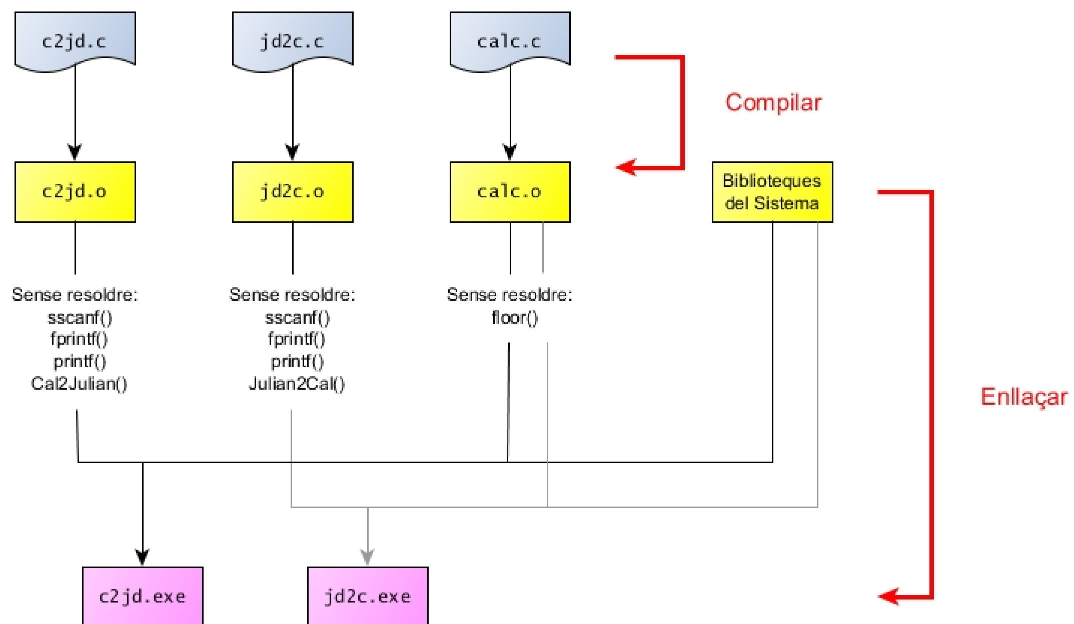
Aureli Alabert

Introducció a la Informàtica

per a estudiants de matemàtiques

teoria i exercicis



Amb la col·laboració de Xavier Xarles i Josep Maria Mondelo

©Aureli Alabert 2011-2020

Amb la col·laboració de Xavier Xarles (2019-2020)

Apunts basats en notes prèvies elaborades per Josep Maria Mondelo (2010).

Es permet la reproducció per a estudi privat. Per a qualsevol altre ús, contactar amb l'autor.

1

Algorismes

Podem definir *algorisme* com una formulació inequívoca, sense ambigüitats, d'un mètode que permet resoldre un cert problema, independentment de les dades concretes. Com a tal, ha de complir dues propietats fonamentals:

1. Ha d'expressar de forma molt clara la successió d'accions elementals que cal seguir per a la resolució del problema.
2. Ha de preveure tot el conjunt de valors que les dades puguin prendre i totes les alternatives que aquestes puguin plantejar.

Una manera d'expressar algorismes és fer servir el llenguatge ordinari. Per exemple, podem explicar així l'algorisme d'Euclides per a calcular el màxim comú divisor de dos enters positius:

1. Dividir el nombre més gran pel més petit.
2. Si el residu de la divisió és zero, el màxim comú divisor és el divisor i hem acabat.
3. Si el residu de la divisió no és zero, dividir el divisor pel residu.
4. Tornar al pas 2.

Malgrat que els passos a seguir són prou clars, hi ha problemes formals: els nombres que al començament del pas 3 són el divisor i el residu, al final d'aquest pas són el dividend i el divisor. Tot i que "ja ens entenem", si formulem algorismes més complicats d'aquesta manera podem produir ambigüitats. I si volem que els executi una màquina hem de ser més precisos.

1.1. Representació mitjançant diagrames de flux

Els *diagrames de flux* (*flow charts*) són una manera gràfica de representar algorismes que es va popularitzar als inicis de la informàtica. En ells:

- El flux d'execució se representa amb fletxes.
- Les operacions elementals amb rectangles.
- Les alternatives amb rombes.
- Les operacions d'entrada o sortida amb romboides.

Vegeu a la Figura 1.1 un exemple de diagrama de flux per al càlcul del factorial d'un nombre enter positiu.

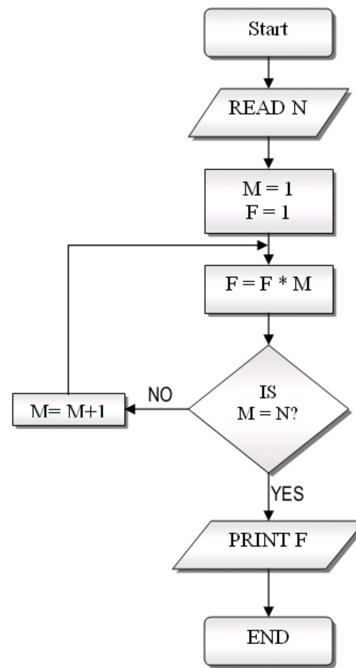


Figura 1.1: Exemple de diagrama de flux per calcular el factorial d'un número (Wikipedia).

L'inconvenient dels diagrames de flux és que es fan molt difícils de dibuixar i d'entendre quan són molt grans.

1.2. Representació mitjançant pseudocodi

El *pseudocodi* és una representació d'algorismes propera als llenguatges de programació, però en la que ens despreocupem de detalls feixucs com ara l'entrada i la sortida (lectura de dades, escriptura de resultats), que distreuen l'atenció de l'algorisme en sí.

Per exemple, al Llistat 1.2 hi ha la formulació en pseudocodi de l'algorisme per trobar el màxim comú divisor de dos nombres que hem descrit abans en llenguatge ordinari. Noteu que totes les possibles ambigüitats han desaparegut.

1.2.1 Estructures algorísmiques

Tot algorisme pot expressar-se mitjançant les tres *estructures algorísmiques* (o de *control de flux*) següents:

(1) Seqüencial. És una seqüència d'instruccions que s'executen una rere l'altra, en l'ordre especificat.

Exemple d'estructura seqüencial:

```
llegeix(a)
b=2*a
escriu(b)
```

(2) Alternativa. Segons si es compleix o no una *condició*, s'executa una seqüència d'instruccions o una altra. La condició és una expressió que

```
 1 INICI
 2    enter m, n, r
 3    llegeix(m,n)
 4    SI m<=0 O n<=0 O m<n LLAVORS
 5       escriu( "Error: cal que m,n>0 i m>=n")
 6       PARAR
 7    FI_SI
 8    r = m % n
 9    MENTRE r != 0 FER
10       m = n
11       n = r
12       r = m % n
13    FI_MENTRE
14    escriu(n)
15 FI
```

Llistat 1.2: Formulació en pseudocodi de l'algorisme d'Euclides per trobar el màxim comú divisor de dos nombres enters. L'operador % vol dir calcular el residu de la divisió entera.

es pot avaluar i dóna com a resultat un sí o un no (verdader o fals).

```
SI condició LLAVORS
   instruccions_1
SI_NO
   instruccions_2
FI_SI
```

(3) Iterativa. Mentre es compleixi una certa condició, es repeteix un grup d'instruccions. Si la condició no es compleix d'entrada, les instruccions mai no es realitzen.

```
MENTRE condició FER
   instruccions
FI_MENTRE
```

Combinant les tres estructures anteriors es pot codificar qualsevol algorisme. Cal pensar que allà on es pot posar una instrucció, es pot posar qualsevol de les estructures. Una *instrucció elemental* és, o bé una *assignació*, com ara `b=2*a`, o bé una *crida a una funció*, com `escriu(b)`[1]. Una assignació el que fa, essencialment, és posar un valor a una *variable*; les funcions les veurem a l'Apartat 1.2.3.

Una manera més formal de dir que tot algorisme es pot posar en termes d'aquestes estructures és la següent:

⟨*algorisme*⟩ ::= `INICI` ⟨*est_seq*⟩ `FI`

⟨*est_seq*⟩ ::= ⟨*instrucció*⟩ ⟨*est_seq*⟩ | ∅

⟨*instrucció*⟩ ::= ⟨*instrucció_elemental*⟩ | ⟨*est_alt*⟩ | ⟨*est_iter*⟩

⟨*est_alt*⟩ ::= `SI` ⟨*condició*⟩ `LLAVORS` ⟨*est_seq*⟩
`SI_NO` ⟨*est_seq*⟩ `FI_SI`

⟨*est_iter*⟩ ::= `MENTRE` ⟨*condició*⟩ `FER` ⟨*est_seq*⟩ `FI_MENTRE`

[1] instrucció = *statement*, assignació = *assignment*, crida a una funció = *function call*.

⟨*condició*⟩ ::= ⟨*expressió*⟩ ⟨*relació*⟩ ⟨*expressió*⟩

I aquí hauríem de seguir definint coses encara indefinides, com ara "instrucció_elemental", "expressió", "relació", si volem una descripció completa, però no cal que baixem a més detall aquí. Aquesta mena de descripció s'anomena *Backus-Naur Form (BNF)*, i s'utilitza habitualment per descriure la sintaxi de llenguatges concrets[2].

Cada llenguatge de programació tradueix les estructures algorísmiques a la seva manera. En principi, podem programar qualsevol algorisme en qualsevol llenguatge sabent com traduir les tres estructures anteriors. Per exemple, en Python[3] l'estructura alternativa s'escriu

```
if condició :
    instruccions_1
else:
    instruccions_2
```

En C, la mateixa estructura s'escriu

```
if( condició )
    { instruccions_1 }
else
    { instruccions_2 };
```

En Python, els canvis de línia i la indentació formen part de la sintaxi del llenguatge; en C no, i s'utilitzen les claus i els punt-i-coma per evitar ambigüitats.

L'estructura iterativa s'escriu en Python com

```
while condició :
    instruccions
```

i en C com

```
while( condició )
    { instruccions };
```

A la pràctica, necessitem més estructures per treballar amb comoditat. Tots els llenguatges de programació en tenen.

(4) Alternativa simplificada. El segon conjunt d'instruccions en l'estructura alternativa pot ser buit. En tal cas, es pot escriure simplement

```
SI condició LLAVORS
   instruccions
FI_SI
```

(5) Alternativa múltiple. Generalització de l'estructura alternativa.

```
SEGONS expressió
   CAS valor_1 : instruccions_1
   CAS valor_2 : instruccions_2
   ...
   CAS valor_n : instruccions_n
   CAP_CAS : instruccions_n+1
FI_SEGONS
```

(6) Iterativa amb condició final. Com la iterativa, però, les accions sempre s'executen almenys una vegada.

[2]Vegeu per exemple la descripció completa en BNF del llenguatge Pascal `http://goo.gl/OFlI5W` o del llenguatge C `http://goo.gl/T6pnO`.

[3]El SageMath segueix la sintaxi del llenguatge Python.

```
FER
   instruccions
FINS_QUE condició
```

(7) Iterativa amb condició intermèdia. Les iteracions es poden interrompre pel mig.

```
FER
   instruccions_1
ABANDONAR_SI condició
   instruccions_2
FI_FER
```

(8) Repetitiva. Es repeteix un conjunt d'instruccions per a un rang de valors d'un índex, conegut d'entrada.

```
DES_DE index = expr_1 FINS_A expr_2 INCREMENT expr_3 FER
   instruccions
FI_DES_DE
```

Quan l'increment és 1 (cas freqüent) no cal posar-lo.

A les estructures iteratives (3), (6), (7) és molt important assegurar-se que la condició de sortida es donarà en algun moment i per tant les iteracions no continuaran per sempre.

Un llenguatge de programació particular pot tenir variants i combinacions d'aquestes estructures, o pot ser que li'n falti alguna.

1.2.2 Tipus de dades i variables

Podem pensar que una *variable* és un nom amb què etiquetem un tros de la memòria de l'ordinador, i on volem dipositar un *valor*.

Quan fem una assignació a una variable (per exemple, `cost = 3`) estem dipositant el valor `3` en una "capsa", etiquetada amb el nom `cost`, que està guardada en un cert lloc de la memòria de l'ordinador. Quan més tard usem el nom `cost` per fer un càlcul, estem llegint el valor que hi ha guardat en aquella capsa.

Però de valors n'hi ha de diversos tipus. Alguns llenguatges de programació necessiten saber d'entrada quin tipus de valor posarem sota cada nom de variable. En tal cas podem parlar de *tipus de variables*. Per exemple, el C ho requereix; el Python no. Això fa al Python més fàcil d'escriure, però en C és més fàcil detectar errors.

Al representar un algorisme en pseudocodi, és convenient explicitar de quin tipus serà cada variable que utilitzem. Els tipus bàsics són:

- Tipus *simples* (o escalars): les *declararem* mitjançant les paraules `enter`, `real` o `caracter`. Al començament del Llistat 1.2 hi ha declarades tres variables enteres, de noms `m`, `n`, `r`.
- *Vectors* (*arrays*): són grups de dades simples del mateix tipus, indexades per un o més índexs. Declararem vectors escrivint el tipus, el nom del vector i el nombre d'elements entre parèntesis quadrats. Per exemple, declararem així un vector de 16 nombres reals anomenat `a`:

  ```
  real a[16]
  ```

  Per accedir al component i-èsim del vector, escriurem `a[i]`. Ara bé, convé tenir clar quins són el primer i l'últim índex. Hi ha dos convenis

habituals: començar per `i=1` o començar per `i=0`. En el primer cas, els 16 components són `a[1]`,...,`a[16]`; en el segon, `a[0]`,...,`a[15]`. En el llenguatge C, els índexs sempre comencen en 0. Per evitar confusions, al escriure pseudocodi podem fer també

```
real a[0..15]
```

Els arrays poden tenir més d'un índex. Per exemple,

```
real a[3,4]
```

pot servir per guardar a la variable `a` una matriu de 3 files i 4 columnes.

```
1  INICI
2     enter v[5000], n, i, j
3     llegeix(n)
4     SI n>5000 LLAVORS
5        escriu("massa gran, memòria insuficient");
6        PARAR
7     FI_SI
8  /* Inicialitzem garbell */
9     DES_DE i=1 FINS_A n FER
10       v[i]=1
11    FI_DES_DE
12 /* Fem el garbell. Per "ratllar" posem a zero */
13    v[1]=0
14    i=2
15    MENTRE i <= sqrt(n)  FER
16       DES_DE j=2 FINS_A n/i FER
17          v[i*j] = 0
18       FI_DES_DE
19       FER
20          i=i+1
21       FINS_QUE v[i]!=0 O i > sqrt(n)
22    FI_MENTRE
23 /* Escrivim els nombres primers */
24    DES_DE i=1 FINS_A n INCREMENT 1 FER
25       SI v[i] != 0 LLAVORS
26          escriu (i)
27       FI_SI
28    FI_DES_DE
29 FI
```

Llistat 1.3: Formulació en pseudocodi del garbell d'Eratòstenes. Les frases parentetizades per `/*...*/` són comentaris explicatius, que es bo utilitzar també en els programes reals.

Com a exemple de l'ús de vectors i les estructures algorísmiques vistes, al Llistat 1.3 trobareu una formulació en pseudocodi del *garbell d'Eratòstenes*, que serveix per trobar tots els nombres primers més petits o iguals que un natural donat n. Recordeu que aquest procediment es pot aplicar a mà de la manera següent: escrivim tots els nombres de 1 a n, ratllem el 1, ratllem tots els múltiples de 2, ratllem tots els múltiples del primer número no ratllat, i així successivament fins a arribar al final. Els que queden no ratllats són els nombres primers.

Noteu que, quan descomponem $m \in \mathbb{N}$, $m \leq n$ com $m = m_1 m_2$ amb $m_1, m_2 \in \mathbb{N}$, necessàriament o bé $m_1 \leq \sqrt{n}$ o bé $m_2 \leq \sqrt{n}$. Per tant, en aplicar el garbell d'Eratòstenes, només cal considerar múltiples de nombres fins a $\lfloor\sqrt{n}\rfloor$.

1.2.3 Funcions

Les *funcions* s'usen per agrupar trossos de codi que duen a terme accions que s'han de fer moltes vegades a partir de dades inicials diferents. També es poden usar per fer més clar un algorisme complicat.

En els llocs de l'algorisme on s'ha d'executar la funció s'escriu la *crida a la funció* (*function call*) en la qual es posa la *llista d'arguments*, que són les dades amb què volem executar-la.

En la *definició de la funció* es posa al principi una *llista de paràmetres*, que són els noms que la funció usarà internament per a les dades que se li passin com a arguments.

Si les accions a fer comporten l'obtenció d'un valor resultat, la funció *retorna* aquest valor. El codi que ha cridat la funció pot aleshores usar aquest valor en els càlculs que estigui fent. Una funció pot perfectament no retornar cap valor.

```
1 FUNCIO tipus nom ( llista_de_parametres )
2     ...
3     RETORN expressio
4     ...
5 FI_FUNCIO
```

Llistat 1.4: Funció en pseudocodi.

Escriurem les funcions en pseudocodi tal com està esquematitzat al Llistat 1.4. El `tipus` és el del valor que retorna la funció. No posarem tipus a les funcions que no retornen res.

Com a primer exemple, el Llistat 1.5 calcula el cosinus de l'angle que formen dos vectors. Si $u, v \in \mathbb{R}^n$, el cosinus del seu angle α es pot calcular com

$$\cos\alpha = \frac{\langle u, v\rangle}{\sqrt{\langle u, u\rangle \cdot \langle v, v\rangle}} ,$$

on

$$\langle u, v\rangle = \sum_{i=1}^{n} u_i \cdot v_i$$

és el *producte escalar* de u i v. En aquest llistat, la funció `pesc` ens ha estalviat haver d'escriure tres estructures repetitives per tal de fer els tres productes escalars necessaris.

Considerem ara el codi següent, que és la definició d'una funció escrita en Python:

```
1  FUNCIO real pesc (enter n, real u[n], real v[n])
2    enter i
3    real p
4    p=0
5    DES_DE i=1 FINS_A n INCREMENT 1 FER
6       p=p+u[i]*v[i]
7    FI_DES_DE
8    RETORN p
9  FI_FUNCIO
10
11 INICI
12   real v[5000],w[5000]
13   enter d,i
14   llegeix(d)
15   llegeix(v[i],i=1..d)
16   llegeix(w[i],i=1..d)
17   escriu(pesc(d,v,w)/
18            sqrt(pesc(d,v,v)*pesc(d,w,w)))
19 FI
```

Llistat 1.5: Programa per trobar el cosinus de l'angle que formen dos vectors.

```
def f(x,y):
    global d
    c=3.
    b=17.
    z=c*b*x*y*d
    return z
```

La funció s'anomena `f` i la llista de paràmetres és `(x,y)`. Les variables `c,b,z` són *locals*. Això vol dir que encara que en el codi que cridi a la funció hi hagi variables amb aquest nom, dins de la funció són unes variables diferents, noves. La variable `d` en canvi és *global*; se suposa que hi ha una variable que es diu així que existeix en algun lloc extern a la funció, i que és la que usarem dins de la funció.

La funció `f` assigna un valor a les variables locals `c` i `b`, i usa aquests valors, més el de la variable global `d`, més el dels paràmetres `x` i `y`, per assignar un valor a la variable local `z`. La funció així definida pot ser cridada des d'una altra funció:

```
a=5; b=2;
r=f(a,b);
```

Els arguments `a` i `b` tenen valors concrets, que són copiats als paràmetres `x` i `y`. La funció `f` retorna un valor, que s'assigna a la variable `r`. Noteu que la variable `b` de fora i la de la dins de la funció no són la mateixa.

Hi ha un detall ambigu en el mecanisme arguments–paràmetres en la crida i execució d'una funció: Què passa si dins de la funció modifiquem un paràmetre? Per exemple, si a la funció de Python que acabem de veure contingués una instrucció com ara `x=3`, hauria de modificar això el valor de l'argument `a` que s'ha passat al paràmetre `x`?

Si pensem que en la variable `x` hem simplement copiat el valor de `a`, està clar que encara que modifiquem `x` dins de la funció, el valor emmagatzemat dins la caixa etiquetada com `a` no canviarà. En aquest cas, diem que hem passat l'argument *per valor* (*passing by value*).

Si, en canvi, pensem que el que hem passat és el lloc on es troba la caixa etiquetada com `a`, i ara li hem posat també l'etiqueta `x`, aleshores quan canviem el valor de `x`, el valor de `a` també canviarà. Direm que hem passat l'argument *per referència* (*passing by reference*).

Depenent del cas, convé una cosa o l'altre. El més pràctic és fer el següent conveni: Quan passem una dada de tipus escalar, la passem per valor, si no es diu el contrari; quan passem un vector, el passem per referència.

Quan vulguem que una variable simple es passi per referència, ho hem d'explicitar d'alguna manera. Per exemple, escrivint al davant del argument la paraula `ref`. Com a exemple, el pseudocodi del Llistat 1.6 fa exactament el mateix que el del Llistat 1.5, però ara la funció `pesc` no retorna cap valor.

```
1  FUNCIO pesc (enter n, real u[n], real v[n], ref real
      p)
2    enter i
3    p=0
4    DES_DE i=1 FINS_A n INCREMENT 1 FER
5       p=p+u[i]*v[i]
6    FI_DES_DE
7  FI_FUNCIO
8
9  INICI
10   real v[5000],w[5000]
11   enter d,i
12   real p1,p2,p3
13   llegeix(d)
14   llegeix(v[i],i=1..d)
15   llegeix(w[i],i=1..d)
16   pesc(d,v,w,p1)
17   pesc(d,v,v,p2)
18   pesc(d,w,w,p3)
19   escriu(p1/sqrt(p2*p3))
20 FI
```

Llistat 1.6: Programa per trobar el cosinus de l'angle que formen dos vectors, en el qual la funció producte escalar torna el resultat per referència.

1.3. Iteració i recursivitat

D'aquí al final del capítol veurem algunes idees algorísmiques bàsiques. Serviran també per practicar l'escriptura en pseudocodi.

Una successió $\{x_n\}_n$ definida de manera *recurrent*,

$$x_n = f(x_{n-1}, x_{n-2}, \dots, x_{n-k}) \ , \quad \text{per } n \geq k$$

es pot generar amb un programa mitjançant l'ús d'estructures repetitives o iteratives. Com a exemple, considerem l'avaluació d'un polinomi,

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \ .$$

Existeix una manera d'avaluar-lo amb $2n$ operacions, coneguda com a *regla de Horner*. Consisteix a reescriure el polinomi de la manera següent (posem $n = 3$ per exemple):

$$\underbrace{a_0 + x \underbrace{\left(a_1 + x \underbrace{\left(a_2 + x \underbrace{a_3}_{p_3}\right)}_{p_2}\right)}_{p_1}}_{p_0}$$

Els valors intermedis $\{p_i\}_{i=0}^{n}$ es poden generar recurrentment així:

```
p[n]=a[n]
DES_DE i=n-1 FINS_A 0 INCREMENT -1 FER
   p[i]=p[i+1]*x+a[i]
FI_DES_DE
```

Al llistat 1.7 trobareu un pseudocodi complet que implementa la recurrència anterior per al calcul de $p_0 = p(x)$ a partir de x i $a_0, \ldots, a_n$. Observeu que no cal usar un vector per guardar els valors intermedis $p_1, \ldots, p_n$.

```
1 FUNCIO real horner (enter n, real a[0..n], real x)
2    real p
3    enter i
4    p=a[n]
5    DES_DE i=n-1 FINS_A 0 INCREMENT -1 FER
6       p=p*x+a[i]
7    FI_DES_DE
8    RETORN p
9 FI_FUNCIO
```

Llistat 1.7: Funció que implementa la regla de Horner per a l'avaluació d'un polinomi.

Una manera d'avaluar recurrències sense usar estructures repetitives és emprar *funcions recursives*. Una funció recursiva és una funció que es crida a si mateixa. Per exemple, si definim el factorial d'un nombre natural de manera recurrent com

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ n \cdot (n-1)! & \text{si } n \geq 1, \end{cases}$$

podem convertir aquesta definició en una funció de manera directa tal com s'ha fet al Llistat 1.8.

Tota funció recursiva ha de contenir alguna instrucció de bifurcació que eviti que s'autocridi indefinidament. Ens hem d'assegurar que sempre arribi un moment en què la condició de bifurcació es compleixi.

Com a regla general, s'ha de tendir a evitar la recursivitat. Les implementacions recursives són molt elegants, però gairebé sempre són menys eficients que les iteratives, degut a:

```
1 FUNCIO enter fact (enter n)
2    SI n==0 LLAVORS
3       RETORN 1
4    SI_NO
5       RETORN n*fact(n-1)
6    FI_SI
7 FI_FUNCIO
```

Llistat 1.8: Funció en pseudocodi que avalua el factorial d'un número de manera recursiva.

- La repetició de càlculs innecessaris. Això acostuma a passar gairebé sempre que la funció es crida a si mateixa més d'una vegada.
- El cost de cridar a funció. Una crida a funció implica moltes operacions elementals per a la màquina. En canvi, una estructura iterativa s'implementa amb moltes menys operacions.
- Cada crida a una funció necessita memòria per a totes les seves variables; aquesta memòria no s'allibera fins que la funció no acaba.

No obstant el seu pitjor rendiment, l'ús de funcions recursives pot ser preferible quan l'expressió de l'algorisme és molt més senzilla en versió recursiva que en versió iterativa, i sabem que la recursió no arribarà a molta profunditat.

Considerem per exemple el *problema de les torres de Hanoi* (Figura 1.9). Tenim tres pals A, B, C, i tenim n discos foradats de mides diferents. Inicialment tots són al pal A, ordenats de baix a dalt de més gran a més petit. L'objectiu del joc és passar tots els discos del pal A al pal C, fent servir el pal B com auxiliar. Les regles són que només podem moure un disc alhora, i mai no podem posar un disc a sobre d'un disc més petit.

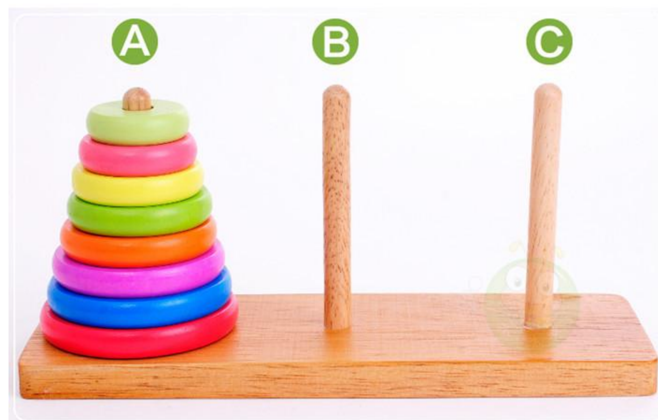


Figura 1.9: El problema de les Torres de Hanoi.

