

        elif low == high:
            return llista[low]

        mid = (low + high) // 2

        items = llista[mid]

        if items == 1:
            return rec_nombre_1_binsearch(llista, low, mid-1) + high - mid + 1

        else:
            return rec_nombre_1_binsearch(llista, mid+1, high)

    return rec_nombre_1_binsearch(llistaBinaria, 0, len(llistaBinaria) - 1)
```

## Solució del problema 5.7, pàgina 68

1. **Solució zeros_final**:

```
def zeros_final(llista):
    """
```

```
    Aquesta funció mou tots els zeros de la llista donada al
↪   final. La funció és una variació del quicksort en què fem
↪   swap dels elements de la llista usant com a pivot el 0.

    Parameters
    ----------
    llista: list

    Returns
    -------
    None
    """
    j = 0
    for i in range(0, len(llista)):
        if llista[i] != 0:
            llista[i], llista[j] = llista[j], llista[i]
            j += 1
```

**Solució del problema 5.12, pàgina 72**

1. **Solució parts_llista**:

```
def parts_llista(llista, valor):
    """
    Aquesta funció divideix la llista en 3 a partir del valor
↪   donat.

    Parameters
    ----------
    llista: list
    valor: int

    Returns
    -------
    None
    """
    primer = 0
    ultim = len(llista) - 1
    i = 1
    j = ultim
    # Nombre de valors iguals
    iguals_v = 0

    while i<j:
        while iguals_v < ultim and llista[iguals_v] == valor:
            iguals_v += 1

        i = iguals_v
        while i < ultim and llista[i] < valor:
            i += 1
```

```
        while j > primer and llista[j] > valor:
            j -= 1

        while i < ultim and llista[i] == valor:
            llista[iguals_v], llista[i] = llista[i],
            ↪  llista[iguals_v]
            i += 1
            iguals_v += 1

        #Intercanviem, fem avançar i i j.
        llista[i], llista[j] = llista[j], llista[i]

# El programa no retorna res, però l'ordre de la llista s'ha
↪   modificat.
```

**Solució del problema 5.16, pàgina 75**

1. **Solució multiplicacio_polinomis**:

```
def multiplicacio_polinomis(polinomi1, polinomi2):
    """
    Aquesta funció multiplica dos polinomis.

    Parameters
    ----------
    polinomi1: tupla
    polinomi2: tupla

    Returns
    -------
    mult_polinomis: tupla
    """
    grauPol1, grauPol2 = len(polinomi1) - 1, len(polinomi2) -
    ↪  1

    # El resultat serà un polinomi de grau suma
    mult_polinomis = [0] * (grauPol1 + grauPol2 + 1)

    # dels graus dels polinomis producte, que en particular té
    ↪  grau + 1 de coeficients.
    for i in range(grauPol1 + 1):
        for j in range(grauPol2 + 1):
            mult_polinomis[i+j] += polinomi1[i] * polinomi2[j]

    return mult_polinomis

    assert multiplicacio_polinomis((1, 2, 3, 4), (4, 3, 2, 1))
    ↪  == [4, 11, 20, 30, 20, 11, 4]
```

**Comentaris**:

- El que fem és fer una multiplicació normal i corrent de polinomis amb l'algorisme habitual. L'única particularitat és que sabem que quan multipliquem dos elements de grau $r$ i $s$ respectivament ens donarà un element de grau $r + s$; per tant, el que fem és sumar-ho a la suma de coeficients del mateix grau dins del bucle. Aquí, com que estan ordenats a la inversa, l'assignem d'una forma més directa que amb els graus per evitar invertir la llista, però es podria fer assignant a l'índex = element del grau del coeficient i després invertint.

  Per tant, la complexitat queda, essent $n$ i $m$ els graus dels polinomis corresponents, $O(n \times m)$.

**Solució del problema 5.17, pàgina 76**

1. **Solució patro_binari**:

```
def patro_binari(patro):
    """
    Aquesta funció troba totes les possibles combinacions de
↪   nombres binaris a partir del patró.