Resoldre el problema per valors petits de n és fàcil. La Figura 1.10 mostra com solucionar-lo per 2 discos en 3 moviments. Podem codificar cada moviment usant el pal origen i el pal destinació. Així, la solució de la Figura 1.10 la podem codificar com AB - AC - BC.

La solució per n discos es pot expressar de manera molt senzilla en termes de la solució per $n-1$ discos. Concretament, per a passar n discos del pal A al pal C fent servir el pal B com a auxiliar, només cal:

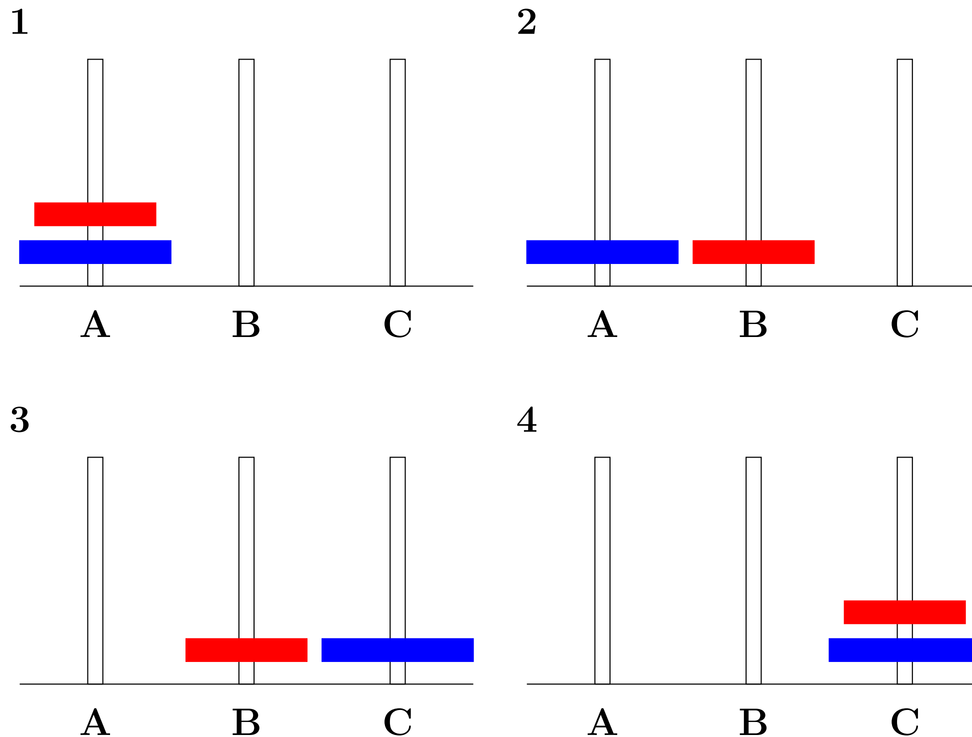


Figura 1.10: Solució del problema de les torres de Hanoi amb 2 discos.

(a) Passar $n-1$ discos del pal A al pal B fent servir el pal C com a auxiliar.

(b) Passar el disc que queda del pal A al pal C.

(c) Passar $n-1$ discos del pal B al pal C fent servir el pal A com a auxiliar.

Aquesta estratègia es pot implementar en una funció recursiva de manera molt fàcil (vegeu l'Exercici 9).

La recursivitat és especialment còmode per codificar algorismes de *backtracking*. Ho il·lustrarem amb el *problema de les vuit reines* (vegeu la Figura 1.11). Es tracta de determinar com col·locar vuit reines en un tauler d'escacs de manera que no n'hi hagi dues en la mateixa fila, columna o diagonal.

A la posició de la figura, la vuitena reina no es pot col·locar a la columna `h`. Provem si hi ha una altra posició possible per a la reina en `g`; com no la trobem, la traiem del tauler i intentem alguna altre posició per a la reina en `f`; com que no n'hi ha, la traiem i provem una altra posició per a la reina en `e`. La posem en `e5`, i tornem a intentar posar una reina a la columna `f`. Procedint així, endavant i endarrere, arribarem a alguna posició vàlida, o a la conclusió que no és possible col·locar-les.

El Llistat 1.12 és un pseudocodi per al problema de les vuit reines. Pressuposa que hem codificat apart les funcions: `LlocSegur()` que retorna sí o no, depenent de si la fila i columna està lliure o està atacada; `CollocaReina()`, per col·locar-la al lloc indicat; i `TreuReina()` per treure-la en cas necessari. Per abreviar, hem posat en una sola instrucció la declaració i el valor inicial de les variables `exit` i `fila`. Això és legal en el llenguatge C.

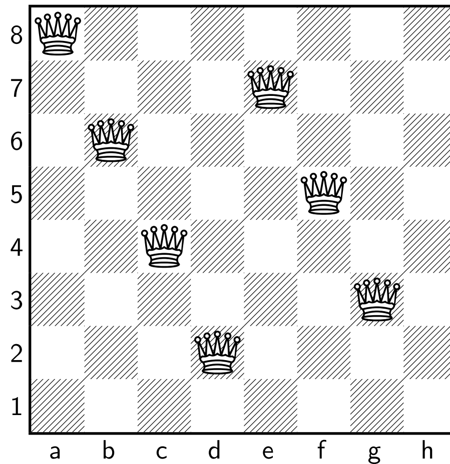


Figura 1.11: Es poden col·locar vuit reines en un tauler d'escacs sense que s'amenacin?

1.4. Algorismes de cerca

Una situació molt comuna a la pràctica és haver de cercar la posició d'un valor donat dins una llista de valors. Un altre situació habitual és haver d'ordenar una llista. Aquests dos problemes ens serviran per seguir introduint idees bàsiques d'algorísmica.

En el cas de la cerca, suposarem que tenim un vector `v`, indexat de 1 a n, contenint un determinat tipus de valor (nombres enters, nombres no-enters, caràcters, cadenes de caràcters, etc), i volem trobar el primer índex `i` tal que `v[i]` és igual a un determinat valor, si existeix.

La manera més senzilla de fer una cerca, i la única raonable si no se sap res sobre la ordenació dels elements del vector, és recórrer el vector des del primer element fins l'últim i comparar cada element amb el valor cercat, fins a obtenir una igualtat o bé arribar al final del vector. Aquest algorisme s'anomena *cerca seqüencial* (o *lineal*).

Està clar que, en el pitjor dels casos, la cerca seqüencial durà a terme n comparacions, si n és la quantitat d'elements del vector. Comptar el nombre d'operacions que fa un algorisme és important per dictaminar la seva eficiència, en comparació amb altres algorismes que resolguin el mateix problema. No és important el número exacte d'operacions (això pot canviar d'un ordinador a un altre i depèn de què entenguem per "operació"), sinó *de quin ordre és* aquesta quantitat en funció de la mida de les dades. Si n és la longitud del vector, el nombre d'operacions, i per tant el temps que es triga a completar la cerca, és proporcional a n. És diu que la cerca seqüencial té una *complexitat* de $O(n)$ [4].

Si el vector està ordenat (per exemple, nombres de més petit a més gran, o noms per ordre alfabètic), aleshores hi ha una estratègia més eficient de cercar, que s'anomena *cerca binària* o *dicotòmica*. En llenguatge ordinari:

1. Es consideren tots els elements del vector.
2. Es compara l'element de mig del vector amb el valor buscat.

[4] Aquesta notació vol dir que el nombre d'operacions està acotat per $C \cdot n$, per alguna constant C.

```
INICI
  PosaReina (1)
FI

FUNCIO PosaReina( enter columna)
  enter exit=0
  enter fila=0
  FER
    fila=fila+1
    SI LlocSegur( fila, columna) LLAVORS
      CollocaReina( fila, columna)
      SI columna<8 LLAVORS
        PosaReina( columna+1)
        SI exit=0 LLAVORS
          TreuReina( fila, columna)
        FI_SI
      SI_NO exit=1
    FI_SI
  FINS QUE exit=1 o fila=8
FI_FUNCIO
```

Llistat 1.12: Pseudocodi per al problema de les vuit reines.

3. Si obtenim igualtat, l'hem trobat. Fi.
4. Si l'element del mig és més gran que el buscat, passem a considerar només els elements de la primera meitat del vector; si no, passem a considerar només els de la segona meitat. Tornem al pas 2.

Als exercicis calcularem la complexitat de la cerca binària i veurem que es inferior a la de la cerca seqüencial.

1.5. Algorismes d'ordenació

Suposem que tenim un vector i volem ordenar els seus elements de més petit a més gran. Suposem també que les úniques operacions que podem fer són comparacions i intercanvis de posició; d'aquesta manera no usarem més memòria de l'ordinador que la que ocupa el propi vector, essencialment.

Hi ha molts algorismes per resoldre aquest problema. Un d'ells és el *mètode de la bombolla*. Es recorre el vector del final al principi, i es comparen parelles d'elements consecutius. Si no estan ordenades, s'intercanvien. En acabar, tenim l'element mínim a la primera posició del vector (ha "surat" fins a aquesta posició, i d'aquí el nom de l'algorisme). Aleshores repetim el procediment considerant els elements del segon a l'últim, i iterem fins que ordenem el dos últims. Vegeu a la Figura 1.13 com actua aquest procediment sobre un vector d'enters de 6 elements.

El mètode de la bombolla requereix fer $O(n^2)$ comparacions, fins i tot quan el vector ja està ordenat d'entrada. Un mètode recursiu d'ordenació fàcil d'implementar i més eficient que el mètode de la bombolla és el *merge-sort* (Llistat 1.14): Consisteix en dividir la llista més o menys per la meitat, ordenar

13,18,5,9,24,7	13,18,7,9,24	13,18,9,24	13,18,24	18,24
13,18,5,9,7,24	13,18,7,9,24	13,18,9,24	13,18,24	18,24
13,18,5,7,9,24	13,18,7,9,24	13,9,18,24	13,18,24	
13,18,5,7,9,24	13,7,18,9,24	9,13,18,24		
13,5,18,7,9,24	7,13,18,9,24			
5,13,18,7,9,24				

Figura 1.13: Ordenació d'un vector de 6 elements mitjançant el mètode de la bombolla.

cada tros, i després barrejar les dues llistes ordenades. Naturalment, cada subllista s'ordena amb el mètode *merge-sort*.

```
FUNCIO merge_sort (enter n, caracter v[1..n])
   enter m
   caracter aux[1..n]

   SI n=1 LLAVORS RETORN
   m=n/2
   merge_sort( m, v[1..m])
   merge_sort( n-m, v[m+1..n])

   aux = v[1..m]
   i=1
   j=m+1
   k=1
   MENTRE i<=m, j<=n FER
     SI v[j]<aux[i] LLAVORS
        v[k+1]=v[j+1]
     SI_NO
        v[k+1]=aux[i+1]
     FI_SI
   FI_MENTRE
   MENTRE i<=m FER
     v[k+1] = aux[i+1]
   FI_MENTRE
FI_FUNCIO
```

Llistat 1.14: Mètode Merge-Sort com a funció en pseudocodi.

Existeixen molts més algorismes d'ordenació, cadascun amb els seus avantatges i els seus inconvenients. Hi ha una pàgina a internet amb simulacions de alguns d'ells, on es veu quins són més ràpid que altres en diverses situacions: http://www.sorting-algorithms.com/

1.6. Complexitat computacional

Encara que les màquines actualment poden fer mil·lions d'operacions per segon, l'eficiència dels algorismes segueix essent una qüestió molt important. Tornant com a exemple als algorismes de cerca, imaginem la situació següent:

La UAB té actualment més de 36 000 estudiants matriculats (entre graus, màsters i cursos d'especialització). Suposem que un administratiu ha de cercar un estudiant particular a la base de dades, a partir del seu NIU. Si la cerca es fa seqüencial, cal accedir als registres de la base de dades un a un i comparar el número buscat amb el número del registre. Encara que en el pitjor dels casos caldrà accedir a tots els registres, podem pensar que en mitjana s'accediria a la meitat (de vegades més, de vegades menys), és a dir a uns 18 000.

Suposem que cada comparació tarda uns 10 mil·lisegons, que és bastant realista si pensem que segurament cal anar a buscar les dades a un servidor remot. Aquesta cerca mitjana tardarà aleshores uns tres minuts, que és un temps inacceptable. Suposant que el temps per fer la comparació baixés a un ml·lisegon, estaríem parlant de 18 segons d'espera per obtenir el registre de l'estudiant en pantalla, que encara és molt.

En cas que es pugui fer cerca binària, els tres minuts d'abans queden reduïts a 15 centèsimes de segon (en el pitjor cas!), que per a l'humà que demana la cerca vol dir que és pràcticament instantània. Més important encara, si el nombre d'estudiants es dupliqués, els tres minuts de mitjana de la cerca seqüencial també es dupliquen, mentre que les 15 centèsimes de la cerca binària passen a ser 16.

El camp de la *complexitat computacional* és l'estudi de l'eficiència dels algorismes pel que fa al temps d'execució i a l'espai de memòria que requereixen. Es comparen algorismes atenent al cost en temps o memòria en funció de la dimensió del problema (la longitud de la llista a cercar en el cas de l'exemple). Normalment, l'anàlisi es fa en base al pitjor cas possible; també s'estudia la complexitat mitjana, però això té molta més dificultat en general.

Quan es parla de "temps" en aquest context, sempre s'està parlant de "quantitat d'operacions". Una particular operació pot tardar més en una màquina que en un altra, i fins i tot una operació "humana" (una suma, un producte, una comparació, ...) pot representar una quantitat d'operacions elementals diferent en una o altra màquina. Però aquesta diferència entre màquines és independent de la dimensió del problema, i es redueix a un factor constant. Per això usem la notació de la "O gran" per denotar la complexitat d'un algorisme. La cerca seqüencial és $O(n)$, és a dir, creix linealment en funció de n. En canvi la cerca binària és $O(\log n)$ i per tant creix com el logaritme: El temps d'execució serà $C \log n$, on C és una constant que només depèn de la màquina (i observeu de pas que la base del logaritme no té importància).

En termes gràfics, és la forma de la funció de n el que importa, i més concretament com creix quan n es va fent gran. Les funcions n^2 i $2n^2$ creixen igual de ràpid, però més ràpid que n; la funció $\exp(n)$ creix més ràpid que qualsevol polinomi.

Per exemple, en el mètode de la bombolla, el nombre exacte de comparacions és $1 + 2 + \cdots + (n-1) = \frac{1}{2}(n^2 - n)$, però podem dir simplement que la seva complexitat és $O(n^2)$.

El mètode d'ordenació merge-sort és $O(n \log n)$. La funció $n \log n$ creix més ràpid que n però més lent que n^p, per qualsevol $p > 1$. Aquest creixement és òptim en el sentit que s'ha demostrat que no existeix cap algorisme general d'ordenació que tingui una complexitat inferior. Això ens porta a parlar de la complexitat computacional d'un problema: El problema de la ordenació té una complexitat computacional d'exactament $O(n \log n)$, perquè hi ha un

algorisme que té aquesta complexitat i no n'hi ha cap que en tingui d'inferior. (Ara bé, el merge-sort necessita $O(n)$ d'espai addicional de memòria.)

Exercicis

1. Dibuixeu un diagrama de flux per al problema del màxim comú divisor de dos enters.

2. Simuleu el funcionament del pseudocodi del Llistat 1.2 (algorisme d'Euclides per calcular el màxim comú divisor de dos enters) sobre algunes dades inicials diferents: Preneu $m = 30$, $n = 12$, i $m = 15$, $n = 8$, per exemple. Simular un algorisme serveix per assegurar-se que fa el que volem que faci. Cal que seguim estrictament allò que està escrit com si fóssim una màquina.

3. Escriviu un pseudocodi per a l'aproximació de $\cos x$ mitjançant el polinomi de Taylor de grau $2n$:

$$\sum_{k=0}^{n}(-1)^k \frac{x^{2k}}{(2k)!}$$

(a) usant només les estructures algorísmiques primàries: seqüencial, alternativa i iterativa;

(b) canviant l'estructura iterativa per una estructura repetitiva.

Se suposa que x i n són dades que cal llegir, i el resultat cal escriure'l.

4. Escriviu l'algorisme d'Euclides en Python, traduint el pseudocodi del Llistat 1.2.

5. Escriviu en pseudocodi funcions que facin els càlculs següents:

(a) La norma euclidiana d'un vector: $\|v\|_2 = (\sum_{i=1}^{n} v_i^2)^{1/2}$

(b) La norma del màxim d'un vector: $\|v\|_\infty = \max_{i=1,\ldots,n} |v_i|$

(c) El producte d'una matriu $n \times m$ per una matriu $m \times p$: $c_{i,j} = \sum_{k=1}^{m} a_{i,k} b_{k,j}$

6. Un polinomi

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

es pot expressar també com

$$a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x a_n) \cdots)$$

(a) Comproveu que el nombre d'operacions necessàries (sumes més productes) per a avaluar un polinomi de grau n mitjançant la segona expressió (coneguda com *regla de Horner*) és $2n$.

(b) Calculeu el nombre d'operacions necessàries per avaluar el polinomi amb la primera expressió i compareu-lo amb el resultat anterior. És un dels mètodes substancialment més eficient (té menys complexitat) que l'altre?

7. Definim la successió $\{F_i\}_{i=1}^{\infty}$ dels *nombres de Fibonacci mòdul* m recurrentment com

$$F_1 = 1,\ F_2 = 1;$$
$$F_i = (F_{i-1} + F_{i-2}) \mod m, \quad \text{per } i \geq 3\ .$$

(a) Escriviu en pseudocodi una funció iterativa anomenada `fibit(n)` i una funció recursiva anomenada `fibrec(n)` que retornin el terme n–èsim de la successió de Fibonacci mòdul $m = 10$.

(b) Programeu en Python les dues versions i feu execucions en el SageMath prenent mesures de temps per diferents valors de n.

Per mesurar el temps d'execució d'un programa en Python, es pot usar la funció `clock()`, del mòdul `time`, que retorna el temps en segons des que hem iniciat la sessió i que el SageMath realment ha usat:

```
from time import clock
StartTime = clock()
<codi a mesurar>
print clock() - StartTime
```

8. (a) Escriviu una funció en pseudocodi que implementi l'algorisme de cerca binària sobre un vector de nombres reals, ordenats de més petit a més gran.

(b) Quantes comparacions caldrà fer en el pitjor dels casos? De quin ordre és la complexitat de l'algorisme? (Aquesta part és més difícil de fer rigorosament; però es pot intuir per on va el resultat.)

9. Escriviu en pseudocodi una funció `hanoi` de l'algorisme recursiu de l'Apartat 1.3 per al problema de les torres de Hanoi. Recordeu que es tracta d'un algorisme recursiu en què, per moure n discos del pal A al pal C, cal moure $n-1$ discos del pal A al B, un disc del A al C, i $n-1$ discos del B al C. La funció es cridarà d'aquesta manera:

```
hanoi( n, A, B, C)
```

Cal que l'algorisme escrigui tots els moviments, disc a disc. Per escriure un moviment, es pot simplement imprimir la lletra de sortida i la lletra d'arribada.

10. Quina seqüència de números escriurà la funció recursiva següent si li passem el valor $n = 1$?

```
FUNCIO Exercici( enter n)
   escriu( n)
   SI n<3 LLAVORS
      Exercici( n+1)
   FI_SI
   escriu( n)
FI_FUNCIO
```

11. Suposem que la funció `suma()` pren dos arguments numèrics i retorna la seva suma. Suposem que la funció `resta()` també pren dos arguments numèrics i retorna la diferència del primer menys el segon. Quin resultat retorna l'expressió següent?

```
suma( suma( a,b), resta( a,b)) - resta( suma(a,b) - resta(a,b))
```

2

Codificació de la informació

2.1. Quantificació de la informació

La unitat mínima d'informació en els ordinadors és el *bit*. Un bit és qualsevol sistema físic que tingui dos estats (sí/no, encès/apagat, etc). A l'interior de l'ordinador tot es redueix a corrents elèctrics, però podem pensar que la informació es guarda i es transmet en petites caixetes que contenen un dels dos estats. Per comoditat, aquests dos estats els denotem per 0 i 1.

La quantitat d'informació es mesura en múltiples d'unitats bàsiques, habitualment en múltiples de bits o en múltiples de bytes. Un *byte* és un grup de 8 bits, i per tant permet representar $2^8 = 256$ estats diferents.

Per als múltiples de les unitats bàsiques (bit, byte), s'usen els prefixos kilo (K), mega (M), giga (G), tera (T), i peta (P), com a les unitats físiques. A diferència de les unitats físiques, en canviar de prefix multipliquem per $1024 = 2^{10}$, en lloc de $1000 = 10^3$, perquè ens va millor treballar amb múltiples de 2 en lloc de múltiples de 10. Així:

$$
\begin{aligned}
1 \text{ byte} &= 1 \text{ B} \\
1 \text{ kilobyte} &= 1 \text{ KB} = 2^{10}\text{B} \\
1 \text{ megabyte} &= 1 \text{ MB} = 2^{10} \text{ KB} = 2^{20}\text{B} \\
1 \text{ gigabyte} &= 1 \text{ GB} = 2^{10} \text{ MB} = 2^{30}\text{B} \\
1 \text{ terabyte} &= 1 \text{ TB} = 2^{10} \text{ GB} = 2^{40}\text{B} \\
1 \text{ petabyte} &= 1 \text{ PB} = 2^{10} \text{ PB} = 2^{50}\text{B}
\end{aligned}
$$

Per motius comercials, els fabricants de discs durs usen els prefixos referint-se a les potències de 10. Per exemple, un disc comercialitzat com de 500 GB, en realitat en tindrà uns 466, segons les definicions anteriors.

Es pot parlar també de bits, kilobits, megabits, etc. Les abreviatures són b, Kb, Mb, ..., respectivament. Noteu que la `b` cal escriure-la en minúscula. Freqüentment això no es respecta en la publicitat de productes i serveis.

Últimament es tendeix a substituir KB per KiB, MB per MiB, etc, per posar èmfasi en què el factor és 2^{10} en lloc de 10^3. Aquí seguirem usant la notació tradicional.

Per mesurar la velocitat de transmissió d'informació s'utilitzen els *bits per segon* (bps) i els seus múltiples (Kbps, Mbps, etc). Quan diem que "tinc un ADSL de 20 megues", o "tinc una fibra òptica de 100 megues", volem dir que

la velocitat a la que descarreguem informació d'internet és de (com a màxim) 20 Mbps (respectivament, de 100 Mbps).

2.2. Representació en base b

Sabem que tot nombre real es pot representar com a suma de les seves xifres (enters entre 0 i 9) multiplicades per potències de 10, més el seu signe (positiu o negatiu). Per exemple,

$$-193.625 = -\Big(1 \cdot 10^2 + 9 \cdot 10^1 + 3 \cdot 10^0 + 6 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}\Big)$$

En general, la suma és infinita perquè poden haver-hi infinits decimals. Si denotem per a_j les xifres, entre 0 i 9, el nombre real $x = \pm a_0 a_1 a_2 \cdots a_p . a_{p+1} a_{p+2} \cdots$ es pot escriure

$$x = \pm\Big(\sum_{j=0}^{\infty} a_j 10^{-j}\Big) \cdot 10^p$$

El número 10 que fa de base de les potències no té res d'especial, i pot substituir-se per qualsevol altre enter més gran que 1:

Teorema. *Fixem $b \in \mathbb{N}$, $b \geq 2$. Tot $x \in \mathbb{R}$, $x \neq 0$, pot ser representat en la forma*

$$x = s\Big(\sum_{j=0}^{\infty} a_j b^{-j}\Big) \cdot b^p \ , \tag{2.1}$$

amb $s \in \{+, -\}$, $p \in \mathbb{Z}$ i $a_j \in \{0, 1, \ldots, b-1\}$.
A més, aquesta representació és única si imposem que $a_0 \neq 0$ i que els a_j no siguin tots $b-1$ d'una posició en endavant, i.e.

$$\forall j_0 \in \mathbb{N},\ \exists j \geq j_0 :\ a_j \neq b - 1.$$

Aquest teorema dóna l'anomenada *representació en punt flotant (o "coma flotant") normalitzada* d'un nombre real. A l'expressió (2.1),

- s és el *signe*,
- $m = \Big(\sum_{j=0}^{\infty} a_j b^{-j}\Big)$ és la *mantissa*,
- b és la *base*, i
- p és el *exponent*.

Alguns exemples amb $b = 10$:

$$\begin{aligned} 1/3 &= +(3.33333\cdots) \times 10^{-1} \\ -10/3 &= -(3.33333\cdots) \times 10^{0} \\ 123.456 &= +(1.23456) \times 10^{2} \\ -0.00034 &= -(3.4) \times 10^{-4} \ . \end{aligned}$$

La representació en punt flotant és similar a la notació científica que empren habitualment les calculadores per escriure números molt grans o molt propers a zero. El punt decimal "flota" i es col·loca després (o de vegades abans) del primer *dígit significatiu.* S'entén per dígits significatius els que queden després de treure els zeros de davant i de darrera del número. Per exemple, -0.00034 es representa en notació científica com 3.4E–4.

Les bases més habituals que es fan anar en informàtica són $b = 2$ i $b = 16$. La base $b = 8$ també s'utilitza, però molt menys. A treballar en una determinada base també se li diu usar un cert *sistema de numeració*:

- Sistema *binari* ($b = 2$). Els dígits (xifres) binaris són 0 i 1. Un dígit binari és doncs un bit. D'aquí ve el seu nom: **b**inary dig**it**).
- Sistema *hexadecimal* ($b = 16$). Es pot considerar una compactació del binari, en el qual cada 4 dígits binaris es converteixen en un d'hexadecimal. És molt convenient des del punt de vista humà. Els dígits que es fan servir són $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}$.
- Sistema *decimal*: És el nostre sistema habitual, amb dígits $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

És convenient saber canviar de base, entre les bases 2, 10 i 16. Per passar de qualsevol base b a base 10 només cal usar l'expressió (2.1) del teorema 2.2, fer les operacions, i expressar el resultat de la forma habitual. Si a més es vol la representació normalitzada, caldrà moure convenientment el punt decimal. Per exemple, considerem el número representat en base 2 per 110.1011 (o, en representació normalitzada, 1.101011×2^2). Obtenim:

$$1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} = 6.6875\ .$$

Passar de base 10 a base b és una mica més feixuc. Ho fem també per $b = 2$, però és evident com es faria en qualsevol altre base. En tot cas, cal considerar per separat la part entera i la part decimal.

- Per a convertir un nombre enter representat en base 10 a la seva representació en base 2, cal pensar una operació que ens permeti obtenir els seus dígits. Si els volguéssim en base 10, estar clar que els podem obtenir, de menys a més significatiu, fent mòdul 10, dividint després per 10 i repetint el procés. Per obtenir els dígits en base 2, fem el mateix: Per exemple, per representar 123 en binari, farem

$$\begin{array}{rcll} 123 \div 2 & = & 61 & \stackrel{\text{residu}}{\longrightarrow} \boxed{1} \\ 61 \div 2 & = & 30 & \longrightarrow \boxed{1} \\ 30 \div 2 & = & 15 & \longrightarrow \boxed{0} \\ 15 \div 2 & = & 7 & \longrightarrow \boxed{1} \\ 7 \div 2 & = & 3 & \longrightarrow \boxed{1} \\ 3 \div 2 & = & 1 & \longrightarrow \boxed{1} \\ 1 \div 2 & = & 0 & \longrightarrow \boxed{1}\,, \end{array}$$

  i per tant $123 = (1111011)_2$.
  (Quan treballem amb diferent representacions a la vegada, millor anar marcant la base d'alguna manera, com ara amb un subíndex.)
- En el cas d'un nombre només amb part decimal, obtindrem els seus dígits en base 2 multiplicant per 2, prenent part entera i repetint el procés. Així, per exemple, per representar 0.1 en base 2, farem

$$\begin{array}{rrcll} & 0.1 \times 2 & = & 0.2 & \stackrel{\text{part entera}}{\longrightarrow} \boxed{0} \\ \stackrel{\text{part decimal}}{\longrightarrow} & 0.2 \times 2 & = & 0.4 & \stackrel{\text{part entera}}{\longrightarrow} \boxed{0} \\ \stackrel{\text{part decimal}}{\longrightarrow} & 0.4 \times 2 & = & 0.8 & \stackrel{\text{part entera}}{\longrightarrow} \boxed{0} \\ \stackrel{\text{part decimal}}{\longrightarrow} & 0.8 \times 2 & = & 1.6 & \stackrel{\text{part entera}}{\longrightarrow} \boxed{1} \\ \stackrel{\text{part decimal}}{\longrightarrow} & 0.6 \times 2 & = & 1.2 & \stackrel{\text{part entera}}{\longrightarrow} \boxed{1} \end{array}$$

Com que la part decimal de 1.2 és 0.2, que ja ha sortit, es tornarà a repetir el grup 0011 ad infinitum. Per tant, $(0.1)_{10} = (0.0\widehat{0011})_2$. Noteu que la representació decimal de 0.1 és finita, mentre que la seva representació binària és periòdica.

- Ajuntant els dos exemples anteriors, veiem que

$$123.1 = (1111011.0\widehat{0011})_2 \ .$$

El mecanisme funciona igual en qualsevol base.

El pas del sistema binari a l'hexadecimal i viceversa és molt senzill. Només cal agrupar els dígits binaris de quatre en quatre cap a la dreta i cap a l'esquerra del punt base, i convertir cada grup, amb la pauta

$$\begin{array}{ccc}
0000 & \longleftrightarrow & 0 \\
0001 & \longleftrightarrow & 1 \\
0010 & \longleftrightarrow & 2 \\
0011 & \longleftrightarrow & 3 \\
0100 & \longleftrightarrow & 4 \\
0101 & \longleftrightarrow & 5 \\
0110 & \longleftrightarrow & 6 \\
0111 & \longleftrightarrow & 7 \\
1000 & \longleftrightarrow & 8 \\
1001 & \longleftrightarrow & 9 \\
1010 & \longleftrightarrow & \mathrm{A} \\
1011 & \longleftrightarrow & \mathrm{B} \\
1100 & \longleftrightarrow & \mathrm{C} \\
1101 & \longleftrightarrow & \mathrm{D} \\
1110 & \longleftrightarrow & \mathrm{E} \\
1111 & \longleftrightarrow & \mathrm{F}
\end{array}$$

Per exemple: El nombre que en el sistema decimal es representa 75.28125, en binari és $(1001011.01001)_2$, i en hexadecimal és $(4\mathrm{B}.48)_{16}$.

2.3. Codificació de nombres en punt flotant

Els ordinadors guarden internament els nombres reals en base 2 usant els dos estats de cada bit, que representen el 0 i el 1. El nombre no és guarda exactament, en general, donat que només pot ocupar una quantitat finita de bits:

$$x \approx s\left(\sum_{j=0}^{t} a_j 2^{-j}\right) \cdot 2^p = \pm(a_0.a_1a_2\cdots a_t) \cdot 2^p \ ,$$

La mantissa ha quedat reduïda a $t+1$ dígits, i l'exponent p també haurà d'estar limitat a un cert rang $p_{\min} \le p \le p_{\max}$. Tant el nombre de dígits de la mantissa com el rang d'exponents es trien segons un compromís entre la varietat de nombres que volem representar i la quantitat de memòria que cada representació ocupa.

Observeu que la quantitat de bits usats per representar la mantissa determina la *precisió* amb què podem aproximar el nombre real que estem representant. Per altra banda, la quantitat de bits que destinem a guardar p determinarà el *rang* de nombres que podrem representar.

La manera concreta com es guarda el nombre pot variar d'un ordinador a un altre, però la majoria segueixen el standard IEEE[1], que veurem a continuació.

- El format *punt flotant de precisió simple* (IEEE-simple) usa 32 bits, dels quals un és per al signe (0 indica +, 1 indica −), 8 són per a l'exponent, i la longitud de la mantissa és de $t = 24$ bits, però només es guarden explícitament 23: Al usar-se la representació normalitzada, la primera xifra en base 2 és sempre $a_0 = 1$, i per tant no cal guardar-la. Els vuit bits destinats a l'exponent codifiquen en binari un nombre entre 0 i 255, que s'interpreta com l'exponent que volem + 127. Vegeu la Figura 2.1.

IEEE simple	s (1)	$e = p + 127$ (8)	Mantissa (23)
IEEE doble	s (1)	$e = p + 1023$ (11)	Mantissa (52)

Figura 2.1: Distribució a la memòria del signe, mantissa i exponent per als formats de punt flotant. S'indica entre parèntesis la longitud de cada part en bits.

- El format *punt flotant de precisió doble* (IEEE-doble) és similar, usant 64 bits, amb: un per al signe, 11 per a l'exponent, i mantissa de $t = 53$ bits, dels quals es guarden 52. Els onze bits destinats a l'exponent codifiquen en binari un nombre entre 0 i 2047, que s'interpreta com l'exponent que volem + 1023. Vegeu la Figura 2.1.

Els formats IEEE-simple i IEEE-doble es corresponen als tipus `float` i `double` del llenguatge C, respectivament.

2.4. Codificació de nombres enters

2.4.1 Enters positius

Considerem primer nombres enters no negatius $x \in \mathbb{N}$. En aquest cas, només cal convertir el nombre a sistema binari i guardar els seus bits, completant amb zeros a l'esquerra, segons la quantitat de bytes que usem.

A nivell de processador és incòmode que la quantitat de bytes que ocupa un enter depengui de la magnitud del número. El que es fa és treballar amb uns quants "tipus enters" de longitud prefixada. Típicament, de 2, 4 i 8 bytes, o de 4, 8 i 16 bytes.

Al guardar els bytes en la memòria, hi ha dues opcions (i això també s'aplica a les representacions en punt flotant):

- Guardar els bytes d'esquerra a dreta. Això és el que fan les màquines "*big-endian*".
- Guardar els bytes de dreta a esquerra. Això és el que fan les màquines "*little-endian*".[2]

[1] Institute of Electric and Electronic Engineering

[2] En *Els Viatges de Gulliver*, de Jonathan Swift, es descriuen les tensions internes de

Per exemple, el número 123, guardat en un enter de 2 bytes, en una màquina big-endian, té l'aspecte

0	0	0	0	0	0	0	0		0	1	1	1	1	0	1	1

mentre que en una little-endian serà

0	1	1	1	1	0	1	1		0	0	0	0	0	0	0	0

Les màquines basades en processadors Intel o AMD són little-endian; per tant, la majoria d'ordinadors personals actuals ho són. Els MacIntosh antics que van amb processadors Motorola són big-endian. En general, no ens ha de preocupar massa aquest aspecte, perquè això queda normalment transparent per al programador. Només en alguns casos especials té importància, per exemple al escriure o llegir fitxers de formats gràfics (bmp, jpeg, etc).

2.4.2 Enters negatius

Considerem ara la representació d'enters negatius. Una possibilitat seria reservar un bit per al signe, com es fa a la representació en punt flotant de nombres amb decimals. Però aquesta és una estratègia dolenta per a nombres enters, perquè complicaria innecessàriament les operacions elementals a nivell de hardware.

Per a representar nombres negatius, el que es fa es representar en sistema binari el seu *complement a dos*. El complement a dos d'un número negatiu es calcula com segueix:

(a) Es representa el número sense el signe en binari.

(b) S'intercanvien els uns per zeros i els zeros per uns.

(c) Se suma 1 al número obtingut.

Per exemple, per representar -13 com a enter de 1 byte, fem:

(a) Representem $+13$:	0	0	0	0	1	1	0	1
(b) Intercanviem els bits:	1	1	1	1	0	0	1	0
(c) Li sumem 1:	1	1	1	1	0	0	1	1

L'avantatge d'aquesta representació és que, per sumar un nombre positiu i un de negatiu, se sumen com si tots dos fossin positius. Per exemple, per sumar 79 i -13, representats com enters d'un byte, sumarem $(01001111)_2$ i $(11110011)_2$, i el resultat és $(101000010)_2$. Ignorant el bit de més que apareix a l'esquerra, obtenim [0|1|0|0|0|0|1|0], que és la representació de $66 = 79 - 13$.

Amb aquesta regla,

- En un enter de t bits es poden guardar els nombres enters des de -2^{t-1} fins a $2^{t-1} - 1$.
- Els nombres negatius i els positius es poden distingir a partir del bit de més a l'esquerra.

Per exemple, amb enters de 1 byte, el nombre més gran que es pot representar és

Lilliput, entre els partidaris de menjar els ous durs començant pel costat gran de l'ou (els big-endians) i els de començar pel costat petit (els little-endians). D'aquí venen els noms en informàtica.

0	1	1	1	1	1	1	1	= +127

i el negatiu més gran

1	0	0	0	0	0	0	0	= -128

2.5. Codificació de text

Els caràcters que podem voler guardar en un fitxer són molt variats, i això posa un problema. Tradicionalment, cada caràcter es codifica en un byte. De fet, amb 7 bits es poden codificar 128 caràcters diferents, i això és suficient per guardar totes les lletres de l'alfabet llatí (majúscules i minúscules, però sense accents), els números, i alguns signes de puntuació comuns. I encara queda lloc per alguns caràcters "no imprimibles", com ara salts de línia, tabuladors, marques de fi de fitxer, etc.

La manera de codificar aquests caràcters més bàsics està ben establerta i és coneguda com el *codi ASCII* [3]. Tots els ordinadors i sistemes utilitzen aquesta codificació. Podeu trobar les taules a internet (e.g. a la Wikipedia). La taula ASCII relaciona cada caràcter amb el número de 0 a 127 que el representa en binari.

Els caràcters 0 al 31 i el caràcter 127 són "no imprimibles", el números van del 48 al 57, les lletres majúscules del 65 al 90, i les minúscules del 97 al 122.

El problema comença al voler introduir lletres accentuades, símbols més estranys, i altres alfabets. Havent-hi encara un bit lliure, no hi hauria problema per encabir 128 caràcters més, però això no s'ha fet de manera estàndard i conviuen diversos convenis. Aquest és el motiu pel qual els programes que tracten amb fitxers de text (com ara editors i compiladors) han de saber quina és la codificació que s'ha usat, per poder tractar-los correctament.

Diversos fabricants van fer extensions diferents del codi ASCII, amb moltes variants adaptades a llengües diferents. Entre ells, IBM, Apple i DEC. Finalment, la codificació coneguda com a ISO-8859 ha anat imposant-se. La variant ISO-8859-1, també coneguda com a Latin1, afegeix els símbols de les llengües europees occidentals; la ISO-8859-2 és adequada per a les llengües europees orientals; la ISO-8859-5 conté l'alfabet ciríl·lic, etc. La codificació coneguda com a Windows-1252 és gairebé igual a la Latin1 (només afegeix caràcters a llocs buits) i no hauria de donar problemes de compatibilitat amb ella.

Tot i així, segueix havent-hi el problema d'identificar la codificació concreta, i el fet que és impossible escriure en un mateix fitxer un text amb trossos d'alfabets molt diferents. Per evitar això s'han desenvolupat codificacions multi-byte; els caràcters ASCII segueixen ocupant un sol byte, però els demés caràcters en poden ocupar fins a quatre. La codificació més estesa actualment amb aquesta idea és la UTF–8. Codifica 1 112 064 caràcters diferents, i per tant pot encabir gairebé tot el que vulguem escriure en qualsevol idioma.

El problema de distingir codificacions persisteix, perquè ara els sistemes tipus Linux (inclòs Mac OSX) tendeixen a afavorir el UTF-8, mentre que en els programes de Windows és més habitual encara usar el ISO-8859.

[3] American Standard Code for Information Interchange.

Exercicis

12. Un ordinador personal actual pot portar de l'ordre de 8–16 GB de memòria RAM, i un disc dur de 500–1000 GB. Un CD-ROM té una capacitat de 650-800 MB. Un DVD, uns 4.7 GB si és de capa simple i uns 8.7 GB si és de capa doble. Un Blu-Ray de capa simple té capacitat per 25 GB, i de capa doble per 50 GB.

(a) Quan es va inventar el CD-ROM ($\sim$ 1985), els discs durs dels ordinadors personals tenien una capacitat d'uns 10 MB. Si tenim ara un disc dur de 640 GB, i suposant que aquestes capacitats són exactes, quantes vegades hi cap *exactament* el disc antic en l'actual?

(b) Quants cops hi cap un CD-ROM de 800 MB en un Blu-Ray de capa doble?

13. Els primers *mòdems* (aparells per transmetre i rebre informació a través de la línia analògica telefònica) transmetien a 300 bps. A finals dels anys 1990's era normal tenir un mòdem de 28800 bps, i més tard de 56 Kbps, fins que van ser substituïts per la tecnologia ADSL i els routers.

(a) Quant tardaríem a descarregar una pel·lícula de 700 MB amb un mòdem de 28800 bps, suposant que sempre anés a velocitat màxima?

(b) Els proveïdors d'internet (*Internet Service Providers*, ISP) anuncien velocitats de ADSL de l'ordre de 20 Mpbs. Però cal tenir en compte que això és la velocitat de baixada/descàrrega (*download*); la velocitat de pujada/càrrega (*upload*) és sempre molt menor (al voltant de 1 Mbps).[4]

Quan tardarem a pujar una col·lecció de 20 fotos fetes amb una càmera de 3 megapíxels a un servidor d'intercanvi de fitxers?
Més dades: 1 megapíxel = 2^{20} píxels, i cada pixel ocupa 24 bits. Estem suposant que la foto no està comprimida; en un format comprimit (JPEG, PNG) pot ocupar de 5 a 10 vegades menys. Suposem també que el servidor al que connectem no redueix la nostra capacitat de càrrega.

(c) Hi ha llocs web que ens permeten mesurar la velocitat de la nostra connexió a internet. Per exemple, `http://www.speedtest.net`. Connecteu-vos-hi, espereu fins que aparegui el botó "begin test" i premeu-lo. Obtindreu una estimació de les velocitats de càrrega i descàrrega de la vostra connexió en aquell moment.[5]

14. En notació matemàtica tradicional, de *punt fix*, un nombre ve representat per una cadena de xifres (de longitud qualsevol), i la posició de l'anomenat "punt decimal" (en base 10) o "punt base" (més en general)[6] s'indica col·locant-hi un senyal explicit (un punt o una coma). Si no hi ha punt base se suposa que està a la dreta de tot, i per tant el número és enter.

Escriviu en la representació de punt flotant normalitzada en base 10:

(a) La velocitat de la llum, 299792458 ms^{-1} .

[4] Els límits teòrics d'aquesta tecnologia són 24 Mbps i 3.3Mbps, respectivament.

[5] Des de la UAB, connectats amb cable aconseguireu velocitats d'entre 80 i 100 Mbps, tant de baixada com de pujada. A través de la wifi serà segurament menys de la meitat.

[6] *radix point*, en anglès

(b) El número de Avogadro, $6.02214129 \times 10^{23}$ mol^{-1} .

(c) Una atmosfera en Pascals, 101325 Pa, i en hectoPascals (hPa).

(d) La massa d'un electró, expressada en unitats de massa atòmica: 0.00054857990946 u.

(e) El percentatge aproximat de població mundial que parla català: 0.16 %.

15. (a) Trobeu la codificació com a IEEE-simple del nombre 1/10.
Dividiu els 32 bits en els 4 bytes corresponents i convertiu el contingut de cada byte a hexadecimal.
Suposeu que l'ordinador és "little-endian" i feu la permutació convenient de bytes.

El que heu obtingut és "l'aspecte" que té el nombre 1/10 dins la memòria de l'ordinador quan està contingut en una variable de C de tipus `float`.

(b) Podeu usar el programa següent en C per determinar si és correcte el que heu contestat a l'apartat anterior, i per practicar amb altres nombres:

```
#include <stdio.h>

int main () {
  float x;
  int i;
  unsigned char* c = (unsigned char*) (&x);
  scanf( "%f", &x);
  printf("Big-endian: ");
  for (i=3; i>=0; i--) {
    printf("%02X ", *(c+i));
    }
  printf("\nLittle-endian: ");
  for (i=0; i<=3; i++) {
    printf("%02X ", *(c+i));
    }
  return 0;
  }
```

16. (a) En C, declarem els enters amb signe usant les expressions
`int`, `short int`, `long int`, `long long int`.
La longitud en bytes de cadascun d'ells depèn de la màquina i el compilador de C instal·lat.

Per conèixer la vostra instal·lació feu un programa en C que apliqui la funció `sizeof()` a cada tipus (per exemple, `sizeof(int)`). Aquesta funció retorna la quantitat de bytes que ocupa el tipus en qüestió. Observeu que `int` coincideix amb algun dels altres tipus.

(b) Observeu què passa quan a una variable que conté el valor $2^{t-1} - 1$ en un enter amb signe de t bytes se li suma una unitat.

(c) Observeu què passa quan a una variable que conté el valor zero en un enter sense signe de t bytes se li resta una unitat. (Els tipus sense signe usen les mateixes declaracions que amb signe, precedides de la paraula `unsigned`, i s'imprimeixen amb `"%u"`.)

17. (a) Trobeu la codificació del número 2147 com a enter de 4 bytes (tipus `long int` en C).
Convertiu cada byte a hexadecimal.
Permuteu els bytes per obtenir la representació interna en una màquina little-endian.

(b) Comproveu si el vostre ordinador és little-endian o big-endian. Podeu usar aquest programa en C:

```
#include <stdio.h>

int main() {
  int a = 0x12345678;
  unsigned char *c = (unsigned char*)(&a);
  if (*c == 0x78)
    printf("little-endian\n");
  else
    printf("big-endian\n");
  return 0;
  }
```

Essencialment, aquest programa guarda en memòria l'enter hexadecimal de quatre bytes 12 34 56 78. Després obté l'adreça de memòria en què està guardat el primer byte i el llegeix. Si el resultat és 78, aleshores la màquina és little-endian. Per entendre el programa, heu d'haver estudiat el tema d'*apuntadors* en C.

18. Expliqueu matemàticament per què funciona la representació en complement a 2 dels nombres negatius per tal de sumar nombres positius i negatius com si tots fossin positius.

19. (a) Quin nombre ve representat pel següent enter amb signe de dos bytes?

1	1	1	1	1	1	1	1		0	1	0	0	0	0	1	0

(b) Comproveu que el vostre ordinador efectivament emmagatzema els nombres negatius seguint aquestes regles. Podeu usar el programa següent:

```
#include <stdio.h>

int main() {
  short int a = -2;
  unsigned char *c = (unsigned char*)(&a);
  printf("%X ", *c);
  c = c+1;
  printf("%X ", *c);
  return 0;
  }
```

Aquí llegim un nombre negatiu (-2) i el posem en un enter de dos bytes. Després escrivim els dos bytes per separat i en hexadecimal. Atenció a la "endianness"!

20. (a) Passeu a base 2, i de base 2 a base 16, el nombre 654.321, amb dues xifres hexadecimals a la dreta del punt.

(b) Trobeu la representació binària en punt flotant de 123.456, amb una mantissa de 12 posicions (binàries).
Transformeu el nombre que heu trobat novament a base 10. Observeu la pèrdua de precisió en el nombre resultant.

(c) Per al cas del tipus `float` del C (mantissa de 23 bits), quin és l'error màxim que es comet al representar un nombre real qualsevol. Quantes xifres en base 10 es pot considerar que són bones per un número guardat així?
I amb el tipus `double` (mantissa de 52 bits)?

21. Desxifreu el missatge següent:

486f6c612c20636172616c6c6f7421

22. El vídeo d'alta definició es reprodueix a 30 fps (fotogrames per segon), i cada fotograma té una resolució de 1920 per 1080 píxels, amb 24 bits per píxel.

És possible veure aquest vídeo en streaming a través d'una connexió

(a) USB 2.0? (velocitat màxima 408 Mbps)

(b) USB 3.0? (velocitat màxima 5 Gbps)

(c) Asymmetric Digital Subscriber Line (ADSL) a 20 Mbps de descàrrega?

(d) fibra òptica a 300 Mbps?

23. Un disc dur té 200 GB disponibles i està rebent dades a 15 Mbps. Quant temps tardarà a omplir-se el disc?

24. La següent funció en Python pretén escriure els números 0.1, 0.2, ..., 1.0 i retornar. Quin problema preveieu que tindrà?

```
def decimes():
  x=0
  while (x!=1):
    x = x + 1./10
    print (x)
  return
```

25. Per codificar àudio digitalment, es mostreja a intervals regulars l'amplitud d'ona de la font de so. Per obtenir la qualitat d'un Compact Disc de música, es prenen 44 100 mostres per segon, i cada mostra es codifica en 32 bits. Quants MB ocupa una hora de música?

Nota: La música codificada en MP3 típicament ocupa una desena part, amb el consegüent estalvi en emmagatzemar i transmetre. Això s'aconsegueix reduint la precisió dels elements considerats menys audibles, de manera que la majoria de la gent no nota la diferència.

3

Maquinari (hardware)

3.1. Estructura d'un ordinador: arquitectura de von Neumann

John von Neumann va ser un dels grans matemàtics del segle XX. Se li atribueix la concepció de l'estructura d'un ordinador, coneguda com a *arquitectura de von Neumann*, que encara és present en els ordinadors actuals. Pero cal donar també crèdit als seus col·laboradors John Eckert (enginyer elèctric) i John Mauchly (físic), que ja havien treballat força en el tema abans que von Neumann s'hi posés.

D'acord amb l'arquitectura de von Neumann, un ordinador està dividit en diverses *unitats funcionals*:

- *Unitats d'entrada*: dispositius a través dels quals s'introdueixen dades i instruccions, que es transformen en senyals elèctrics. Un ordinador n'acostuma a tenir diverses (exemples: teclat, ratolí, escàner, lector d'empremtes digitals, videocàmera, etc).
- *Unitats de sortida*: dispositius en què es mostren resultats dels programes executats. En aquests dispositius, els senyals elèctrics es transformen en informació perceptible per l'usuari (exemples: pantalla, altaveu, impressora).
- *Memòria interna* o *principal*: dispositiu on s'emmagatzemen tant les dades com les instruccions durant l'execució de programes. Un programa, abans de ser executat, ha d'estar carregat a la memòria interna.
  - La major part de la memòria interna és de tipus RAM[1] (*Random Access Memory*). El contingut d'aquesta memòria es pot llegir i escriure. Es pot pensar com una tira llarga de bytes, numerats amb enters a partir de zero. El número d'ordre d'un byte és la seva *adreça*.
    La memòria RAM és volàtil: tot el seu contingut es perd en apagar l'ordinador.
  - Una petita part, que conté els programes d'arrencada de l'ordinador i algunes dades més, essencials per al seu funcionament, és de tipus ROM (*Read-Only Memory*). El contingut d'aquesta memòria,

[1] Els noms DRAM, SDRAM, RDRAM, DDR-SDRAM, i similars només fan referència a diferents tecnologies per realitzar físicament la RAM.

antigament, "venia de fàbrica" en hardware i no es podia alterar. Actualment s'utilitza la tecnologia de "memòria flash" (vegeu més endavant) i hi ha procediments per anar-la actualitzant.
La ROM no és volàtil. El seu contingut roman inalterat quan s'apaga l'ordinador; no necessita alimentació elèctrica.

- *Memòria externa* o *secundària* o *d'emmagatzematge massiu*: El seu contingut és manté sense alimentació elèctrica, i es pot extreure sense que l'ordinador deixi de ser funcional. N'hi ha de tipus molt diferents. Els veurem a l'Apartat 3.3.
- *Unitat Central de Procès* (CPU, *Central Processing Unit*). Familiarment, el "processador" el "microprocessador" o el "micro". Consta de:
  - El *Decodificador d'Instruccions* (*Instruction Decoder*), que decodifica i fa executar les instruccions dels programes.
  - La *Unitat Aritmètico–Lògica* (ALU, *Arithmetic–Logic Unit*), que executa les operacions dictades pel decodificador d'instruccions.
  - Els *Registres* (*Registers)*: Formen una petita memòria; cada registre pot contenir una dada. Els operands sobre els que la ALU fa els càlculs han d'estar en els registres, i el resultat de les operacions també queda dipositat en ells.
  - La *Unitat de Gestió de Memòria* (*Memory Management Unit*: És la interfície amb la memòria principal. Va a buscar instruccions i dades a la memòria principal i hi escriu noves dades.