    Parameters
    ----------
    patro: string

    Returns
    -------
    combinacions: list
    """
    # Conversió a llista
    if type(patro) == str:
        patro = list(patro)

    if len(patro) == 1:
        # Cas base
        return ['0','1'] if patro[0] == '?' else patro

    else:
        # Ja hem fet els canvis 1->
        combinacions = patro_binari(patro[1:])
        nova_llista = []
        for elem in combinacions:
            if patro[0] == '?':
                nova_llista.append('0' + elem)
                nova_llista.append('1' + elem)

            else:
                nova_llista.append(patro[0] + elem)

        return nova_llista
```

**Solució del problema 5.20, pàgina 78**

1. **Solució cerca_binaria recursiva**:

```
def cerca_binaria(arr, low, high, x):
    # Comprova el cas base.
    if high >= low:

        mid = (high + low) // 2
        # Si l'element és just al mig.
        if arr[mid] == x:
            return True

        # Si l'element és més petit que 'mid', només pot
        ↪ trobar-se a la part esquerra.
        elif arr[mid] > x:
            return cerca_binaria(arr, low, mid - 1, x)

        # Altrament l'element només pot trobar-se a la part
        ↪ dreta.
        else:
            return cerca_binaria(arr, mid + 1, high, x)
    else:
        # L'element no és a la llista.
        return False
```

2. **Solució cerca_binaria iterativa**:

```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    mid = 0

    while low <= high:
        mid = (high + low) // 2

        # Comprova si la x es troba a la posició 'mid'.
        if arr[mid] < x:
            low = mid + 1

        # Si la x és més gran, ignora la meitat esquerra.
        elif arr[mid] > x:
            high = mid - 1

        # Si la x és més petita, ignora la meitat dreta.
        else:
            return True

    # Si hem arribat aquí, vol dir que l'element no era a la
    ↪ llista.
    return False
```

## Solució del problema 5.20, pàgina 78

1. **Solució cerca_quasiord**:

```python
def cerca_quasiord(x, nres, low, high):
    """
    Aquesta funció, donada una llista d'enters quasiordenada,
    ↪ en què un element pot estar posicionat un índex per sota o
    ↪ per sobre de la seva posició correcta, i un valor, troba
    ↪ la posició d'aquest valor.

    Parameters
    ----------
    x: int
        El nombre que es vol trobar.
    nres: int
        La llista de nombres que es vol trobar.
    low: int
        L'índex més baix de la subllista on estem cercant ara.
    high: int
        L'índex més alt de la subllista on estem cercant ara.

    Returns
    -------
        posicio: int
    """

    if low > high:
        return -1

    mid = (low + high) // 2
    items = nres[mid]

    # Comparem amb l'índex que tocaria.
    if items == x:
        return mid
    # Comparem amb l'índex inferior.
    if mid - 1 >= low and nres[mid - 1] == x:
        return mid-1
    # Comparem amb l'índex superior.
    if mid + 1 <= high and nres[mid + 1] == x:
        return mid+1
    # El movem dues posicions.
    elif x < items:
        return cerca_quasiord(x, nres, low, mid-2)
    # El movem dues posicions.
    else:
        return cerca_quasiord(x, nres, mid+2, high)
```

## Solució del problema 5.21, pàgina 79

1. **Solució estrictament_creixents**:

```python
def estrictament_creixents(N):
    """
    Aquesta funció troba tots els nombres de N dígits en què
↪   aquests són estrictament creixents.

    Parameters
    ----------
    N: int

    Returns
    -------
    llista: list
    """

    def estr_creix_rec(cadena, n, previ, llista):
        # Ja no queden dígits per calcular.
        if n == 0:  llista.append(int(cadena))

        # El mòdul 11 és per a \textit{n} = 0.
        for i in range(previ + 1, (10 - n + 1) % 11):
            # Fem 10-previ crides, per als valors entre previ
            ↪   i 9 del dígit enèsim.
            estr_creix_rec('{}{}'.format(cadena, i), n - 1, i,
            ↪   llista)

    llista = []
    estr_creix_rec('', N, 0, llista)
    return llista

    assert estrictament_creixents(8) == [12345678,  12345679,
    ↪   12345689,  12345789,  12346789,  12356789,  12456789,
    ↪   13456789,  23456789]
```

**Comentaris**:

- Usem una funció auxiliar per encapsular tots els paràmetres necessaris en les crides recursives.
- Anem recorrent totes les possibilitats per cada dígit tenint en compte els límits imposats pel nombre de dígits.