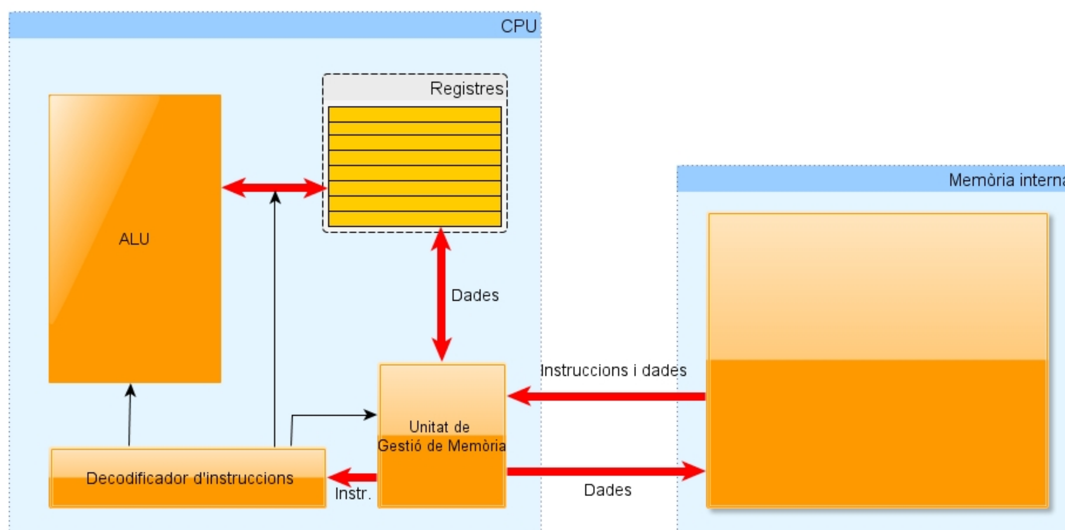


Figura 3.1: Detall de l'activitat de la CPU.

Vegeu un esquema abstracte de CPU i memòria principal a la Figura 3.1. Les instruccions del programa que s'està executant es llegeixen des de la memòria principal i passen al decodificador d'instruccions. Aquest ordena les accions oportunes als demés components de la CPU. Per exemple, per sumar dos nombres que resideixen a la memòria principal:

1. La unitat de gestió de memòria va a buscar el primer sumand a la seva posició de memòria i el diposita en un dels registres.
2. La unitat de gestió de memòria va a buscar el segon sumand i el diposita en un altre registre.

3. Els registres passen a la ALU, on es calcula la suma, que es diposita en un altre registre.
4. La unitat de gestió de memòria guarda el resultat a la memòria principal.
5. Acabada l'execució de la instrucció, la unitat de gestió de memòria va buscar la següent instrucció del programa i la diposita en el decodificador d'instruccions.

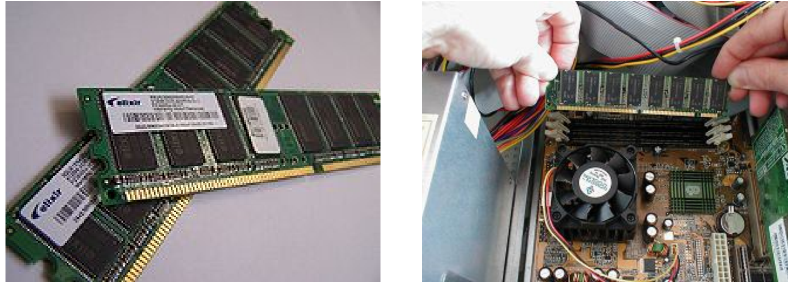


Figura 3.2: Memòria RAM. És molt fàcil canviar-la en un ordinador de sobretaula o en un portàtil. La CPU es troba sota el ventilador que es veu a la imatge.

3.2. CPU

Un dels paràmetres principals que determinen el rendiment d'un ordinador és la velocitat de la CPU (*clock rate*): Les CPU actuals tenen velocitats per sobre del GigaHertz (GHz). Un Hertz és un *cicle per segon*. Aquí, un cicle és una pulsació donada per un aparell anomenat *rellotge*. Els canvis en l'estat del ordinador només es produeixen quan el rellotge genera una pulsació. Un GigaHertz és igual a 10^9 Hz; per tant, dir que una CPU va a 1 GHz vol dir que es produeix una pulsació del rellotge cada nanosegon (10^{-9} segons).

Com que el senyal elèctric no pot viatjar a més velocitat que la llum, veiem que durant un cicle de rellotge, el senyal no ha pogut recórrer més de 30 centímetres. En un processador de 3 GHz, no més de 10 cm. Enviar una ordre a la memòria, trobar la dada correcte i posar-la als registres de la CPU tarda uns quants cicles de rellotge. En conclusió, incrementant només la velocitat del rellotge no podem fer anar gaire més ràpid un ordinador. Calen altres estratègies.

Una bona estratègia es "acostar" la memòria a la CPU usant *cache memory* (*memòria cau*)[2]. Aquesta és una petita memòria, molt ràpida (i cara), situada físicament dins el mateix xip de la CPU. Quan es vol llegir una posició de memòria, es llegeix tot un bloc consecutiu de posicions memòria i s'emmagatzema en la memòria cache. Quan, posteriorment, la CPU vol accedir a una altra dada, si aquesta ja és a la cache, ho pot fer en un o dos cicles de rellotge; si no hi és, anirà a la memòria principal a buscar un altre bloc i carregar-lo en la cache. Això últim implica més temps que anar a buscar només una dada, però ve compensat amb avantatge per les moltes vegades que només cal accedir a la memòria cache.

De fet, el concepte funciona tan bé que actualment s'usen dos o tres *nivells de cache*. El nivell L1 és petit i molt ràpid, el L2 és més gran i no tan ràpid, etc. Si la dada que es busca no està en el nivell L1, és busca en el L2, etc, i si no

[2]"cau" és el nom recomanat en català, però gairebé tothom l'anomena "*catxé*".

està en cap cache aleshores s'accedeix a la memòria principal. Típiques mides actuals són 256 KB per la cache L1, 1 MB per la L2 i 8 MB per la L3.

Cada nivell de cache està dividit en *línies*. Cada línia conté una còpia d'un conjunt de bytes consecutius de la memòria principal. Quan una dada que es busca es trobar en una cache tenim un *cache hit*; en cas contrari és un *cache miss*.

Els registres constitueixen la memòria d'accés més ràpid. Es poden accedir en un cicle de rellotge, per tant de l'ordre de dècimes de nanosegon, mentre que la cache L1 s'accedeix en aproximadament 1 nanosegon, la cache L2 de l'ordre de 10, i la memòria principal de l'ordre de 100 nanosegons. Una L3 està en un temps intermedi entre la L2 i la memòria principal.

L'arquitectura real i les estratègies per aprofitar la memòria cau són molt variades i en constant evolució. La idea bàsica sí que és un concepte perdurable.

3.3. Memòria externa

La *memòria externa* o *d'emmagatzematge massiu* es fa servir per emmagatzemar dades de forma permanent, és a dir, que no desapareixen quan l'ordinador s'apaga[3].

Diversos dispositius poden ser considerats memòria externa:

- **Discos durs** *(Hard disks)*.

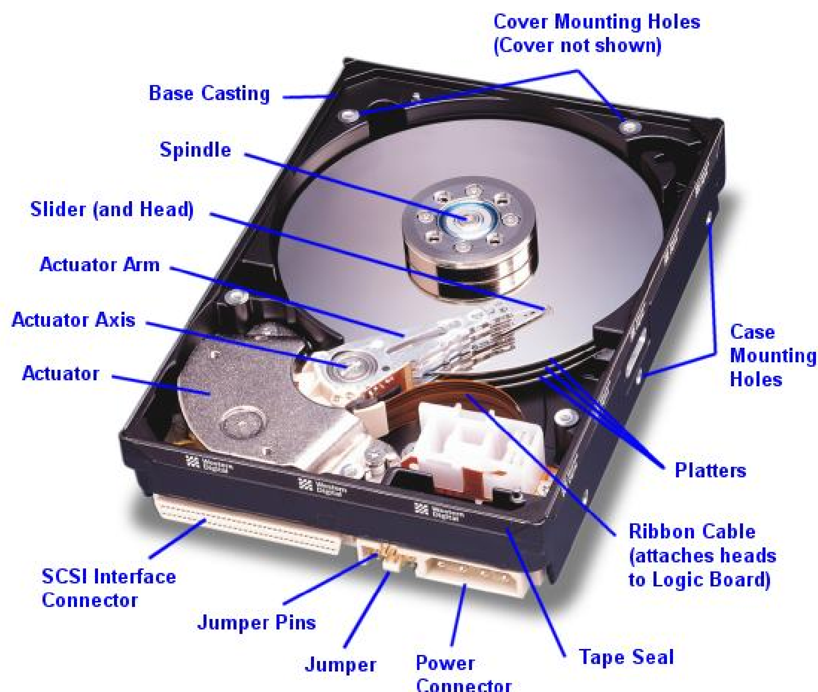


Figura 3.3: Disc dur per dins

Consten de diversos discos apilats (*platters*, entre 3 i 6), formant un cilindre. La informació es guarda de forma magnètica en les dues cares

[3]En anglès, la paraula "memory" gairebé sempre fa referència a la memòria interna (temporal), i es tendeix a usar "storage media" per a l'emmagatzematge permanent, no depenent d'alimentació elèctrica.

dels discos i es llegeix i s'escriu mitjançant capçals (*headers*), un per cada cara de cada disc, mentre tot el cilindre gira al voltant d'un eix. Noteu que aquesta arquitectura està pensada per a poder accedir molt ràpidament a zones del disc molt separades.

Els discos durs estan tancats hermèticament, per tal de poder girar a gran velocitat i que els capçals es mantinguin a distàncies ínfimes dels discos, de l'ordre de nanòmetres. Una velocitat de gir típica és 7200 rpm, el que permet una *velocitat de transferència* mitjana de l'ordre de 1 Gbps. El *temps d'accés* mitjà (temps necessari per trobar un bloc de dades i començar a transmetre-les) és de l'ordre de 10 mil·lisegons.

L'adjectiu "dur" de disc dur (*hard disk*) és en oposició a "disc tou" (*floppy disk*), que fa referència als antics *diskettes*, ja desbancats per les memòries flash. Els floppies tenien una capacitat de 1.44 MB, i des que els ordinadors van incorporar discos durs, només s'utilitzaven per transportar informació.

- **Cinta magnètica** (*Magnetic tape*).

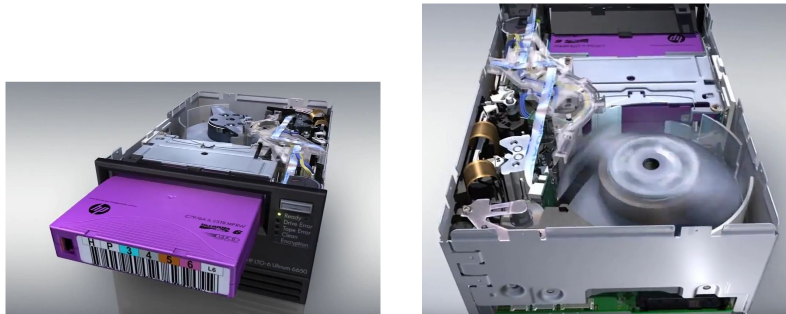


Figura 3.4: Cartutx de cinta magnètica i un lector de cartutxos

Similar a les antigues cintes de cassette per música, però amb bobines molt més llargues i cintes més gruixudes. És un dels mitjans d'emmagatzematge més antics. Comparada amb els discos durs, té l'inconvenient que només s'hi pot accedir *seqüencialment* (per anar d'una punta a l'altra de la cinta cal rebobinar-la tota), i per tant el temps d'accés és molt llarg. Com a avantatge, el cost per bit emmagatzemat és molt més baix. Malgrat la seva llarga història, la cinta magnètica continua essent un dels mitjans més fiables per fer còpies de seguretat.

- **CD (*Compact Disk*), DVD (*Digital Versatile Disk*) i Blu-Ray**.

Figura 3.5: Compact Disk i lector de Compact Disks

Tots ells estan basats en el mateix principi: el disc és una superfície reflectant, sobre la qual es graven les dades fent marques físiques que poden ser llegides amb l'ajuda de llum, concretament un raig làser molt focalitzat mentre el disc gira. Les marques fan que la llum làser que arriba es reflecteixi en dues possibles direccions, cosa que permet emmagatzemar informació en sistema binari. Es diu que són medis *òptics* d'emmagatzematge (*optical storage*).

La principal diferència entre CDs i DVDs és la seva capacitat. En un CD es poden emmagatzemar uns 800 MB, mentre que un DVD té una capacitat de fins a 8.7 GB. Físicament, consten d'una única pista contínua (com els discos de vinil de música), dibuixant una espiral que es grava de dins cap a fora. Els Blu-Ray usen lectors amb làsers més precisos, i la capacitat arriba fins a 50 GB.

Com les cintes magnètiques, són medis que es guarden separats de l'ordinador i són fàcilment muntables i desmuntables en una unitat lectora. A diferència de les cintes, permeten accedir a gran velocitat directament a la zona que interessa per llegir dades.

La velocitat de transferència de dades és de 1200 Kbps si se'ls fa girar igual que un CD d'àudio, a 200 rpm (velocitat 1x). Hi ha lectors que multipliquen aquesta velocitat 52 vegades (52x). El temps d'accés mitjà va des de 400 ms per 1x fins a 80 ms per 52x.

- **Memòries flash**.

Figura 3.6: Les targetes SD i els pen-drive són memòries flash

La memòria flash determina el valor de cada bit amb càrregues elèctriques, usant un fenomen físic anomenat *Fowler-Nordheim tunneling*, però no necessita alimentació elèctrica per mantenir-se, a diferència de la memòria RAM.

És el tipus de memòria que porten els *pen-drives*, fàcilment transportables i connectables als ordinadors mitjançant el port USB (Universal Serial Bus), les targetes SD (Secure Digital Cards)[4], i els smartphones en general.

Les memòries flash comencen també a fer la feina tradicional dels discos durs, donat que no tenen les limitacions mecàniques d'aquests, i són més ràpides. Aquests aparells s'anomenen *solid-state drives* (SSD), perquè

[4]Les sigles SDHC i SDXC és refereixen a targetes SD amb "high-capacity" i "extended capacity".

no tenen parts mòbils, i el seu atractiu és la velocitat, el silenci, el poc consum energètic i la fiabilitat. La velocitat de transferència és de l'ordre de 4 Gbps, amb un temps d'accés de 100 microsegons. S'utilitzen ja en ordinadors portàtils lleugers. L'inconvenient principal, actualment, és que el cost per bit és encara superior al del disc dur, i les capacitats són menors.

Una limitació teòrica de la memòria flash és que admet només una quantitat finita de cicles d'esborrat/escriptura. A la pràctica, aquest nombre és de l'ordre de 100 000 cicles, abans que el desgast aparegui i es perdi fiabilitat.

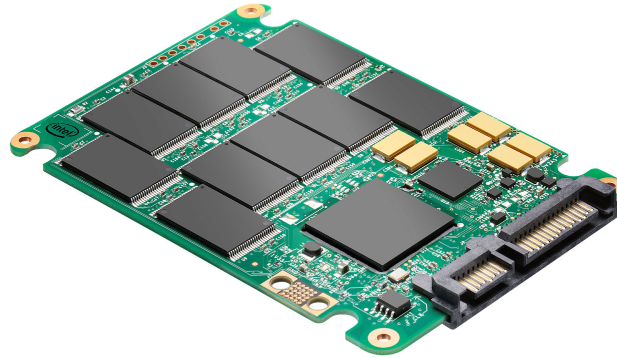


Figura 3.7: Un solid-state drive per dins

3.4. El disc dur

3.4.1 Sistemes de fitxers

Abans de poder-lo utilitzar, un disc dur s'ha de *formatar*. La superfície del disc ve de fàbrica amb unes marques que separen diferents *blocs* o *sectors* de mida 4 KB (fins l'any 2011 l'estàndard era de 512 B). El formatat consisteix en crear un *sistema de fitxers* (*file system*), que d'entrada estarà buit.

El sistema de fitxers és una estructuració (en *fitxers* i *directoris*) de la informació guardada, juntament amb un conjunt de *metadades* (noms de fitxers, dates de creació i modificació, localització del fitxers en els sectors físics, permisos d'accés, etc) sobre aquesta informació.

També s'anomena per extensió *sistema de fitxers* al propi software amb què el sistema operatiu accedeix al disc dur després del formatat.

El típic sistema de fitxers que utilitzen les versions modernes de Windows per formatar discos durs s'anomena `NTFS` (New Technology File System). En canvi, els antics sistemes `FAT16` i `FAT32` són l'estàndard per les targetes SD que s'usen en càmeres fotogràfiques i telèfons, i en els pen-drive, perquè tots els sistemes operatius en ús actualment són capaços de llegir medis formatats amb aquests sistemes de fitxers.

El sistema de fitxers típic de Linux i Android és `ext4`. Linux pot llegir i escriure en un sistema de fitxers `NTFS`; en canvi, Windows no sap llegir els sistemes `ext4`.

El sistema de fitxers utilitzat durant molt temps pel MacOS (i el iOS per smartphones) s'anomena `HFS+` (Hierarchical File System Plus). Ha estat subs-

tituït pel `APFS` (Apple File System) a partir del MacOS High Sierra i el iOS 10.3 (2017). El MacOS pot llegir `NTFS`, però no escriure-hi, i no pot veure sistemes `ext4`[5].

Qualsevol medi que hagi de guardar la informació en fitxers ha de tenir definit un sistema de fitxers. Per tant, tot això s'aplica també a SSDs, CDs, pendrives, etc.

3.4.2 Particions

Abans de formatar-lo, el disc dur es pot *particionar*, és a dir, es pot dividir en diferents particions, cadascuna de les quals por ser formatada amb un sistema de fitxers diferent. Permet pensar en un únic disc físic com si fossin diversos discos individuals. Una partició formatada s'anomena també un *volum*. A la pràctica, partició i volum venen a ser sinònims.

Es poden veure, crear i modificar les particions que tenim en un disc dur amb les eines proporcionades pels mateixos sistemes operatius, o per software de tercers. Per exemple, en Windows 10, podem obrir des del menú inici, successivament, Windows Administrative Tools - Computer Management - Disk Management, i veurem de manera gràfica les particions del disc (o discos) que tinguem instal·lats, la mida de cada partició i l'espai que tenim ocupat amb fitxers, entre altres coses. També podem eliminar particions, crear-ne de noves i formatar-les, o canviar la seva mida. En Linux, es poden utilitzar els programes GParted o KDE Partition Manager; en MacOS el Disk Utility del menu Utilities[6].

3.4.3 Memòria virtual i paginació

Tots els sistemes operatius actuals per ordinadors poden treballar amb *memòria virtual*, que permet usar més capacitat de memòria interna que la que hi ha físicament instal·lada. Quan la memòria interna està plena, el sistema operatiu mou les parts menys usades a la memòria secundària (és a dir, a la pràctica, al disc dur). Quan torna a necessitar alguna dada d'aquella part, la carrega a la memòria interna altre cop. El concepte és similar al que hem vist entre la memòria cache de la CPU i la memòria interna.

L'escriptura i la lectura de la memòria interna al disc dur i viceversa es fa en blocs sencers de la mateixa mida (típicament 4 KB) anomenats *pàgines*. L'intercanvi de pàgines de memòria s'anomena *paginació* (*paging*), i és una operació lenta. Si hi ha massa paginacions, el rendiment de l'ordinador es degrada visiblement.

Per fer la paginació, Windows té un fitxer anomenat `pagefile.sys`, que per omissió es troba en el directori arrel de la partició on hi ha el sistema operatiu instal·lat (per tant, segurament en el directori `C:\`). És un fitxer ocult; cal activar l'opció de veure fitxers ocults si es vol localitzar.

En Linux, la paginació se sol anomenar *swapping*, i es fa a un *swap file* o, més habitualment, a una *swap partition*, és a dir, una petita partició del disc

[5] Totes aquestes limitacions de compatibilitat entre sistemes operatius i sistemes de fitxers es poden solucionar amb software de tercers; no són insalvables.

[6] Modificar particions i formatar són operacions delicades i és convenient fer còpies de seguretat abans de posar-s'hi. Formatar una partició esborra tots els fitxers que conté. De fet no els esborra, sinó que "oblida" on són.

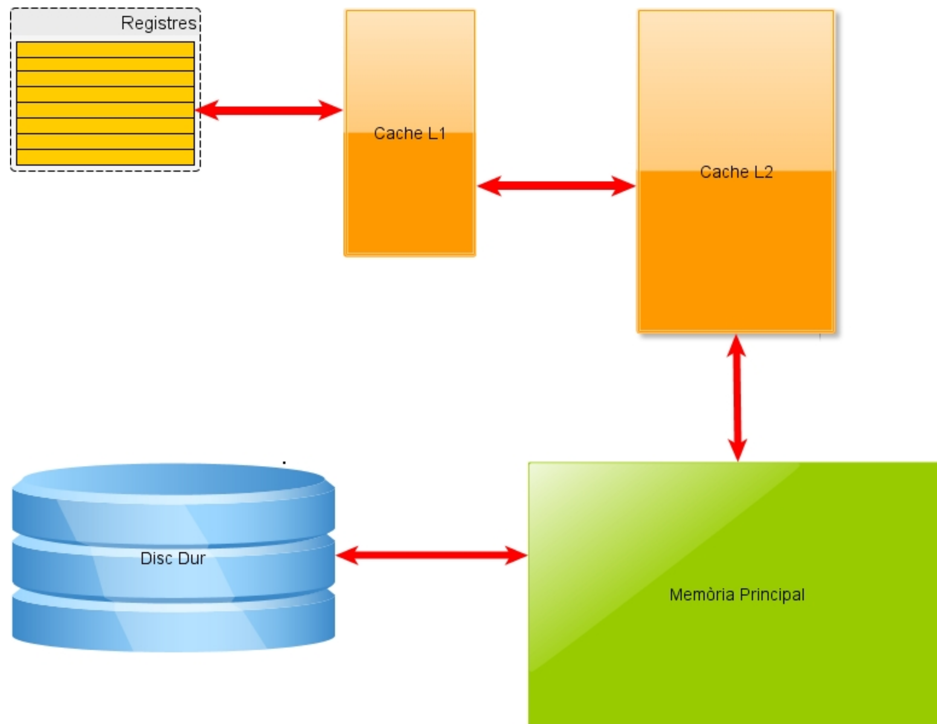


Figura 3.8: Jerarquia de memòries: CPU (Registres, caches L1 i L2), Memòria Principal, i Disc Dur per a memòria virtual

dedicada exclusivament a swapping.

3.5. El procés d'arrencada

El sistema operatiu, del qual parlarem en el proper capítol, és el programa que ha de prendre el control de l'ordinador quan s'engega, i ha d'estar sempre executant-se mentre l'ordinador està en marxa.

Per aconseguir això, en principi només caldria que el hardware de la CPU estigués dissenyat de forma que quan li arribi electricitat, la seva unitat de gestió de memòria anés a buscar en la memòria RAM la posició on hi ha la primera instrucció del sistema operatiu. Però la memòria RAM no té cap programa carregat quan arrenca l'ordinador i, per tant, això no és possible.

En realitat, la CPU va a buscar un programa que es troba, de manera permanent, en la petita memòria ROM. Aquest programa serà l'encarregat d'anar a buscar i carregar en memòria RAM el sistema operatiu. El problema aleshores és: com sap aquest programa on es troba el sistema operatiu?

Actualment s'està fent la transició entre dos sistemes: En el primer, el programa en ROM s'anomena familiarment "la BIOS" (Basic Input-Output System); aquest sistema s'està abandonant progressivament per un altre en què el programa que pren el control inicialment s'anomena "la UEFI" (Unified Extensible Firmware Interface).

Suposarem per simplificar que el sistema operatiu es troba en disc dur, que és el cas més habitual, i que només hi ha un disc dur connectat. De fet, podria haver-hi més d'un disc dur, i el sistema operatiu també pot estar en un CD o un pen-drive, per exemple.

- La BIOS pressuposa que el primer sector del disc dur no pertany a cap

partició, i que conté una taula amb el lloc físic del disc on comença cada partició, i un petit programa anomenat *boot loader*. Aquest primer sector s'anomena Master Boot Record (MBR).

La taula de particions sap quina d'elles conté el sistema operatiu i on. Per tant, el boot loader pot llegir la taula i transferir el control al sistema operatiu. També pot arrencar encara un altre programa intermedi que ens presenti un menú de sistemes operatius, si n'hi ha més d'un en el disc dur, o faci alguna altra tasca. En tot cas, està clar que un cop s'ha pogut començar, es pot fer una arrencada en cadena de programes que acabin amb un sistema operatiu en memòria RAM i funcionant.

És en el moment de formatar el disc dur que es crea el Master Boot Record. Si el formatat es produeix des d'un sistema operatiu concret (freqüentment en el moment d'instal.lar-lo), aquest s'encarrega d'escriure el MBR.

Per disseny, el MBR té 512 bytes, i no hi ha cap gaire informació. Això limita el nombre de particions i la mida total del disc dur. Per això aquest sistema s'ha abandonat.

- La UEFI pot llegir sistemes de fitxers FAT32. Per tant, no cal usar un espai màgic especial al principi del disc dur sinó llegir d'una partició estàndard, que no cal que sigui gaire gran, amb un sistema de fitxers estàndard. Aquesta partició conté una taula de les altres particions del disc i el codi boot loader que es vulgui.

  La UEFI es pot configurar des d'un sistema operatiu en funcionament, perquè les dades estan en una partició ordinària, cosa que no és possible amb la BIOS. (Per configurar la BIOS, cal aturar l'arrencada de l'ordinador amb una determinada combinació de teclat mentre s'està executant, fer els canvis en una interfície no gaire amigable, i després tornar a arrencar.)

  És possible usar un disc amb MBR des de la UEFI, perquè aquesta pot funcionar en un mode "BIOS compatible".

En realitat, el primer programa que pren el control a la ROM quan engeguem l'ordinador és el POST (Power-On Self Test), que verifica el funcionament del hardware de l'ordinador (particularment, la CPU, els discos, i les unitats d'entrada i sortida), que hi ha memòria RAM instal·lada, i que funciona bé. Al acabar la verificació, transfereix el control a la BIOS o la UEFI.

Exercicis

26. Feu una taula dels temps d'accés, ordenats de més petit a més gran, a: registres de la CPU, cache L1, cache L2, memòria principal, disc dur, CD 52x, pen-drive via USB 3.0, SSD, i targeta SD. Poseu-los tots en una unitat comuna (nanosegons, microsegons, ...). Apart de les dades que trobareu en el text, necessiteu saber que el temps d'accés a pen-drive connectat per USB 3.0 és de l'ordre de 1 microsegon, el del CD 52x és de 80 mil·lisegons, i el d'una targeta SD és de 1 mil·lisegon.

Nota: Entre els pen-drive i entre les targetes SD hi ha una enorme varietat, segons fabricant i model. Els temps que hem posat aquí només és una referència molt genèrica.

Nota: La velocitat de gir dels CD no es pot incrementar gaire més (a 52x, quantes revolucions per minut són? Quina és la velocitat lineal de l'aresta exterior?). A aquesta velocitat ja es produeixen molts errors de lectura, soroll i vibracions. Els errors de lectura queden normalment corregits pel software de detecció i correcció d'errors, però quan un CD es nega a funcionar bé, baixar la velocitat de lectura pot ajudar.

27. Feu una taula de les velocitats de transferència, de més gran a més petita, de disc dur, CD 52x, pen-drive via USB 3.0, SSD i targeta SD. Necessiteu saber, apart de les dades que trobareu en el text, que la velocitat de transferència de dades de les targetes SD pot ser d'uns 500 Mbps (depenent molt del model).

Nota: Les velocitats de transferència també depenen de si es tracta d'una operació de lectura o d'escriptura. En general, tots els temps que hem usat en aquest capítol es refereixen a temps de lectura, però els d'escriptura no són molt diferents.

28. Suposem que una CPU ha d'anar a buscar 100 dades a la cache L1, amb un temps d'accés de 1 ns, i que s'aconsegueix un *cache hit rate* del 100%. L'operació es completa per tant en 100 ns.

Suposem ara que s'obté un 99% hit rate, i la dada que falta està a la cache L2, amb un temps d'accés de 10 ns. Això vol dir que la CPU ha necessitat 99 nanosegons per 99 lectures i 10 nanosegons per a una lectura. En total 109 nanosegons. Una reducció d'un 1% en el hit rate ha alentit la CPU en un 9%.

A la realitat, s'ha observat experimentalment que la cache L1 té típicament un hit rate entre 95% i 97%. Quin percentatge d'alentiment representa això, suposant que els L1 *cache miss* han quedat coberts per la cache L2?

Quantes vegades més lent seria aproximadament tot el procés si no hi hagués cache L2 i calgués anar a buscar a la memòria principal entre el 3 i el 5% de les dades que falten?

29. Suposem que tenim una CPU connectada a la memòria interna RAM a través de dos nivells de memòria cache, L1 i L2. Suposem que s'observa experimentalment un *hit rate* del 95% a la cache L1 i del 98% a la cache L2 (la cache L2 només actua quan hi ha hagut un *cache miss* a la L1).

Quantes vegades més ràpida és l'arribada de dades a la CPU en aquesta situació que si no hi hagués cap memòria cache?

30. Alguns ordinadors actuals porten, a part d'un disc dur tradicional, un petit Solid-State Drive (de entre 20 i 64 GB) que s'utilitza com a memòria cache del disc dur. Dins la jerarquia de memòries de la Figura 3.8, va entre la memòria principal (RAM) i la memòria secundària (disc dur). Físicament, el SSD pot estar separat, o bé integrat en el mateix dispositiu físic del disc dur; en aquest últim cas, es parla d'un *hybrid hard disk*.

Aquesta cache s'utilitza per accelerar la càrrega de programes, entre arrencades successives de l'ordinador. Per exemple, des de l'engegada en fred fins a tenir el sistema operatiu a ple funcionament, passant pel login, el temps total es pot reduir a gairebé la meitat.

D'acord amb les dades de velocitat de transferència, si un programa està en el SSD, fins quants cops més ràpid es pot carregar després de tornar a arrencar l'ordinador? Entre el 75% i el 100% d'aquesta quantitat s'observa en efecte a la pràctica.

31. La memòria flash, a diferència dels suports magnètics o òptics, es degrada amb el temps, però a la pràctica no té gaire importància: Suposem que tenim una targeta SDHC de 4GB, i que no fem res més que sobreescriure-la sencera contínuament a la velocitat màxima de 500Mbps.

Quants segons es triga a sobreescriure un cop tota la targeta? Quants dies trigaríem a sobreescriure-la 100 000 vegades? (Les memòries flash més modernes ja parlen de 1 milió de cicles d'esborrat/escriptura abans de degradar-se).

4

Sistemes operatius

4.1. Què fa el sistema operatiu

El **sistema operatiu** (*operating system*, OS) és un conjunt de programes que accedeixen directament al hardware i el gestionen. Ofereixen als demés programes una visió abstracta dels recursos de l'ordinador. Per exemple, els programes no s'han de preocupar de l'estructura física del disc dur, ni de quines són les posicions lliures de la memòria interna per guardar-hi dades.

A la Figura 4.1 podem veure les tres capes, de baix a dalt: Hardware, Sistema Operatiu, Programes.

Figura 4.1: El sistema operatiu com a capa intermèdia entre el hardware i la resta de programes

El sistema operatiu posa els recursos de hardware de l'ordinador a disposició dels programes. Com ja hem vist al Capítol 3, aquests recursos els podem dividir en:

- CPU: El sistema operatiu és el responsable de decidir en cada moment quin dels programes que estan en aquell moment "en execució" (*running*) ha de ser "atès" per la CPU. Tot i que amb els processadors multi-nucli (*multi-core processors*) diversos programes sí poden ser atesos al mateix temps, amb la qual cosa hi ha un cert nivell intrínsec de "paral·lelisme", cal pensar que la CPU en general ha de repartir "llesques de temps"

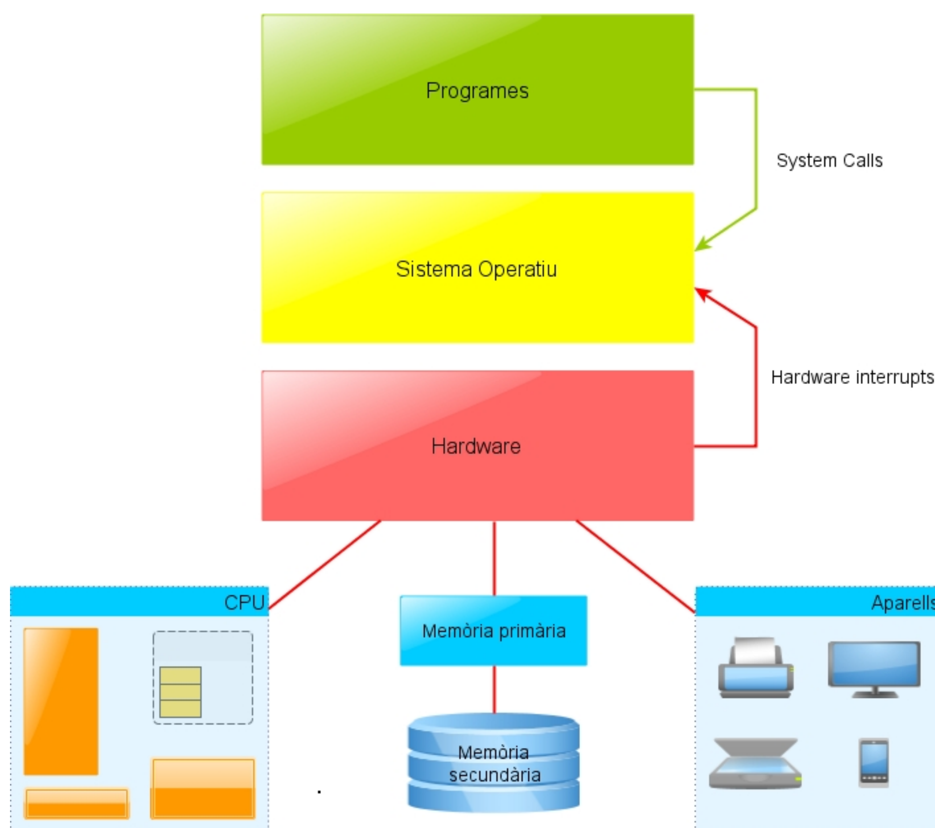


Figura 4.2: System calls, hardware interrupts i detall del hardware

(*time slices*) entre tots els programes que resideixen en memòria. La sensació de simultaneïtat d'execució ve del fet que les llesques de temps són molt fines, de l'ordre d'uns pocs mil·lisegons.

- Memòria: Molts programes en execució poden voler accedir a la memòria principal al mateix temps, i pot ser que en conjunt en demanin més de la que hi ha disponible. El OS és responsable de donar-los la memòria i de decidir què fer quan no hi ha prou memòria física disponible. També ha d'ocupar-se d'accedir al disc dur i altres dispositius de memòria secundària (i per tant gestionar els sistemes de fitxers) a petició dels programes.
- Dispositius d'entrada i sortida: El sistema operatiu s'ocupa de la relació dels programes amb tots els dispositius connectats a l'ordinador, per tal que no sigui cada programa qui hagi de entendre's-hi i saber com funciona cada aparell, a més d'impedir que puguin col·lisionar usant el mateix recurs. La comunicació amb un aparell concret (que pot tenir unes característiques que el sistema operatiu en principi no coneix), es fa a través de programes que s'acoblen al OS, anomenats *controladors* (*drivers*), que acostumen a venir de fàbrica amb l'aparell i s'han d'instal·lar expressament. Un exemple típic és el de la impressora.

Les peticions dels programes al sistema operatiu s'anomenen *system calls*. El hardware notifica esdeveniments al sistema operatiu mitjançant *hardware interrupts*. Un cop iniciat, el sistema operatiu pràcticament només fa que respondre a system calls i hardware interrupts. Vegeu l'esquema a la Figura 4.2.

Els sistemes operatius més populars actualment per ordinadors personals són

Windows, Linux i macOS[1]. El Linux té moltes variants, anomenades *distribucions* (RedHat, Suse, Fedora, Ubuntu, Debian i Mint són algunes de les més consolidades).

Els grans ordinadors (*mainframes*) tenen altres sistemes: z/OS, diversos Unix, Linux-on-Z.

Els aparells petits (smartphones, tablets) també segueixen el concepte d'arquitectura de von Neumann i han de considerar-se ordinadors en tots els sentits. En destaquen els sistemes iOS per a iPhone, de Apple, germà petit del macOS; Android, impulsat per Google i usat per marques com HTC i Samsung, i que és semblant a Linux; i Windows Mobile[2], usat en telèfons Lumia, fabricats per Microsoft[3]. Per a smartTV hi ha diversos OS destacables: Android TV (Sony, Sharp, Philips), Firefox OS (Panasonic), Smart Hub (Samsung), WebOS (LG).

Ordinadors			Smartphones			Tablets		
	2019	2020		2019	2020		2019	2020
Windows	79.5	76.5	Android	75.2	70.7	iOS	71.0	60.3
macOS	14.1	19.0	iOS	22.7	28.8	Android	28.8	39.4
Linux	1.6	1.6	KaiOS	0.7	0.1	Windows	0.2	0.1
Chrome	1.2	1.1	Windows	0.3	0.1	Linux	0.1	0.1

Figura 4.3: "Market share" de sistemes operatius l'abril de 2019 i de 2020, segons dades de `http://gs.statcounter.com/os-market-share`.

4.2. Interfície amb l'usuari

Hom podria dir que el OS també està pendent de l'usuari. Per exemple, parlem de "dir-li al sistema operatiu que arrenqui un programa". En realitat, quan l'usuari fa alguna acció amb un dispositiu d'entrada (ratolí, teclat, etc), està generant una interrupció de hardware.

Per facilitar les accions de l'usuari amb els dispositius d'entrada, el OS utilitza els dispositius de sortida, principalment la pantalla, per donar un feedback sobre quina és la situació de l'ordinador, què està fent, què pot fer l'usuari en aquell moment, etc.

4.2.1 L'escriptori

El que acostumem a veure a la pantalla del nostre ordinador personal s'anomena *escriptori* (*desktop*). En un veritable escriptori, ha d'haver-hi:

- un "fons d'escriptori" (*wallpaper*),
- icones (*icons*),
- una barra de tasques (*taskbar*) i potser altres barres d'eines (*toolbars*),

[1]Fins el juny del 2016 s'anomenava oficialment Mac OS X.

[2]Anomenat Windows Phone fins el 2015. Els números de versió són els mateixos que el de Windows per PC. El 2015 es va passar del Windows Phone 8.1 al Windows 10 Mobile.

[3]L'abril de 2014, Microsoft va comprar la divisió de telèfons mòbils de Nokia i els ha renomenat Lumia.

- un menú d'escriptori (*desktop menu*), anomenat també menú inici (*start menu*), i
- una fletxa o símbol apuntador (*pointer*, *cursor*), que respon als moviments d'un aparell senyalitzador (*pointing device*), com ara un ratolí o un touchpad.

A més, a l'escriptori es poden obrir "finestres", que contenen alguna "informació". Típicament, una finestra és una interfície a un programa que està funcionant, però cal tenir en compte que un programa funcionant no té perquè mostrar cap finestra, o en pot mostrar múltiples, completament separades.

Un escriptori és un exemple de *Graphical User Interface* (GUI), però no tota GUI és un escriptori. Per exemple, en un smartphone, típicament tenim un wallpaper, i icones, però no un dispositiu senyalitzador o finestres que s'obren i es tanquen. Anàlogament, es diu que un programa posseeix o presenta una GUI si es relaciona amb l'usuari de manera gràfica, responent a esdeveniments que crea l'usuari amb els dispositius d'entrada i mostrant la informació mitjançant elements gràfics (botons, menús, checkboxes, sliders, selectors de fitxers, etc., anomenats en general *controls*).

L'escriptori l'hem de pensar com una capa externa, la més externa del sistema operatiu, posada a sobre del nucli del sistema. No és imprescindible l'escriptori per interactuar amb el sistema. Tot i que ens facilita moltes coses, perquè fa la interacció més visual i més fàcil, no necessàriament és més eficient o més ràpid; i no tot el que podem voler demanar al sistema operatiu es pot fer a través de la interacció amb l'escriptori. De vegades cal baixar a un nivell més proper al nucli del OS.

De fet, l'escriptori i el nucli del sistema operatiu poden ser molt independents. Per exemple, en Linux hi ha diversos escriptoris diferents per escollir. Alguns dels més populars són KDE, GNOME, Cinnamon, Unity, Xfce, però hi ha moltes altres possibilitats. Es pot tenir més d'un escriptori instal·lat i arrencar amb el que es vulgui cada cop. La distribució `openuab` de les aules d'informàtica de la Facultat porta el KDE.

"L'escriptori" és una metàfora: Fa pensar la pantalla com si fos la taula de treball de l'usuari, sobre la qual es poden posar documents, carpetes i eines, que a més es poden moure usant el mecanisme d'arrossegar i deixar (*drag and drop*). Un document obert en una finestra representa un paper sobre l'escriptori, en una altra finestra pot haver-hi una calculadora, etc.[3]

Actualment existeixen també alguns "web desktop": A través d'un navegador d'internet ordinari, s'accedeix a un entorn d'escriptori, que permet executar i treballar amb programes en un ordinador remot com si fossin locals. Un exemple pioner és eyeOS[4]. El Chrome OS de Google es pot considerar també un web desktop.

Un altre concepte semblant és el de "web-based operating system", un sistema operatiu pensat per ser gestionat només remotament. Per exemple, els discs durs de xarxa (NAS, *Network Attached Storage*), són en realitat petits ordinadors, sense dispositius d'entrada i sortida, que es gestionen remotament

[3]La metàfora no se segueix estrictament, és clar: La paperera no sol estar sobre la taula, per exemple.

[4]Creat per una empresa d'Olesa de Montserrat, comprada per Telefónica l'abril de 2014, i liquidada el juny del 2017.

mitjançant un navegador web. Els *routers* que connecten una xarxa local a internet

també es poden pensar així.

4.2.2 La consola

Podem interactuar més directament amb el OS a través de la *consola*.

Originalment el mot consola es refereix a un cert tipus de moble. En informàtica, també en origen, vol dir un sistema físic consistent en un teclat i una pantalla, des d'on es controla un ordinador i que només toca l'operador d'aquell ordinador (no els usuaris en sí). Per extensió, hom parla de *mixing console* (taula de mescles, en música), *video game console* (comandaments i pantalla per videojocs), etc. La idea general és la de "centre de control".

En Linux, macOS i Windows tenim programes que emulen una consola, representant-la com una finestra dins l'escriptori: En diem "console window", o "terminal window" o de maneres similars.

Per exemple, Windows ve amb una consola que tècnicament s'anomena `Win32 Console`, i el seu executable està en el fitxer `conhost.exe`. Aquesta finestra pot contenir programes que estan dissenyats per funcionar sense interfície gràfica (tot i que sí poden mostrar text amb colors). La seva funció més habitual actualment es contenir el programa `cmd.exe`, que normalment té una drecera en el *start menu*, sota `Accessories` anomenada "Command Prompt" (o "Símbolo de sistema", o "Indicador d'ordres", segons idioma).

Quan arrenquem el Command Prompt, sigui mitjançant la drecera o escrivint cmd.exe a `Run...` en el menú d'escriptori, s'obre la finestra Win32 Console i veiem que encapsula un espai que no és una interfície gràfica (GUI), sinó de text (anomenada *Text User Interface*, TUI; o més habitualment *Command Line Interface*, CLI). Quan no existien escriptoris ni altres GUI, aquesta era la única interfície al sistema operatiu i es mostrava directament a pantalla completa.

El `cmd.exe` és un "interpret d'ordres" o "interpret d'instruccions" (*Command-line interpreter*, o *shell*)[5]. Presenta a la pantalla un *prompt*, que indica que està disponible per rebre ordres. El prompt ens indica addicionalment on ens trobem dins de l'arbre de directoris (*directory tree*). Sempre hi ha un directori actual (*current directory*) i, si no s'especifica el contrari, moltes instruccions es refereixen només a aquest directori.

En Linux, existeixen una varietat de programes que emulen la consola: `konsole`, `gnome-terminal`, `xterm`, ... El shell del Linux més típic (n'hi ha també altres) s'anomena `bash` i és molt més complet i complex que el `cmd.exe` de Windows. El macOS usa també `bash`.

Més en general, s'usen CLI quan un programa admet un ampli vocabulari d'instruccions, possiblement amb moltes opcions, de forma que és molt més ràpid especificar-les amb una interfície de text que amb una GUI pura. Els shells proporcionats pels sistemes operatius són un exemple d'aquesta situació, però no és la única. Per exemple, el llenguatge R, especialitzat en manipulació de dades i càlculs estadístics, permet treballar-hi interactivament, introduint

[5]Procureu no dir "intèrpret de comandes", que no té adequat en català.

les ordres d'una en una. Per a això, és lògic que existeixin "consoles" de R. El mateix passa amb el llenguatge Python, vegeu les Figures 4.4 i 4.5.

4.3. Instruccions bàsiques del shell

4.3.1 Windows

Algunes instruccions bàsiques per al shell del Windows són:

- `cd (chdir)` per saber directori actual i canviar de directori.
- `dir` per veure el contingut del directori actual.
- `tree` per veure un arbre de directoris sota el directori actual.
- `del (erase)` per esborrar fitxers.
- `copy, xcopy, move` per copiar i moure fitxers entre directoris diferents.
- `ren (rename)` per canviar el nom d'un fitxer.
- `rd (rmdir)` per esborrar un directori buit.
- `help` ofereix la llista d'instruccions que es poden usar. Existeix un fitxer anomenat `WinCmdRef.chm`, que es pot trobar a internet, i que és la referència definitiva de totes les instruccions del shell de Windows.

En Windows també hi ha, apart del *current directory*, el concepte de *current drive*, la "unitat" (partició) que estem mirant. Les particions de disc dur i altres dispositius tenen assignada una *lletra d'unitat* (*drive letter)* i per a cada unitat hi ha un directori actual. Per canviar d'unitat només cal escriure la lletra seguida dos punts, per exemple, `D:`.

La majoria d'instruccions tenen variants, a les que s'hi accedeix amb *switches* (opcions). Per exemple,

- `dir /p` fa una pausa després d'omplir la pantalla de la consola,
- `dir /s` llista tots els fitxers en el directori actual i tots els seus subdirectoris,
- `dir /o` permet que la llista surti ordenada amb diferents criteris (nom, extensió, data, mida, . . .),

i es poden combinar. Per exemple,

```
dir /p /s /oe
```

fa les accions anteriors combinades, ordenant els fitxers per extensió. El switch `/?` ens mostra tots els possibles switches, amb una breu explicació de cadascun.

Les instruccions que admeten noms de fitxers com a paràmetres, normalment permeten especificar un conjunt de fitxers amb els *wildcards* (comodins). Per exemple, `fig*` fa referència a tots els fitxers amb nom començant per `fig`, mentre que `*.txt` representa a tots els fitxers que acaben amb `.txt`. D'aquesta manera, la instrucció `dir /s *.txt` ens llista tots els fitxers d'aquesta mena que hi ha en el directori i subdirectoris, i la instrucció `del *` esborra tots els fitxers del directori (irrecuperablement!). Un altre wildcard és el signe d'interrogació, com ara en `dir figura?.jpg`, que substitueix només un caràcter o cap.

Els *operadors de redirecció* permeten redirigir la sortida o els arguments d'entrada d'una instrucció cap a altres entitats que no són la pantalla i el teclat. Per exemple,

- `systeminfo > MySystem.txt`
  La sortida de `systeminfo` és una mica llarga i és més còmode llegir-la en un fitxer. Amb l'operador `>` l'enviem al fitxer donat (i si el fitxer existia, queda sobreescrit per el nou).
- `tree | more`
  La sortida de `tree` va a parar a l'entrada del programa `more`, que mostra la informació de manera pausada, controlable amb la tecla de retorn i la barra d'espai.
- `type file1.txt | sort > file2.txt`
  La sortida de `type`, que no fa més que mostrar per pantalla el contingut d'un fitxer, es redirecciona a l'entrada de `sort`, que ordena les línies del fitxer alfabèticament, i la sortida de `sort` es redirecciona a un nou fitxer. Usant `>>` en lloc de `>`, el resultat s'afegiria al final de `file2.txt`.
- `find "Memory" < MySystem.txt`
  Ensenya totes les línies del fitxer que continguin la cadena `Memory`.

Si en lloc d'una instrucció del shell, escrivim el nom d'un programa (o sigui, el nom del fitxer que conté l'executable), el programa s'executa. El Windows buscarà el programa en el directori en curs i, si no el troba, el buscarà en els directoris llistats en el *path del sistema*. Per exemple, no hi ha problema en escriure `notepad` o `explorer`[6] a qualsevol directori, perquè aquests programes estan al path. Si l'extensió d'un fitxer està associada a algun programa, escrivint el nom del fitxer també arrencarà el programa corresponent redirigint-li el fitxer especificat.

Les ordres que entén l'intèrpret d'instruccions de Windows es completen amb algunes estructures algorísmiques bàsiques que permeten fer *scripts* (vegeu el fitxer `WinCmdRef.chm` mencionat abans per més detalls sobre el llenguatge). Els scripts de Windows s'anomenen tradicionalment *batch files* i s'han de guardar amb extensió `.bat`.

Automatitzar tasques repetitives és més fàcil usant scripts que a través del escriptori. Alguns programes, com ara alguns editors, tenen també un llenguatge de scripting intern que permet automatitzar tasques freqüents.

La Win32 Console es pot configurar anant al menú contextual de la icona a l'esquerra de la barra superior. A l'apartat "propietats" es pot jugar amb la mida de la finestra, el tipus de lletra, els colors, etc. De tota manera, és una consola relativament modesta. Una alternativa és `Console2` [7], una consola que permet tenir obertes diferents sessions en pestanyes, de manera que és còmode per treballar a la vegada en diversos directoris sense haver de saltar constantment de l'un a l'altre. El llenguatge darrera la consola és el mateix; només canvia la interfície gràfica.

4.3.2 Linux (i macOS)

Les últimes versions del sistema operatius dels Apple Macintosh són variants de Linux. La consola de macOS, anomenada Terminal (del fitxer `Terminal.app`),

[6]Els programes es diuen `notepad.exe` i `explorer.exe`, però es pot abreviar així.
[7]http://sourceforge.net/projects/console/files/

proporciona la interfície a l'interpret d'instruccions `bash`, que és el mateix que típicament es troba en els Linux, i per tant tot és igual en els dos sistemes pel que fa a la consola.

Algunes instruccions bàsiques del `bash`, anàlogues a les que hem vist per Windows, són:

- `cd, pwd` per saber el directori actual i canviar de directori.
- `ls` per veure el contingut del directori.
- `ls -R` per veure tots els fitxers en el directori i subdirectoris.
- `rm` per esborrar fitxers.
- `cp, mv` per copiar i moure fitxers entre directoris diferents.
- `mv` per canviar nom de fitxers (el "movem" al mateix directori).
- `rmdir` per esborrar un directori buit.
- `help` ofereix la llista d'instruccions que es poden usar. Cal tenir en compte que només es llisten instruccions que el bash interpreta internament. Però, per exemple, `ls` no és una instrucció interna, sinó un programa apart. Per obtenir ajuda sobre una instrucció concreta es pot fer, per exemple, `ls --help`, o bé `man ls`.

La majoria d'instruccions tenen moltes variants, més encara que les corresponents de Windows, fins a nivells gairebé caòtics. El programa `ls`, per exemple, té actualment més de 50 switches, que poden interactuar entre ells de maneres no gaire fàcils de preveure.

Els wildcards funcionen de manera similar a Windows, i es poden usar en qualsevol instrucció que admeti noms de fitxers com a arguments. De fet, apart de `*` i `?`, que tenen el mateix significat vist, hi ha altres símbols i construccions més complexes.

Els operadors de redireccionament d'entrada i sortida són similars als de `cmd.exe`. Com a exemples addicionals als vistos abans, considereu com es pot afegir tot un fitxer al final d'un altre fitxer: O bé els obrim en un editor i fem un cut-and-paste, o bé simplement fem

```
cat file2.txt >> file1.txt
```

La instrucció `cat` és l'homòloga del `type` de Windows. Si el que es vol és ajuntar els dos fitxers en un tercer, es pot fer així:

```
cat file1.txt file2.txt > file3.txt
```

I, de fet, un podria anar escrivint línies en un fitxer sense necessitat d'un editor, tot fent anar

```
echo "Aquesta és la nova línia del fitxer" >> myfile.txt
```

Això últim sembla una excentricitat, però pot ser una construcció útil dins un script. Les tres instruccions anteriors funcionen també en Windows canviant `cat` per `type`. Un anàleg del `find` de Windows és el programa `grep`.

Els scripts en Linux no han de portar necessàriament cap característica especial en el nom. Se'ls identifica perquè la primera línia conté els caràcters `#!` seguits del path al intèrpret del llenguatge en què està escrit el script. Per exemple, `#! /bin/bash`, o `#!/usr/bin/python`.

En Linux no hi ha el concepte de "lletra d'unitat". Tots els sistemes de fitxers connectats al sistema han de "muntar-se" en un únic arbre de directoris. Per

exemple, un costum dels usuaris de Linux veterans és tenir els fitxers personals en una partició diferent de la resta. Aleshores aquesta partició es munta dins de l'arbre general, com a `/home/usuari` i tot és exactament igual que si no hi haguessin particions. Anàlogament, el sistema de fitxers d'un pen-drive o un CD-ROM ha de muntar-se en un directori, per exemple, típicament, com a `/media/cdrom` o `/mnt/cdrom`. La instrucció `mount` s'utilitza per muntar explícitament sistemes de fitxers, però actualment això sol funcionar de manera automatitzada al arrencar l'ordinador o quan es connecta físicament el dispositiu en qüestió.

És convenient, tant en Windows com en Linux, desmuntar un sistema de fitxers abans de desconnectar de l'ordinador el dispositiu que el conté. Això és degut a què el sistema operatiu no té perquè accedir als dispositius quan un programa li ho demana, sinó que intenta fer-ho quan té moltes operacions pendents per fer amb ell, per raons d'eficiència. A més d'això, cada cop que es fa una operació de lectura, si hi ha memòria interna lliure el OS guarda un bloc gran a la memòria interna, de manera que, si algun programa vol accedir a les mateixes dades, les pot agafar de la memòria interna en comptes del dispositiu. Per aquests motius, en un instant de temps donat no està garantit que el sistema de fitxers es trobi en un estat coherent (un fitxer pot estar a mig escriure, i les dades que descriuen la seva localització al dispositiu poden estar a mig actualitzar). En Linux s'usa la instrucció `umount`, o alguna eina gràfica. En Windows, es parla de fer una "extracció segura" de hardware.

4.4. Programes i processos

En l'argot informàtic es parla no només de programes, sinó també d'aplicacions, processos, serveis, i fils d'execució, sempre en referència a algun objecte que "corre" o "s'executa" a l'ordinador.

S'acostuma a anomenar *aplicació* (*application*, o *"app"* en dispositius petits) a un programa amb interfície gràfica, que proporciona una eina completa per a una tasca que l'usuari hagi de realitzar. Per exemple, una suite ofimàtica, un navegador d'internet, un lector de fitxers pdf, un editor de fitxers, ..., poden anomenar-se tots aplicacions. És un concepte no gaire precís, que s'usa més aviat amb fins comercials. Tècnicament, no hi ha cap diferència entre "aplicació" i "programa".

La paraula *programa* es pot usar en diferents sentits. Un fitxer font en `C` l'anomenem "programa" quan l'estem escrivint. Quan s'està executant, en canvi, acostumem a dir-li "programa" al codi que està en la memòria RAM. També diem "programa" per referir-nos al fitxer que conté el codi executable. I encara, en un sentit més abstracte, es diu que un programa és qualsevol representació d'un algorisme; sota aquest punt de vista, un diagrama de flux o un pseudocodi són també "programes". No sol haver-hi confusió entre els diferents conceptes, sempre i quan siguem precisos quan calgui.

Un mateix codi executable pot tenir diverses instàncies executant-se al mateix temps. El fitxer executable és el mateix, però s'ha carregat diverses vegades en memòria, i les execucions són independents. Això pot passar perquè nosaltres, explícitament, hem arrencat diverses vegades el programa, o ho pot fer el programa pel seu compte si ha estat dissenyat així. Per exemple, podem tenir perfectament diverses instàncies de la consola executant-se (i veient-se

en pantalla simultàniament); per altra banda, el navegador Google Chrome arrenca sense intervenció de l'usuari tres còpies del programa `chrome.exe`, i una més per cada pestanya nova que s'obri i per cada extensió instal·lada, tot i mostrar una sola finestra contenidora). També hi ha programes dissenyats de manera que no es poden executar dues còpies d'ell simultàniament. Tenir diverses finestres independents tampoc implica que hi hagi més d'una còpia d'un programa en memòria: Per exemple, Microsoft Word pot tenir obertes diverses finestres, però només una còpia de `winword.exe` en memòria.

Cada instància d'un executable que està corrent s'anomena un *procés*. El que està en execució són per tant *processos*. Podem considerar "procés" com a sinònim de "programa en funcionament".

Un *servei* és un procés que corre sense interactuar amb l'escriptori i que un cop arrencat segueix executant-se indefinidament, monitoritzant periòdicament esdeveniments que li puguin interessar i responent a ells. Per exemple, el `Print Spooler service` de Windows captura els documents que s'envien a imprimir, els desa temporalment en disc, i envia un senyal a la impressora (a través del seu driver) per tal que els vagi imprimint al seu ritme. D'aquesta manera, els processos poden enviar documents ràpidament a una cua d'impressora i continuar immediatament amb altres tasques.

Que un programa no presenti una interfície i estigui funcionant "per darrera" (en "background"), no vol dir que sigui un servei. La característica que defineix el servei és que "sempre està escoltant". A més, diversos serveis poden estar funcionant dins un mateix executable. En Linux, els serveis solen anomenar-se *dimonis* (*daemons*)[8].

Un *fil d'execució* (*thread of execution*) és la unitat més petita de execució que el sistema operatiu pot gestionar. Un procés pot subdividir-se en diversos fils, i el OS s'ocupa d'assignar-los temps d'execució individualment. Però els diversos fils comparteixen recursos, com ara la memòria, de manera que un fil pot llegir una posició de memòria escrita per un altre fil. Això no és possible entre processos diferents; processos diferents només poden interactuar a través dels mecanismes de comunicació entre processos proveïts pel OS. Una situació típica en què es necessita més d'un fil d'execució és quan un programa presenta una interfície gràfica a la vegada que ha de fer càlculs intensius per darrera. Un botó per aturar els càlculs, per exemple, no podrà funcionar si càlculs i interfície gràfica no s'han separat en dos fils.

Exercicis

32. (a) Si un sistema operatiu dóna llesques de temps de 10 mil·lisegons a cada procés i la CPU executa una mitjana de 2 instruccions elementals per nanosegon, quantes instruccions en mitjana pot executar un procés durant una sola llesca de temps?

(b) En les antigues màquines d'escriure, un bon mecanògraf professional podia teclejar fins a 300 pulsacions per minut. Segons el *Llibre Guinness dels Rècords Mundials*, la millor marca amb un teclat d'ordinador la té Barbara Blackburn, amb una velocitat sostinguda de 750 pulsacions per minut durant 50 minuts.

[8]Al·lusió al *Maxwell's daemon*, ésser imaginari emprat pels físics.

Suposant llesques de temps de 10 mil·lisegons, quantes llesques es poden concedir a altres processos entre dos caràcters consecutius de la senyora Blackburn?

33. El sistema operatiu suspèn immediatament la llesca de temps assignada a un procés quan detecta que l'operació en curs involucra algun perifèric més enllà de la memòria principal. Motiu:

Suposem que el sistema operatiu dóna llesques de temps de 50 mil·lisegons nominals. El temps d'accés a una dada del disc dur és d'uns 10 mil·lisegons de mitjana, segons hem vist al capítol 3. Si la CPU pot executar 2 instruccions elementals per nanosegon, quantes instruccions d'un altre procés es poden executar mentre esperem que el disc dur respongui?

Això ens dóna una idea de la complexitat de l'algorisme que assigna llesques de temps: Ha de mantenir en la memòria principal una *taula de processos*, contenint informació sobre la seva prioritat, l'àrea de memòria que ocupa, i si el procés està *preparat* per continuar o *esperant* que es completi alguna operació (arribada de dades del disc dur, una pulsació del teclat, etc). Quan cal canviar a un altre procés, es completa el cicle actual de rellotge, es guarda l'estat del procés en curs, es decideix a quin altre procés es concedeix la llesca de temps següent, i es carrega a la CPU l'estat del nou procés que s'havia guardat prèviament (següent instrucció a executar, contingut dels registres, etc).

34. Per veure quins processos estan funcionant en Windows, tenim diverses opcions:

(a) Des de la consola, podem usar l'ordre `tasklist`. Afegint el switch `/svc` es veuen els serveis continguts en cada procés. En efecte, diversos serveis poden estar funcionant dins un mateix procés. Això és típic dels serveis aportats pel propi Windows, que s'executen dins de processos instàncies de `svchost.exe` (dels quals segurament n'hi ha més d'un funcionant).
Afegint el switch `/v` obtenim una sortida amb més informació.
Executeu `tasklist /svc`, redirigint la sortida a un fitxer. Mirant el fitxer, identifiqueu alguna aplicació que tingueu oberta i anoteu el seu PID (número d'identificació del procés). Mateu el procés amb la instrucció `taskkill`, mirant com s'utilitza en el fitxer `WinCmdRef.chm`, o buscant-ho a internet.

(b) Podem fer la combinació de tecles `Ctrl-Alt-Del`, per obrir el `Task Manager`, una finestra gràfica que ofereix una pestanya *Applications* (programes que mostren finestres, essencialment) i *Processes*, on es veu informació semblant a la que dóna `tasklist`.

(c) Hi ha un programa encara més complet, anomenat Process Explorer[9], fet per un equip de Microsoft anomenat Windows Sysinternals, i que proporciona molta més informació sobre el que està passant.

35. Per veure quins processos estan funcionant en Linux, podem fer-ho:

(a) Des de la consola, amb l'instrucció `ps`. Veureu el PID de cada procés, com en el cas del `tasklist` de Windows. Per matar un procés, feu `kill` seguit del PID del procés.

[9] `http://goo.gl/g88d`

(b) Amb alguna eina gràfica que porti la distribució concreta de Linux. Per exemple, el KSysGuard, que forma part del software lligat al escriptori KDE, és similar al Task Manager de Windows.

36. Per veure els serveis instal·lats, tenim en Windows el programa `services.msc` (el més fàcil per arrencar-lo és escriure el seu nom al menú inici, `Run...`, però també s'hi pot arribar des del Control Panel, sota Administrative Tools). Es mostra gràficament una llista de tots els serveis amb el seu status (si ha arrencat o no) i la seva manera d'arrencar: Automàtica (quan arrenca el sistema operatiu), Manual (quan se'l necessita per primer cop), o Desactivat (si no volem que arrenqui mai). Molts d'aquests serveis són força inútils i es poden deixar desactivats per estalviar recursos, però requereix paciència esbrinar quins interessen i quins no.

Figures

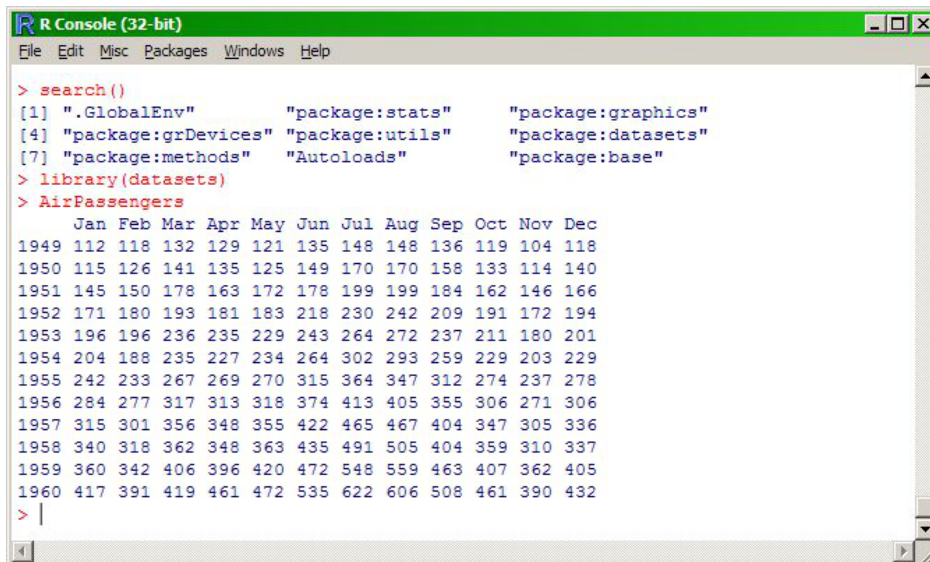


Figura 4.4: Consola per al llenguatge R

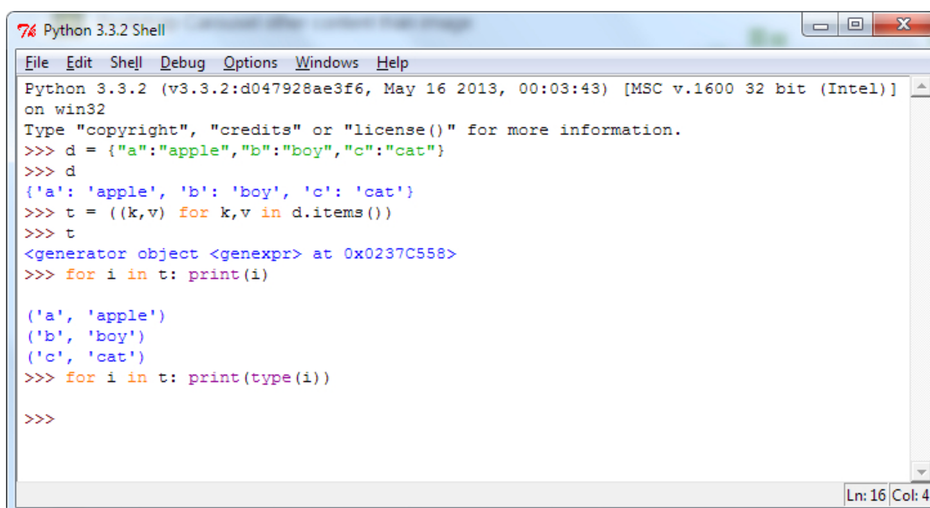


Figura 4.5: Consola per al llenguatge Python

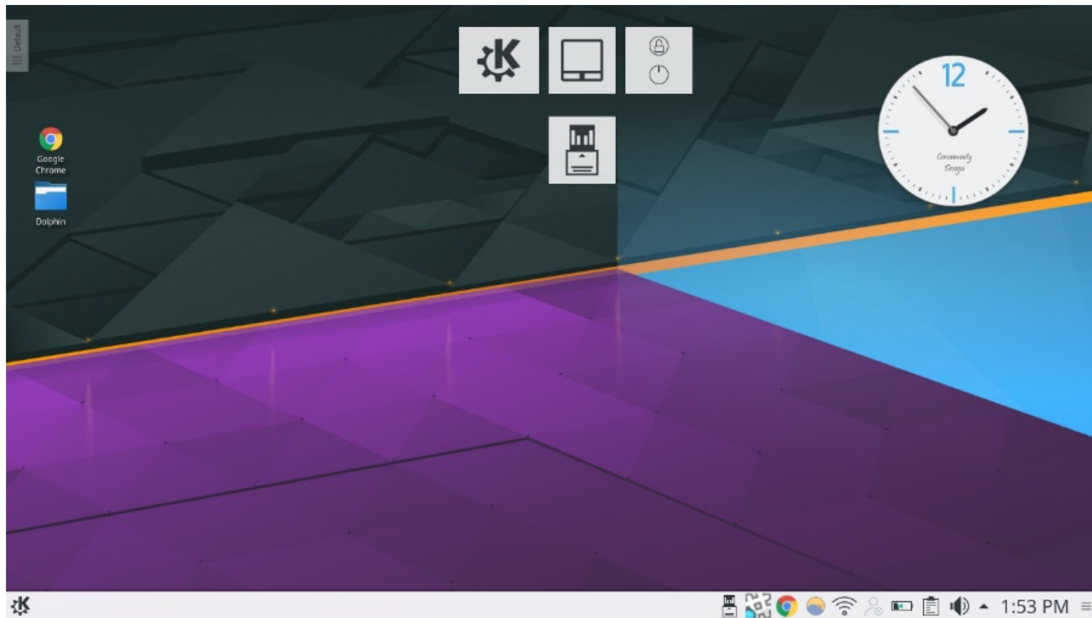


Figura 4.6: L'escriptori KDE

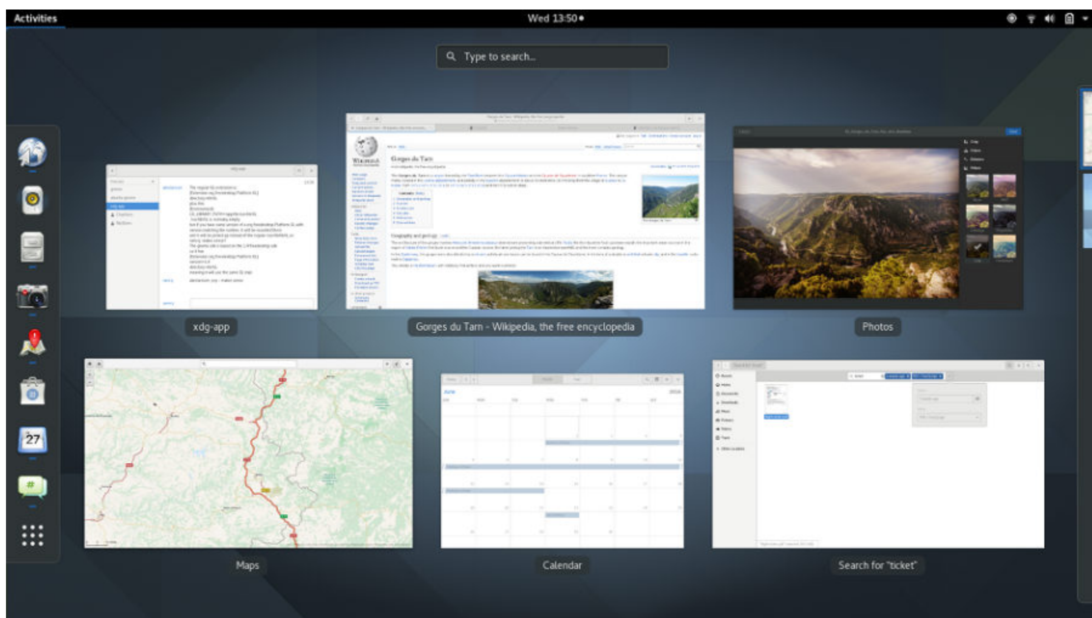


Figura 4.7: L'escriptori GNOME

Figura 4.8: L'escriptori Cinnamon

Figura 4.9: L'escriptori Unity

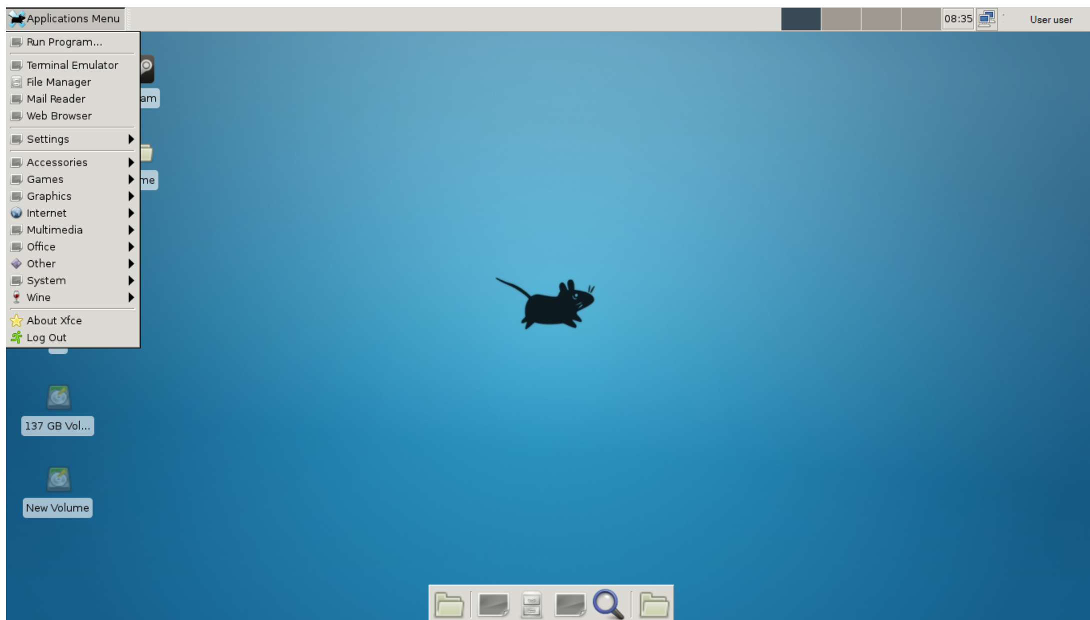


Figura 4.10: L'escriptori Xfce

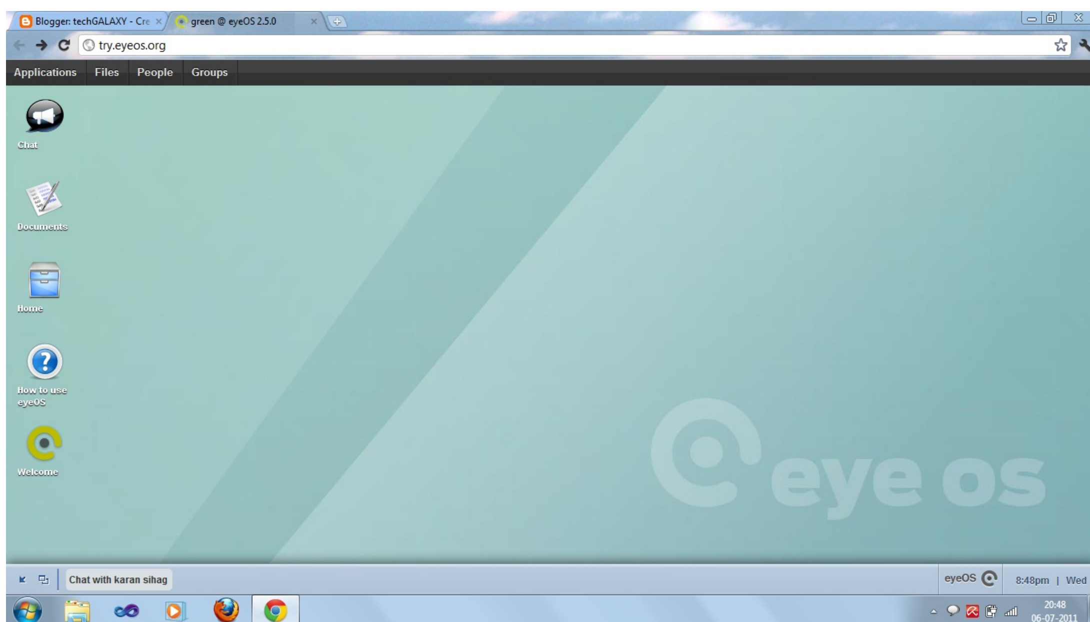


Figura 4.11: El web desktop eyeOS funcionant dins el navegador Chrome

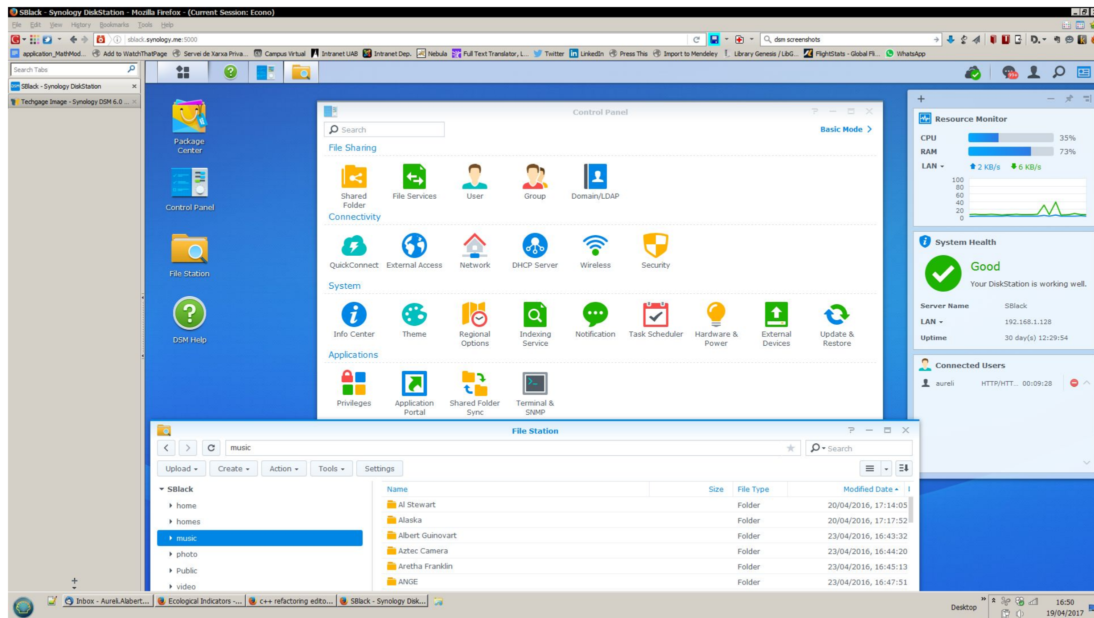


Figura 4.12: El sistema operatiu DSM d'un NAS Synology mostrant un escriptori, dins el navegador Firefox d'un ordinador remot

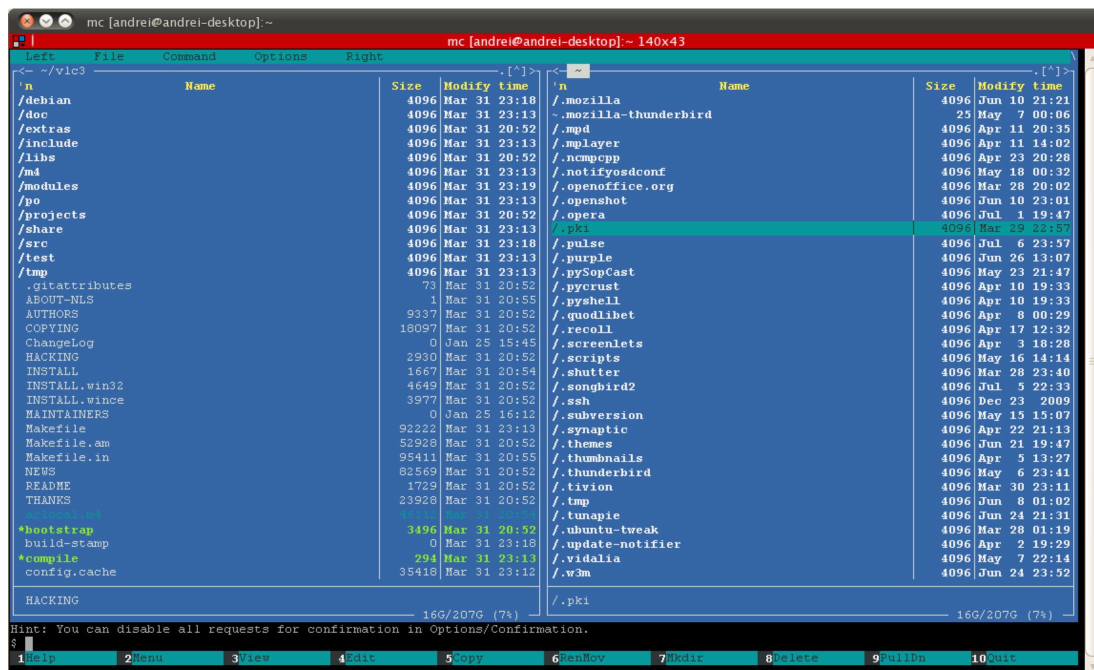


Figura 4.13: El gestor de fitxers Midnight Commander funcionant en una consola de Linux Ubuntu. També té versions per a Windows i macOS.

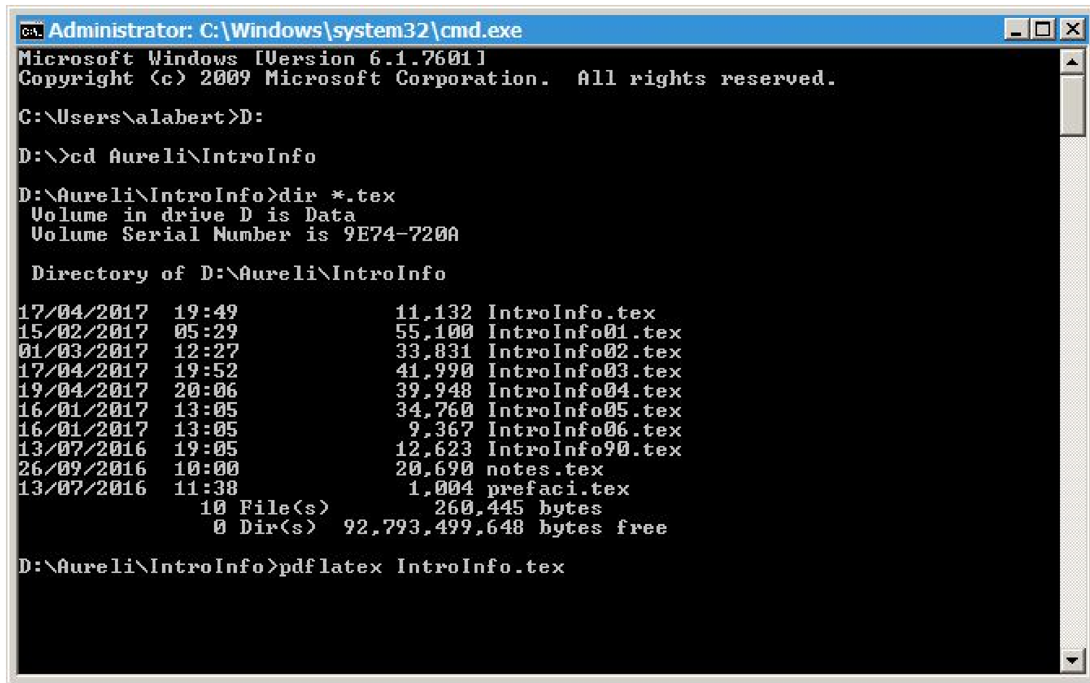


Figura 4.14: L'interpret d'ordres `cmd.exe` dins la Win32 Console de Windows

5

Llenguatges i compilació

5.1. Llenguatges i codi màquina

Un *llenguatge de programació* és un conjunt de símbols, i de regles per combinar aquests símbols, que permet expressar algorismes.

Tot programa escrit en qualsevol llenguatge de programació s'ha de convertir a *codi màquina* abans de poder-se executar. El codi màquina està fet de les instruccions elementals que la CPU pot executar, com ara una operació aritmètica amb dades en els registres, un accés a memòria per llegir o escriure, o una decisió de quina és la següent instrucció a executar segons el resultat d'una operació prèvia. Cada instrucció en codi màquina és simplement una seqüència de bytes que provoca una determinada acció de la CPU gràcies al disseny físic d'aquesta.

El codi màquina està molt lligat al hardware. Cada CPU té en principi el seu propi conjunt d'instruccions; ara bé, pel que fa a ordinadors personals hi ha dos únics estàndards que conviuen actualment: *x86* i *x64*.[1] Els processadors x64 poden executar les instruccions x86; per tant, es poden considerar una "extensió", i tots els ordinadors personals que es venen actualment porten un processador tipus x64, fabricat per Intel o per AMD. Quan el compilador produeix un executable, ho fa per un determinat estàndard.

Dels processadors x86 es diu que tenen una "arquitectura de 32 bits", mentre que els x64 són de "64 bits". Això fa referència a la mida dels registres i la diferència més notable a la pràctica és que 32 bits només poden contenir 2^{32} adreces de memòria diferents. Cada adreça de memòria referencia un byte. Això fan 4 GB de memòria, i per tant un ordinador amb un processador de 32 bits no podrà usar més d'aquesta quantitat. En canvi, amb 64 bits es pot referenciar una quantitat enorme de memòria.

L'executable ha de tenir un cert format, i cada sistema operatiu pot requerir un format diferent. Això fa que el codi màquina generat pel compilador no només depengui del tipus de processador, sinó també del sistema operatiu on s'ha d'executar. Per exemple, Linux i Windows són incompatibles en aquest aspecte. I si el programa en qüestió usa biblioteques de funcions proporcionades directament pel sistema operatiu[2], encara s'hi introdueix més incompatibilitat.

[1] També anomenats x86-32 i x86-64, i d'altres maneres
[2] habitualment anomenades API (Application Programming Interface)

El `gcc`, si no se li diu el contrari, produeix executables per a processadors x64 i per el mateix sistema operatiu en el que s'està executant.

És possible escriure programes en codi màquina mitjançant una representació simbòlica que s'anomena *llenguatge assemblador* (*assembly language*), tot i que actualment només es fa en situacions molt particulars.

En llenguatge assemblador cada instrucció correspon a una instrucció elemental en codi màquina. Podem dir què és una representació llegible del propi codi màquina. No obstant, en assemblador s'usen "símbols" a manera de variables, per no haver d'escriure posicions de memòria concretes de les dades.

Per exemple, el programa següent en `C`, contingut en un fitxer `sumar.c`,

```
int main () {
  int suma;
  int i;
  suma=0;
  for (i=1; i<=5; i++) suma=suma+i;
  return 0;
  }
```

fa el mateix que el programa en assemblador:

```
  .file  "sumar.c"
  .def  ___main;  .scl  2;  .type  32;  .endef
  .text
.globl _main
  .def  _main;  .scl  2;  .type  32;  .endef
_main:
  pushl  %ebp
  movl  %esp, %ebp
  andl  $-16, %esp
  subl  $16, %esp
  call  ___main
  movl  $0, 8(%esp)
  movl  $1, 12(%esp)
  jmp  L2
L3:
  movl  12(%esp), %eax
  addl  %eax, 8(%esp)
  incl  12(%esp)
L2:
  cmpl  $5, 12(%esp)
  jle  L3
  movl  $0, %eax
  leave
  ret
```

En realitat, aquest codi assemblador no ha estat escrit de zero, sinó que l'ha produït el mateix compilador de C, invocat com

```
gcc -Wall -S sumar.c
```

Això crea un fitxer `sumar.s`, que és el que s'ha copiat aquí.

Els llenguatges de programació han estat desenvolupats per tal d'evitar haver d'escriure en assemblador. Permeten expressar algorismes de manera més

propera al llenguate natural i a la notació matemàtica que no pas el llenguatge assemblador, que està completament lligat a l'estructura d'una CPU concreta. Al mateix temps, el codi és "portable": A partir del mateix fitxer font es poden produir executables per a diferents CPU i diferents sistemes operatius.

5.2. Tipus de llenguatges

Els llenguatges de programació es classifiquen seguint diferents criteris, però dins de cada criteri no sempre és fàcil separar les diferents classes.

El criteri de classificació més important és el de la manera i moment en què les instruccions es converteixen a codi màquina. Tenim, en aquest sentit:

- Els *llenguatges compilats*: El programa es converteix en un seguit d'instruccions en codi màquina, que s'emmagatzemen en un *fitxer executable*, preparat per ser carregat en memòria i executat.

  Exemples: C, C++, Fortran, Pascal.

- Els *llenguatges interpretats*: Un programa anomenat *intèrpret* executa (interpreta) el programa línia a línia. No es transforma a codi màquina ni es crea un fitxer permanent executable. El fitxer font s'anomena de vegades *script*, i es parla de *llenguatges de scripting*. Per a la majoria de llenguatges interpretats existeixen CLI[3], que permeten fer *sessions interactives*, introduint i executant les instruccions una a una.

  Un programa interpretat s'executa de manera molt més lenta que un programa compilat; l'avantatge és la portabilitat entre plataformes[4].

  Exemples: Python, R, BASIC, PHP.

- *Bytecode:* El bytecode és una semicompilació del fitxer font, en una seqüència de bytes independent de màquina. El bytecode per tant no es pot executar sobre la CPU, sinó que s'executa sobre una *màquina virtual*, un software que ha d'estar instal·lat apart en l'ordinador que pretén executar el bytecode i que sí que és diferent per a cada plataforma. La màquina virtual pot interpretar el bytecode, o bé compilar-lo *Just-in-Time*, cada cop que cal executar el programa.

  L'avantatge d'aquesta idea és que l'execució és més ràpida que en el cas dels llenguatges interpretats, tot mantenint la seva portabilitat. El desavantatge és que encara és més lent que el codi màquina nadiu, i que la màquina virtual sol ocupar grans quantitats de memòria quan s'executa.

  Exemples: Java, Ruby.

En rigor, no és correcte dir que això és una classificació de llenguatges. No és ben bé el llenguatge en sí que és interpretat o compilat; és més aviat l'existència de compiladors o intèrprets d'un llenguatge que el fan compilable o interpretable. Per exemple, hi ha llenguatges, com ara BASIC, que es poden interpretar o compilar; per altra banda, actualment el R i el Python, llenguatges tradicionalment interpretats, permeten construir bytecode de tot o una part del programa.

[3]Vegeu el capítol anterior.

[4]Una *plataforma* és una combinació de CPU i sistema operatiu.

En la mateixa línia, és possible i raonable escriure scripts en Python o R, per exemple, que cridin a funcions compilades en C (aquelles que continguin càlcul intensiu i per tant necessitin més velocitat d'execució), de manera que el resultat final és un programa mig interpretat, mig compilat.

Altres possibles criteris de classificació són: A quin tipus d'aplicació real està orientat el llenguatge? A quin estil de programació dóna suport? A quina distància està del codi màquina i del llenguatge humà?

Des del punt de vista de les aplicacions, els podem qualificar com a llenguatges de propòsit general (per exemple, el C ho és), o orientat a aplicacions específiques (per exemple, Fortran per càlcul científic, R per a manipulació de dades i càlculs estadístics, PHP i Javascript per a la creació de pàgines web dinàmiques, etc.

Una tercera manera de classificar llenguatges és respecte de l'estil de programació a què donen suport, el que de vegades s'anomena *programming paradigm*. Es parla de programació *imperativa o procedural*, programació *declarativa*, programació *funcional*, i programació *orientada a objectes*, entre altres. No hi ha una classificació clara de paradigmes. Tal com hem parlat fins ara dels programes, com a representació i implementació efectiva d'algorismes, estem parlant només de llenguatges procedurals.

L'anomenada Programació Orientada a Objectes (*Object Oriented Programming*, OOP) és un estil fonamental en la pràctica actual, per a tot tipus d'aplicacions. Els *objectes* són estructures que contenen dades de diversos tipus, juntament amb les funcions que manipulen aquestes dades. Cada objecte pertany a una certa *classe* (l'anàleg a "tipus de variable"). La feina de programació i el codi en sí són molt més clars, menys propensos a errors, i molt més reutilitzables d'un projecte a un altre. Els llenguatges més importants que ofereixen total suport a la OOP són C++, Java i Python.

5.3. Compilació en C

5.3.1 Codi font, codi objecte i codi executable

En aquesta secció i fins al final del capítol els conceptes s'il·lustren amb el cas concret del llenguatge `C`, però són força generals.

En un llenguatge compilat com el C, el text del programa s'anomena *codi font*, i es troba en un fitxer de text que es pot veure amb qualsevol editor de fitxers. Usant el compilador `gcc`,

```
gcc -o hola.exe -Wall hola.c
```

obtenim el codi màquina en un fitxer *executable* `hola.exe`. Amb l'ordre

`./hola.exe` [5]

el sistema operatiu engegarà un procés per a aquest programa.

Però en realitat, el pas de codi font a codi màquina no és directe, sinó que es passa sempre per un estat intermedi anomenat *codi objecte*. El codi màquina

[5] En Windows els programes executables en codi màquina han d'estar en fitxers que acabin en `.exe`. Quan se'ls vol executar des de la consola, no cal escriure el `.exe` (per exemple, és equivalent escriure `explorer.exe` o `explorer`). En canvi, en Linux els executables no tenen perquè tenir un nom especial, però per executar-los cal escriure estrictament el nom del fitxer, acompanyat del directori on es troba, encara que sigui el directori en curs, representat aquí per `./`

el genera a partir del codi objecte un altre programa anomenat *enllaçador* (*linker*). Aquest pas intermedi facilita la *programació modular*[6]. Quan els programes són llargs, es millor trencar-lo en *mòduls*, agrupant unes quantes funcions i guardant cada mòdul en un fitxer.

Els diversos mòduls d'un programa es compilen per separat. Quan el compilador compila un mòdul i es troba una crida a una funció que es troba en un altre mòdul, no té tota la informació sobre l'adreça on estarà situada la funció en el codi màquina final. Per tant, no pot generar el fitxer executable, sinó que ha de generar una mena de "codi intermedi", que és codi màquina amb referències sense resoldre. Aquest és el codi objecte. Un altre motiu pel qual els mòduls d'un programa es compilen per separat és que no s'hagi de recompilar tot el programa si es fa una modificació que afecta només un mòdul.

L'*enllaçador* processa tots els codis objecte alhora, resol les referències i genera el codi màquina per al fitxer executable.

Vegeu a la Figura 5.1 un esquema d'aquest funcionament. Imaginem que el nostre programa `prog.c` usa la funció `printf()`. El compilador ha de comprovar que l'estem usant correctament; per això necessita la seva *declaració* (vegeu el Capítol 1). Aquesta es troba en el fitxer `stdio.h`, que incorporem al programa amb la directiva `#include <stdio.h>`. Un cop satisfet, el compilador produeix el codi objecte `prog.o`. L'enllaçador incorpora al programa la *definició* (el que realment fa) la funció, que es troba en la biblioteca del sistema `libc.a`. Observeu que el llenguatge C en sí no sap què vol dir `printf()`; només li cal saber que s'està utilitzant bé.

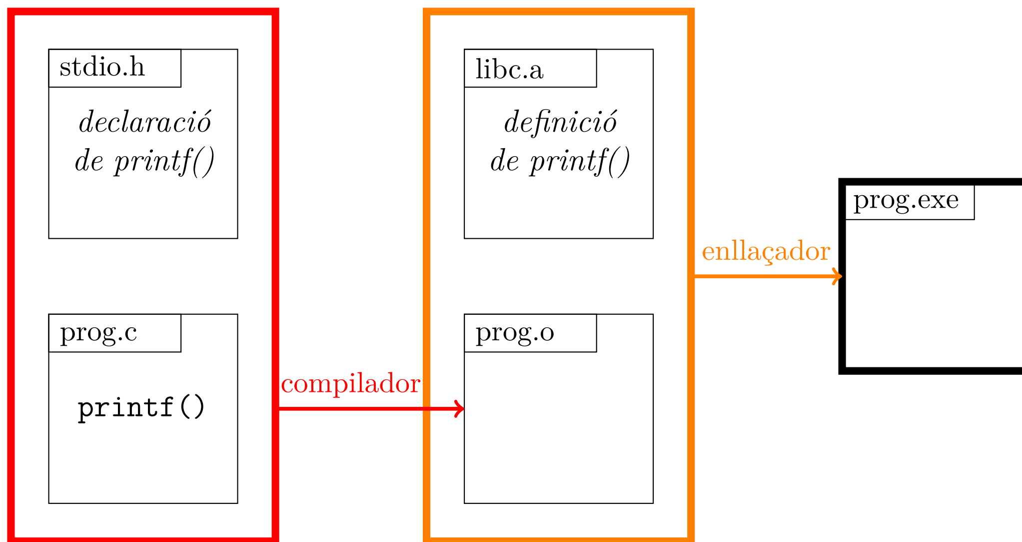


Figura 5.1: Esquema del treball de compilador i enllaçador

5.3.2 Exemple

Volem fer un convertidor de dates de calendari a dies julians i viceversa. El *dia julià* és una unitat usada en astronomia, i es defineix com l'interval mesurat en dies, des del 1 de gener de 4713 a.C., a migdia sobre el meridià de Greenwich.

[6] Aquest no és un concepte que tingui una definició clara i universal; és suficient tenir la idea intuïtiva de què vol dir.

A l'hora de dissenyar el programa, veiem que necessitem dos tipus d'accions:

- Funcions que fan els càlculs, o sigui la conversió d'un a l'altre sistema. Necessitarem una funció `Julian2Cal()`, que passi de dia julià a data de calendari, i una altra funció `Cal2Julian()`, que passi de calendari a dia julià.
- Interacció amb l'usuari (entrada i sortida). Aquesta part la podem posar dins la funció `main()`. De fet, tenint en compte que el programa s'executarà des de la consola, farem dos executables: un que passi de data de calendari a dia julià, i un que faci l'operació contrària.

Dividirem el projecte en quatre mòduls:

- Un mòdul amb les funcions `Julian2Cal()` i `Cal2Julian()`, que anomenarem `calc.c`. Vegeu-ne el codi en el Llistat 5.3.
- Un mòdul amb el programa que converteix de dia julià a data de calendari, que anomenarem `jd2c.c` (Llistat 5.4).
- Un mòdul amb el programa que converteix de data de calendari a dia julià, que anomenarem `c2jd.c` (llistat 5.5).
- Un mòdul amb les declaracions de les funcions `Julian2Cal()` i `Cal2Julian()` (Llistat 5.6), que serà el `calc.h`.

Aquesta divisió permet també la inclusió de les funcions `Julian2Cal()` i `Cal2Julian()` en altres programes diferents dels que farem aquí, sense necessitat de copiar i enganxar codi.

Observacions:

- Fixem-nos que no hi ha funció `main()` en el primer fitxer; només conté les dues funcions dels càlculs. Però sí inclou el fitxer `math.h`, per tal que el compilador trobi la declaració de la funció `floor()` i comprovi que la estem usant correctament.
- En el mateix fitxer, no s'hi inclou el fitxer de capçalera `stdio.h`, perquè no hi ha cap funció d'entrada o sortida.
- Anàlogament, el compilador necessita les declaracions de la funció `Julian2Cal()` i la funció `Cal2Julian()` quan compila els fitxers `jd2c.c` i `c2jd.c`, respectivament. En lloc de posar aquestes declaracions en els mateixos fitxers, fem un `#include` d'un fitxer que les conté. Aquest fitxer no està en els directoris estàndard on el compilador busca els fitxers de capçalera, i per això cal posar el seu nom entre cometes en lloc dels parèntesis angulars.

El procés de generació dels executables per a aquests dos programes es troba esquematitzat a la Figura 5.2. A sota de cada objecte s'indiquen quines són les referències sense resoldre (funcions no definides al corresponent codi font). Noteu que algunes d'elles (`sscanf()`, `fprintf()`, `printf()`, `floor()`) no estan definides a cap dels nostres mòduls (però sí declarades!), sinó que es troben a les *biblioteques del sistema*[7]. D'aquí que el pas per codi objecte sigui necessari fins i tot per a programes que consten d'un únic mòdul.

Podem generar els executables de diverses maneres. En primer lloc, podem fer:

[7]Vegeu l'apartat 5.5.

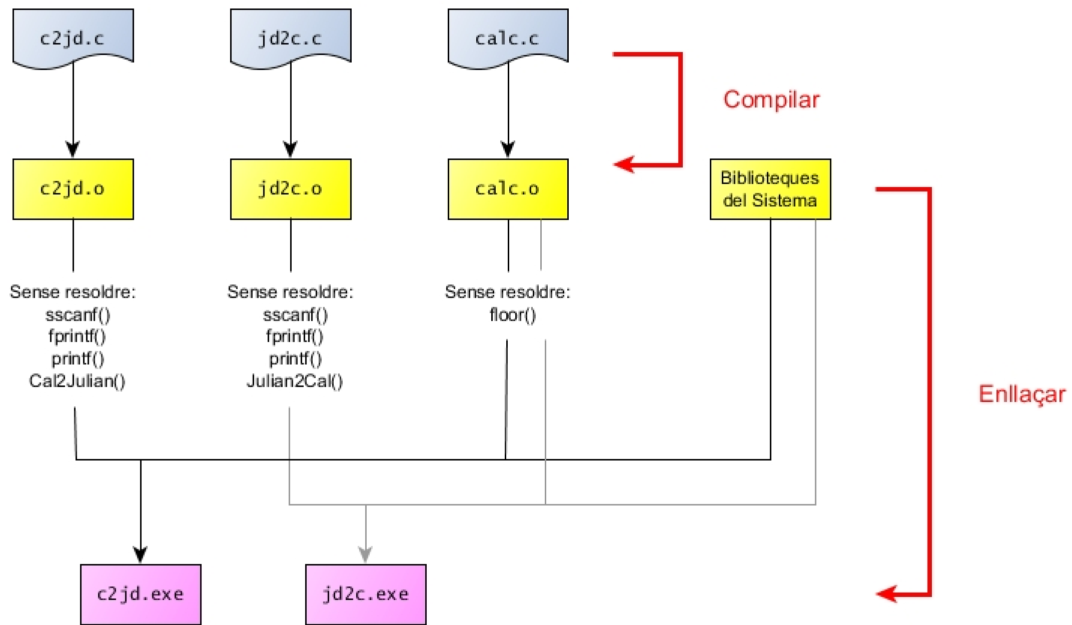


Figura 5.2: Esquema de la generació dels programes `c2jd.exe` i `jd2c.exe` a partir dels seus mòduls.

```
gcc -o jd2c.exe -Wall jd2c.c calc.c -lm
gcc -o c2jd.exe -Wall c2jd.c calc.c -lm
```

Cada executable depèn de dos fitxers font `.c`, que hem de passar per tant al `gcc` com a arguments. L'única regla aquí és que hi hagi entre tots els mòduls una funció anomenada `main()`, i només una.

El switch `-lm` hi és per tal d'enllaçar amb la biblioteca de funcions matemàtiques, on hi ha la definició de la funció `floor()` (en algunes instal·lacions del `gcc` aquesta biblioteca s'enllaça automàticament, i en tal cas no caldria explicitar-ho).

Una segona opció és fer la compilació i l'enllaçat en passos separats:

`gcc -c -Wall c2jd.c`	compila `c2jd.c` → `c2jd.o`
`gcc -c -Wall jd2c.c`	compila `jd2c.c` → `jd2c.o`
`gcc -c -Wall calc.c`	compila `calc.c` → `calc.o`
`gcc -o c2jd.exe c2jd.o calc.o -lm`	enllaça `c2jd.o calc.o` → `c2jd.exe`
`gcc -o jd2c.exe jd2c.o calc.o -lm`	enllaça `jd2c.o calc.o` → `jd2c.exe`

La opció `-c` fa que el `gcc` faci només la fase de compilació, i no enllaci. Fet així, els fitxers de codi objecte `.o` no s'esborren al final del procés.

Com a tercera opció tenim el mecanisme dels *makefile*, molt útil quan el projecte involucra diversos mòduls. Els *makefile* són uns fitxers escrits en un llenguatge senzill de scripting, que són processats per un programa anomenat `make`.

El Llistat 5.7 mostra el contingut d'un fitxer de nom `makefile` per "fer" el projecte de conversió de dates.

Un fitxer makefile està format per estructures del tipus

```
objectiu : dependències
<TAB>  ordres per generar ‘objectiu’ a partir de ‘dependències’
```

És imprescindible que totes les línies d’ordre comencin per un caràcter tabulador.

Així, el fitxer `makefile` que hem escrit especifica

- que el fitxer `jd2c.exe` es genera a partir de `jd2c.o` i `calc.o` amb l’ordre
  `gcc -o jd2c.exe jd2c.o calc.o -lm`
- que `jd2c.o` es genera a partir de `jd2c.c` amb l’ordre
  `gcc -c -Wall jd2c.c`

i així successivament. L’objectiu `all` té dependències però no té cap ordre associada. L’objectiu `clean` no té dependències, però sí que té una ordre associada (que esborra tot el codi objecte intermedi). Observeu que hi ha objectius que són fitxers (com `jd2c.exe` i `jd2c.o`), i objectius que són “etiquetes” (com `all` i `clean`).

Per tal de “fer” un objectiu, hem de cridar `make` amb l’objectiu al costat. Per exemple, l’ordre

```
make jd2c.exe
```

genera l’executable `jd2c.exe`. Quan es crida `make` sense arguments, es fa el primer objectiu del fitxer; en aquest cas, és equivalent a fer `make all`, i es generen els dos executables.

El programa `make` només engega les ordres que són estrictament necessàries. Imaginem que modifiquem només el fitxer `c2jd.c`. Si fem després `make`, veurem que no es recompilen `calc.c` ni `jd2c.c`, perquè no és necessari.

`make` busca un fitxer que s’anomeni `makefile` en el directori actual. Si volem tenir més d’un fitxer d’aquests en un directori, els podem posar noms diferents i passar-los a `make` com a paràmetre:

`make -f NomMakeFile`.

5.4. Scope

En un programa llarg o en un projecte amb diferents fitxers, tindrem problemes amb el nom de les variables si no els podem reutilitzar. Per això hi ha el concepte de *scope*[8] *d’un identificador*. (Els identificadors són els noms de variables i funcions que nosaltres escollim.) La part de programa on un identificador pot ser referenciat és el scope d’aquell identificador; és a dir, la part de programa on és reconegut i podem usar-lo.

5.4.1 Scope en C

En C, el scope d’un identificador comença en la seva declaració; finalitza en diferents llocs segons on aparegui la declaració:

- Si la declaració apareix dins un *bloc*, és a dir entre una clau que obre `{` i una clau que tanca `}`, el scope acaba al acabar el bloc. Es diu que l’identificador té un scope de bloc o *local*. En particular, el codi d’una funció està contingut en un bloc. Per tant, les variables declarades dins del codi de la funció són locals d’aquella funció. No són accessibles fora d’ella.

[8]Possibles traduccions: abast, camp d’acció, àmbit.

- Si la declaració no apareix dins de cap bloc, el scope acaba al final del fitxer. Es diu que té un scope de fitxer o *global.* Si hem usat directives `#include`, el contingut dels fitxers inclosos es considera part del fitxer també.

Si es declara altre cop un identificador dins un bloc estrictament contingut en el seu scope, aquest queda "suspès", i l'identificador es referirà a un nou objecte local fins que acabi el bloc. Acabat el bloc, l'objecte més exterior torna a ser visible.

Suposem per exemple un fitxer amb el contingut següent:

```
double x=0.1, z=1.5;
int f( int y) {
  int x = y+1;
  return x*z;
  }
int main() {
  z = f(3) + x;
  }
```

Les variables `x` i `z` declarades com a `double` al principi del fitxer tenen un scope global. Dins del bloc de definició de la funció `f()` es declara una altra variable de nom `x`, que és de tipus `int`. Per tant, la referència a `x` en `return x*z` es refereix a la variable local, mentre que la referència en `z = f(3) + x` es refereix a la variable global (`double`). L'identificador `z`, tant dins com fora de la funció es refereix al mateix objecte.

L'us de variables amb scope global està desaconsellat, i sempre hi ha alternatives.

El temps de vida d'una variable és el temps de vida del scope on està definida: quan l'execució surt d'aquest scope, la memòria que ocupava la variable queda lliure, a disposició del sistema operatiu.

Per les regles que acabem de veure, quan cridem a una mateixa funció per segona vegada en el mateix programa, totes les variables declarades en la instància anterior (incloses les de la llista de paràmetres), han deixat d'existir. En algunes ocasions molt particulars, ens pot interessar que una variable persisteixi entre crides successives a una funció. Això és possible declarant-la com a `static`. Per exemple:

```
int func() {
  static int n = 0;
  n = n + 1;
  return n
  }
```

La inicialització `n=0` només es fa la primera vegada. La funció `func()` retorna quantes vegades hem cridat a la pròpia funció.

Podem allargar el scope d'una variable global més enllà d'un fitxer. Per entendre la idea, va bé distingir conceptualment entre declarar variables i definir variables, igual com amb funcions: La instrucció

```
extern int n;
```

declara una variable, però no la defineix; diu que està definida en un altre lloc. Al llegir una declaració, de variable o de funció, el compilador pren

nota del nom i el tipus en una taula de noms. Quan en llegeix la definició, aleshores realment assigna memòria a la variable, o genera realment el codi de la funció.

5.4.2 Scope en Python

En Python podem usar una variable "global" des de dins d'una funció directament. Per exemple

```
def func():
    print(z)
z = 2
func()
print(z)
#Imprimeix 2 2
```

Podem crear una variable dins d'una funció amb el mateix nom que una variable de fora. Igual que amb el C, la variable de fora queda "a l'ombra" (*variable shadowing*) mentre s'executa la funció:

```
def func():
    z = 1
    print(z)
z = 2
func()
print(z)
#Imprimeix 1 2
```

Fins i tot podem usar una variable dins una funció i cridar la funció des d'una altre funció que conté una variable amb el mateix nom:

```
def func1():
    z = 1
    print(z)
def func2():
    z = 2
    func1()
    print(z)
z = 3
func2()
print(z)
#Imprimeix 1 2 3
```

Però no podem usar variable global dins d'una funció i després crear-ne una amb el mateix nom:

```
def func():
    print(z)
    z = 1
z = 2
func()
print(z)
#Dóna error
```

Variables estàtiques. Es pot definir una variable estatica per una funció associada al nom de la funció definint-la com `nom_de_la_funció.nom_de_la_variable`

```
def func():
    func.i += 1
    print(func.i)

func.i = 0;
print(func(), func(), func())
# Imprimeix 1,2,3
```

5.5. Biblioteques

Les biblioteques (*libraries*) són fitxers contenint funcions ja compilades, llestes per enllaçar. Una biblioteca pot tenir dues versions: La versió *estàtica* (*static library*) i la versió *dinàmica* (*dynamic library* o *shared library*).

Les biblioteques estàtiques són fitxers amb extensió `.a`. L'anomenada *C standard library* es troba en un fitxer de nom `libc.a` i el `gcc` sempre l'enllaça a tot programa, de manera que no cal fer-ho explícitament[9]. La funció `printf()`, per exemple, està definida en aquesta biblioteca.

La biblioteca de funcions matemàtiques és al fitxer `libm.a`, i en principi sí cal demanar explícitament que s'enllaci. Si, per exemple, es troba en el directori `/usr/lib/`, es pot escriure

```
gcc -Wall -o hola.exe /usr/lib/libm.a hola.c
```

o, de manera més abreviada, amb el switch `-l`,

```
gcc -Wall -o hola.exe -lm hola.c
```

i no cal posar ni el path, ni l'extensió `.a` ni les primeres lletres `lib`. En general, si una biblioteca està en el fitxer `libmystuff.a`, només cal escriure `-lmystuff` en la crida al `gcc`.

Les biblioteques dinàmiques es caracteritzen pel fet que el seu codi objecte no es copia al fitxer executable en el moment d'enllaçar. El fitxer executable serà sensiblement més petit, però en el moment que s'executi cal que la biblioteca estigui present a l'ordinador i es carregui en memòria en aquell moment. D'aquesta manera, una biblioteca dinàmica en memòria es pot compartir entre diferents programes i s'estalvien recursos.

Les biblioteques dinàmiques es troben en fitxers amb extensió `.so`, de *shared object* (en Linux), o bé `.dll`, de *dynamic link library* (en Windows).

Si no se li diu el contrari, el `gcc` busca una biblioteca dinàmica, i l'enllaç en sí es produirà en temps d'execució (*dynamic linking*). Si es vol forçar l'enllaç estàtic, es pot posar el path sencer al fitxer concret `.a`, o bé usar el switch `-static`:

```
gcc -Wall -o hola.exe -static -lmystuff hola.c
```

---

[9] Segons el compilador, pot tenir un altre nom, com ara `libgcc.a`

Exercicis

37. Com hem vist a l'Apartat 5.3, el `gcc` és en realitat un programa genèric que fa d'interfície als programes que realment fan la feina. Per veure tot el que fa el `gcc`, useu el switch `-v` (de "verbose"): Escriviu un programa que simplement digui "hola" per pantalla, en un fitxer `hola.c` i feu

```
gcc -v -Wall -o hola.exe hola.c
```

La informació que surt és atapeïda, però amb una mica de paciència es veu que primer es crida el programa `cc1`, que és realment el compilador de `C`. Aquest produeix un fitxer temporal amb extensió `.s` en llenguatge assemblador. Després crida el `as`, que produeix a partir del `.s` un fitxer temporal en codi objecte `.o`. Finalment crida el programa `ld`, que és l'enllaçador, que a partir del `.o` produeix l'executable `.exe`. És possible que en lloc de `ld` aparegui `collect2`, que en determinades ocasions fa alguna operació prèvia i després crida al `ld`.

GCC (GNU Compiler Collection) és en realitat una col·lecció de programes que no només construeixen programes escrits en `C`, sinó en alguns altres llenguatges, com ara `C++`, `Fortran`, `Java` i `Pascal`.

38. Es compilarà bé el programa següent en C? Per què?

```
#include <stdio.h>
int main() {
  {int x = 3;
   x = 2 * x;}
  printf("%d", x);
  }
```

Comproveu-ho.

39. Quantes variables de nom `a` es declaren en la funció següent? Quantes prenen en algun moment el valor 2? Quin valor retorna la funció?

```
int func() {
  int a = 0;
  a = a + 1;
  for (int i=0; i<10; i++) {
    int a = 1;
    a = a + 1;
    }
  return a;
  }
```

40. Considereu la variant següent del problema anterior: En un fitxer tenim aquest codi. Què imprimirà quan s'executi?

```
#include <stdio.h>

int func() {
  int a = 0;
  a = a + 1;
  for (int i=0; i<10; i++) {
    static int a = 1;
    a = a + 1;
    printf("%d\n", a);
  }
  return a;
}

int main() {
  printf("%d", func());
}
```

```
1  #include <math.h>
2
3  /* Canvi de data de calendari a dia julià */
4  double Cal2Julian(double year,
5  double month, double day) {
6  double a, b;
7  if (month<=2) {
8  year=year-1; month=month+12;
9  }
10 if (year + 0.01*month + 0.0001*day >= 1582.1015) {
11 a = floor(year/100);
12 b = 2-a+floor(a/4);
13 }  /* correccio reforma gregoriana */
14 else { b = 0;}
15 return floor(365.25*(year+4716)) +
16 floor(30.6001*(month+1)) + day + b - 1524.5;
17 }
18
19 /* Canvi de dia julià a data de calendari */
20 void Julian2Cal (double jd, double *yy, double *mm,
21 double *dd) {
22 double z, f, b, c, d, e;
23 jd = jd+0.5;
24 z = floor(jd);
25 f = jd-z;
26 b = z;
27 if (z >= 2299161) {
28 double alf = floor((z-1867126.25)/36524.25);
29 b = b + 1 + alf - floor(alf/4);
30 } /* correccio reforma gregoriana */
31 b = b + 1524;
32 c = floor((b-122.1)/365.25);
33 d = floor(365.25*c);
34 e = floor((b-d)/30.6001);
35 *dd = b - d - floor(30.6001*e) + f;
36 if (e<14) {*mm = e-1;}
37 else {*mm = e-13;}
38 *yy = 4716;
39 if (*mm<=2) *yy = *yy-1;
40 *yy = c - (*yy);
41 }
42
```

Llistat 5.3: Codi del fitxer calc.c

```
#include <stdio.h>
#include "calc.h"

int main (int argc, char *argv[]) {
double jd, y, m, d;
if (argc!=2 || sscanf(argv[1], "%lf", &jd)!=1) {
fprintf(stderr, "%s jd\n", argv[0]);
return -1;
}
Julian2Cal(jd, &y, &m, &d);
printf("%.16G %.16G %.16G\n", y, m, d);
return 0;
}
```

Llistat 5.4: Codi del fitxer `jd2c.c`

```
#include <stdio.h>
#include "calc.h"

int main (int argc, char *argv[]) {
double d, m, y;
if (argc<4
|| sscanf(argv[1], "%lf", &y)!=1
|| sscanf(argv[2], "%lf", &m)!=1
|| sscanf(argv[3], "%lf", &d)!=1) {
fprintf(stderr, "%s y m d\n", argv[0]);
return -1;
}
printf("%.16G\n", Cal2Julian(y, m, d));
return 0;
}
```

Llistat 5.5: Codi del fitxer `c2jd.c`

```
/* Canvi de data de calendari a data juliana */
double Cal2Julian (double, double, double);

/* Canvi de data juliana a data de calendari */
void Julian2Cal (double, double *, double *,
double *);
```

Llistat 5.6: Codi del fitxer `calc.h`

```
# Tot
all: jd2c.exe c2jd.exe

# Fitxers objecte i executables
jd2c.exe: jd2c.o calc.o
	gcc -o jd2c.exe jd2c.o calc.o -lm
jd2c.o: jd2c.c
	gcc -c -Wall jd2c.c
c2jd.exe: c2jd.o calc.o
	gcc -o c2jd.exe c2jd.o calc.o -lm
c2jd.o: c2jd.c
	gcc -c -Wall c2jd.c
calc.o: calc.c
	gcc -c -Wall calc.c

# Neteja
clean:
	rm c2jd.o jd2c.o calc.o
realclean: clean
	rm jd2c.exe c2jd.exe

```

Llistat 5.7: Fitxer makefile per “fer” el convertidor de dates

6

Programació Orientada a Objectes

Un dels criteris de classificació dels llenguatges de programació respon al "paradigma" de programació a què donen suport. Un d'aquests paradigmes o estils s'anomena *programació orientada a objectes* (Object Oriented Programming, OOP).

En les últimes dècades, els programes han esdevingut cada cop més grans i complicats. Per exemple, el codi de base de Microsoft conté més de 50 milions de línies de codi. Està clar que això només es pot mantenir treballant amb equips molt grans; per tant és necessari fer programes que siguin clars i llegibles per tothom. La programació orientada a objectes ajuda a fer això possible.

Tant el C++ com el Python i el Java tenen les facilitats necessàries per practicar la OOP. El C++ es pot pensar com una ampliació del llenguatge C amb aquest propòsit. El llenguatge Python, amb un creixement de popularitat molt important en els últims tres anys[1], es basa de manera fonamental en aquesta idea[2]

En aquest capítol veurem una petita introducció a la programacio orientada a objectes, usant el C++ i el Python com exemples.

6.1. Les estructures en C

Com a primer pas per entendre què són els "objectes", veiem primer una construcció que està disponible ja en el C estàndard: Les *estructures*, que són tipus de dades compostos d'altres tipus de dades més simples. Per exemple,

```
struct fruita {
  int pes;
  float preu;
  };
```

defineix un nou tipus de dada, anomenat `fruita`, que està compost d'un `int` i un `float`. Després podem declarar variables d'aquest nou tipus, de la manera

[1] Tiobe `https://www.tiobe.com/tiobe-index/` és un índex de popularitat de llenguatges molt reconegut

[2] És important saber que el Python 3 i el Python 2 no són llenguatges compatibles. El Python 2.7 encara està en manteniment fins el 2020; però òbviament, tot projecte nou s'ha de començar en Python 3.

habitual:

```
struct fruita poma;
struct fruita taronja, pera;
```

Per accedir a cadascun dels *membres* de `poma`, `taronja`, `pera`, s'usa un punt '`.`', que s'anomena *operador d'accés a membres*:

```
poma.preu = 0.75;
k = poma.pes;
```

Sabem que es poden usar apuntadors a qualsevol tipus de variable; en particular també a les variables que són estructures:

```
struct fruita* platan;
```

defineix un apuntador a una variable de tipus `fruita`. Per accedir als components s'escriu

```
(*platan).preu = 0.60;
```

Els parèntesis en aquest cas són necessaris perquè l'operador d'accés '`.`' té precedència sobre l'operador de contingut '`*`'. Sense els parèntesis, la instrucció seria equivalent a `*(platan.preu) = 0.60;`, que donaria un error en aquest context. Es pot usar també la construcció alternativa

```
platan -> preu = 0.60;
```

6.2. Les classes en C++

Tot el que segueix és propi del C++ i no funciona en C. Els conceptes són comuns a altres llenguatges amb orientació a objectes, cadascun amb la seva sintaxi pròpia.

El compilador `gcc` compila també C++. L'opció de llenguatge s'estableix amb el switch `-x`:

```
gcc -x c++ -o hola.exe hola.cpp
```

(és costum usar el sufix `.cpp` per als fitxers font en C++), però és més habitual i ràpid escriure a la consola l'ordre equivalent

```
g++ -o hola.exe hola.cpp
```

El concepte de *classe* és una extensió del de *estructura*. Les classes poden tenir com a membres no sols variables, sinó també funcions. Aquesta idea implementa un concepte abstracte que s'anomena *encapsulació*: l'agrupació de les dades amb els mètodes que operen sobre aquestes dades en un sol ens, impedint que la informació continguda pugui ser accedida des d'altres elements del programa externs a la classe.

Un *objecte* és una "instància" d'una classe: es pot pensar que la classe defineix un tipus de variable, i que l'objecte és la variable. Per exemple, el codi

```
class Rectangle {
 private:
  int x, y;
 public:
  void setDimen(int, int);
  int computeArea();
  };

Rectangle rect;
```

defineix una classe anomenada `Rectangle`, que té per membres dues variables enteres `x`, `y`, i dues funcions, `setDimen()` que accepta dos `int` i no retorna res, i `computeArea()`, que no accepta arguments i retorna un `int`. Després s'instancia (es declara) un objecte de la classe `Rectangle`, anomenat `rect`.

L'*especificador d'accés* `public:` indica que als membres que van declarats després s'hi podrà accedir des de fora de la definició de la classe, mentre que els que porten l'especificador `private:` (o no porten res), les variables `x` i `y` en aquest cas, només podran ser llegits i modificats a través d'altres membres de la classe.

La idea és que les variables `x` i `y` continguin la llargada i l'alçada del rectangle, la funció `setDimen()` estableixi valors per a aquestes variables, i la funció `computeArea()` calculi l'àrea del rectangle. Per tant, ara hem de definir aquestes dues funcions, que dins de la classe només estan declarades.

```
void Rectangle::setDimen(int a, int b) {
  x = a;
  y = b;
  }

int Rectangle::computeArea() {
  return x*y;
  }
```

La única diferència amb les definicions de funcions a què estem acostumats és el *operador de scope* `::` precedit del nom de la classe. Això és necessari perquè podem tenir definides altres classes contenint funcions membres amb els mateixos noms.

L'objecte `rect` pot accedir a les seves funcions `setDimen()` i `computeArea()` amb el mateix operador de pertinença '`.`' que hem vist per a les estructures. Per exemple:

```
rect.setDimen(3, 4);
s = rect.computeArea();
```

estableix els valors 3 i 4 per a les variables `x` i `y` pròpies de l'objecte `rect`, després en calcula l'àrea, i la diposita a la variable `s`. Observem que no cal passar-li les variables; les `x` i `y` que apareixen a la definició de `computeArea()` només poden ser les de l'objecte `rect`.

Naturalment, podem definir tots els objectes que vulguem d'una classe. La Figura 6.1 és un programa complet que defineix dos objectes de la classe `Rectangle`, n'estableix les dimensions, i calcula i escriu llurs àrees.

Està clar que si s'intenta executar la funció `computeArea()` abans de la funció `setDimen()` el resultat és imprevisible, perquè les variables `x` i `y` de l'objecte

```
1  #include <stdio.h>
2
3  /* Definició de la classe Rectangle */
4  class Rectangle {
5    int x, y;
6   public:
7    void setDimen(int, int);
8    int computeArea();
9    };
10
11 /* Definició de les funcions de Rectangle */
12 void Rectangle::setDimen(int a, int b) {
13   x = a;
14   y = b;
15   }
16
17 int Rectangle::computeArea() {
18   return x*y;
19   }
20
21 /* main */
22 int main() {
23   Rectangle rectA, rectB; // Declara dos objectes
24                           // de la classe Rectangle
25   rectA.setDimen(3, 4);
26   rectB.setDimen(7, 6);
27   printf("%d %d", rectA.computeArea(),
      rectB.computeArea());
28   return 0;
29   }
```

Llistat 6.1: Un programa complet en C++ il·lustrant la OOP.

no hauran estat inicialitzades. Per evitar-ho, es pot usar una funció especial anomenada *constructor*, que es crida automàticament quan es crea un nou objecte de la classe. Aquesta funció constructor ha de tenir el mateix nom de la classe i no ha de retornar cap tipus, ni tan sols `void`. A més, es una funció que no es pot cridar explícitament.

Continuant el nostre exemple, la funció constructor substituiria a la funció `setDimen()` i es definiria així:

```
Rectangle::Rectangle(int a, int b) {
  x = a;
  y = b;
  }
```

La definició de la classe quedaria

```
class Rectangle {
  int x, y;
 public:
  Rectangle(int, int);
```

```
int computeArea();
};
```

i la construcció dels dos objectes d'aquesta classe seria així:

```
Rectangle rectA(3, 4);
Rectangle rectB(7, 6);
```

La funció anomenada *destructor* té la funcionalitat oposada al constructor. Es crida automàticament quan un objecte és destruït; per exemple, quan acaba el seu *scope*[3]. És útil definir-la explícitament quan cal fer alguna acció addicional en el moment de destruir l'objecte. Ha de tenir el mateix nom de la classe, precedit per una titlla ~.

Una altra de les diferències importants entre el C i el C++ és la possibilitat en aquest últim de *sobrecarregar* funcions. Això vol dir que podem tenir diverses funcions amb el mateix nom però amb arguments o tipus de retorn diferents, i el compilador sabrà distingir quina d'elles s'està cridant a partir de com es fa la crida.

El llistat de la Figura 6.2 incorpora al programa de la Figura 6.1 els elements anteriors: Incorporem un constructor substituint a la funció `setDimen()`; el sobrecarreguem amb un altre constructor alternatiu que ens permetrà establir uns valors per omissió de les dimensions d'un rectangle; i finalment afegim un destructor que ens informarà per pantalla de la destrucció d'un objecte.

6.3. Les classes en Python

En aquest apartat farem un exemple de programació orientada a objectes en Python; els nostres objectes seran triangles.

Primer de tot hem de decidir què és un triangle. Hem optat per a definir-lo com un conjunt de tres punts del pla $\mathbb{R}^2$ (això inclourà els triangles "degenerats" formats per tres punts alineats). Així, un triangle es crearà donant tres punts, com ara

```
T1=Triangle([0,0],[0,12],[16,12])
```

Per definir la classe hem d'usar una funció `__init__` donant com a paràmetres el nom amb que la referirem internament (tradicionalment s'utilitza `self`, però no caldria!), i les dades que les defineixen (els tres punts). També hi podem afegir les funcions convenients[4]; en aquest cas, la funció `area()`, que calcularà l'àrea del triangle donat pels tres vèrtexs.

```
1  class Triangle:
2      def __init__(self, punt1,punt2,punt3):
3          self.__p1 = punt1
4          self.__p2 = punt2
5          self.__p3 = punt3
6          self.vertexs=[punt1,punt2,punt3]
7      def area(self):
8          p1=self.__p1
9          p2=self.__p2
10         p3=self.__p3
```

---

[3] Vegeu Apartat 5.4.

[4] En el context de la OOP les funcions membres d'una classe s'anomenen també *mètodes*

```
11        p123=(p2[0]-p1[0])*(p3[1]-p1[1])
12        p132=(p2[1]-p1[1])*(p3[0]-p1[0])
13        area=abs((p123-p132)/2)
14        return(area)
```

Observem que les variables `__p1`, `__p2` i `__p3` comencen amb un doble guió baix. Això vol dir que seran variables privades de la classe. Per tant no es podran referenciar des de fora d'ella. La variable `vertexs` és pública. Si ara fem

```
1 T1=Triangle([0,0],[0,12],[16,12])
2 print(T1.area())
3 print(T1.vertexs)
```

hem creat un triangle format pels vèrtexs $(0,0)$, $(0,12)$ i $(16,12)$, i hem calculat i imprès la seva àrea i els seus vèrtexs. Al llistat 6.3 hem incorporat altres funcions a la classe Triangle.

Després, podem calcular l'àrea, la llargada dels costats, el baricentre, i el circumradi:

```
1 T1=Triangle([0,0],[0,12],[16,12])
2 print(T1.area())
3 print(T1.costats())
4 print(T1.baricentre())
5 print(T1.circumradi())
```

En OOP és molt útil *derivar* classes a partir d'altres classes, afegint-hi elements. Aquesta propietat s'anomena *herència*. Per exemple, podem definir la classe dels triangles rectangles donant només els dos catets (que situarem amb un vèrtex a l'origen).

```
1 class TriangleRectangle(Triangle):
2     def __init__(self, catet1,catet2):
3         Triangle.__init__(self,[0,0], [0,catet1],
     [catet2,catet1])
4         self.catet1=catet1
5         self.catet2=catet2
6     def hipotenusa(self):
7         return((self.catet1**2+self.catet2**2)**0.5)
```

Un objecte de la classe `TriangleRectangle` té tots els atributs de la classe `Triangle` i a més una funció `hipotenusa()` que no tenen els objectes de la classe `Triangle`.

```
1  #include <stdio.h>
2
3  /* Definició de la classe Rectangle */
4  class Rectangle {
5    int x, y;
6   public:
7    Rectangle(int, int);
8    Rectangle();
9    ~Rectangle();
10   int computeArea();
11   };
12
13 /* Definició de les funcions de Rectangle */
14 Rectangle::Rectangle(int a, int b) {
15   x = a;
16   y = b;
17   }
18
19 Rectangle::Rectangle() {
20   x = 5;
21   y = 5; // per omissió, un rectangle serà
22          // un quadrat de costat 5
23   }
24
25 Rectangle::~Rectangle() {
26   printf("\nHem destruit un rectangle!");
27   }
28
29 int Rectangle::computeArea() {
30   return x*y;
31   }
32
33 /* main */
34 int main() {
35   Rectangle rectA(3, 4);
36   Rectangle rectB;
37   printf("%d %d", rectA.computeArea(),
38                   rectB.computeArea());
39   return 0;
40   }
```

Llistat 6.2: Modificació del programa de la Figura 6.1, incorporant constructor, sobrecàrrega de funcions, i destructor. El programa imprimeix les àrees dels rectangles `rectA` i `rectB` i després el missatge de la línia 26 dues vegades.

```
1 import numpy
2
3 def distancia(ps):
4     return((sum((x-y)**2 for x,y in zip(*ps))**0.5))
5
6 class Triangle:
7     def __init__(self, punt1,punt2,punt3):
8         self.__p1 = punt1
9         self.__p2 = punt2
10        self.__p3 = punt3
11        self.vertexs=[punt1,punt2,punt3]
12    def area(self):
13        p1 = self.__p1
14        p2 = self.__p2
15        p3 = self.__p3
16        p123 = (p2[0]-p1[0])*(p3[1]-p1[1])
17        p132 = (p2[1]-p1[1])*(p3[0]-p1[0])
18        area = abs((p123-p132)/2)
19        return(area)
20    def costats(self):
21        p = self.vertexs
22        pp=[[p[j] for j in range(3) if j!=i] for i
       in range(3)]
23        return [distancia(ps) for ps in pp]
24    def perimetre(self):
25        return sum(self.costats())
26    def baricentre(self):
27        p=self.vertexs
28        baricentre=[sum(a)/3 for a in zip(*p)]
29        return(baricentre)
30    def inradi(self):
31        return(2*self.area()/self.perimetre())
32    def circumradi(self):
33
       return(numpy.prod(self.costats())/(4*self.area()))
```

Llistat 6.3: La classe Triangle amb algunes funcions. Hem utilitzat la funció `zip()` que agafa dos llistes i les converteix en una llista de parelles, formades per l'element enèsim de cada llista. A més hem aplicat el operador $*$ a la llista dels punts per a "esborrar" els parèntesis. Importem també la biblioteca `numpy` per usar una funció que calcula el producte dels elements d'una llista.