Organitzant el codi. Estructures algorísmiques

PID_00275246

Autors que han participat col·lectivament en aquesta obra

Lluís Beltrà
Kenneth Capseta
Maria Jesús Marco Galindo
Antonio Ponce

Idea i direcció de l'obra

Maria Jesús Marco Galindo

Disseny i edició gráfica

Asunción Muñoz

Material docent de la UOC

Tercera edició: febrer 2021
© Luis Beltrà Valenzuela, Kenneth Capeseta Nieto, Antonio Ponce Tarela, Asunción Muñoz Fernández, Maria Jesús Marco Galindo
Tots els drets reservats
© d'aquesta edició, FUOC, 2020
Av. Tibidabo, 39-43, 08035 Barcelona
Realització editorial: FUOC

Cap part d'aquesta publicació, incloent-hi el disseny general i la coberta, no pot ser copiada, reproduïda, emmagatzemada o transmesa de cap manera ni per cap mitjà, tant si és elèctric com químic, mecànic, òptic, de gravació, de fotocòpia o per altres mètodes, sense l'autorització prèvia per escrit dels titulars dels drets.

Continguts

Organitzant el codi
Estructures algorísmiques
Estructures algorísmiques: composició d'accions
L'assignació
Sintaxi de l'acció fonamental d'assignació
Composició d'accions
1. Estructura seqüencial
2. Estructura alternativa
Sintaxi de l'estructura alternativa o condicional
Què passaria si…
Estructures alternatives més elaborades
3. Estructura iterativa
Sintaxi de l'estructura iterativa o repetitiva
Què passaria si…
Una altra estructura iterativa
Estructura d'un algorisme
Sintaxi de l'estructura d'un algorisme
Conceptes clau
Com treballem les estructures en JavaScript?
Sintaxi de les estructures alternatives en JS
Sintaxi de les estructures iteratives en JS
Provar el codi en JS
Què passaria si…

Organitzant el codi
Estructures algorísmiques

Quan ja es tenen clares les dades que ha de tractar un algorisme (o un programa) i com representar-les, el següent pas és determinar els passos a seguir per a resoldre el problema que ens planteja l'enunciat (estratègia), i concretar-los en les instruccions i sentències de l'algorisme.

Així podrem completar el pas 3: **dissenyar l'algorisme**. Però, com es construeix un algorisme?

Exemple

Reprenem l'algorisme d'Euclides per **calcular el màxim comú divisor** entre dos números diferents de zero.

1 Entenem el problema

D'entrada, tindrem dos números positius i haurem de calcular un número també enter positiu que serà el màxim comú divisor entre els dos de l'entrada.

2 Pensem com resoldre'l

L'estratègia de solució ja la tenim detallada a la unitat anterior, seguint l'algorisme d'Euclides. Primer, cal comprovar quin dels dos números d'entrada és més gran, que serà el dividend, l'altre serà el divisor i després, recordem-ho:

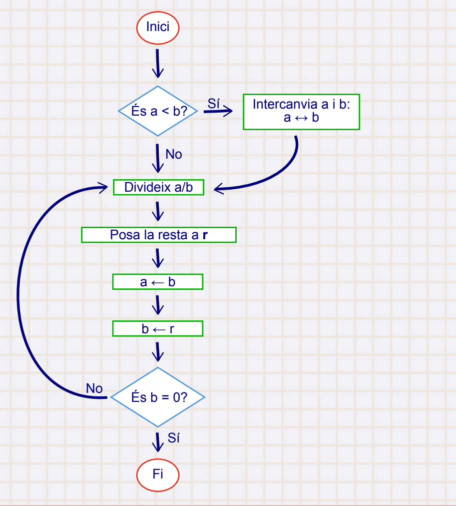
3 Dissenyem l'algorisme

A la unitat 1 hem fet el diagrama de flux que representa l'estratègia de solució. Vegem ara com fer l'algorisme, que serà el pas previ a poder programar la solució.

a=126
b=72

Ja tenim clar el problema a resoldre.

Ja tenim l'estratègia per calcular el mcd.



3 Dissenyem l'algorisme

Primera qüestió: com representem i obtenim les dades?

Amb el que hem vist a la unitat anterior, tindríem:

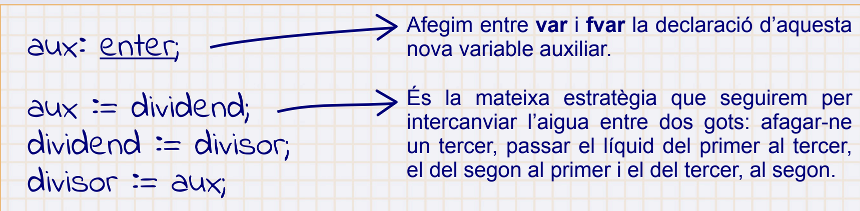
```
var
    dividend : enter;
    divisor : enter;
    residu : enter;
fvar

dividend := llegirTeclat();
divisor := llegirTeclat();
```

Les dades d'entrada.

Segona qüestió: indicar les instruccions per fer els càlculs. Algunes expressions ja les sabem indicar. Per exemple, com comprovar si el dividend és més petit que el divisor (ja que, en aquest cas, els hauríem d'intercanviar per poder començar les divisions). L'expressió seria:

```
(dividend < divisor)
```

Si aquesta expressió resultés certa, seguint l'algorisme d'Euclides hauríem d'intercanviar els valors. Per això necessitem definir una **variable intermèdia** que anomenarem "aux" (variable auxiliar):



Recordeu aquesta estratègia sempre que us calgui intercanviar els valors de dues variables.

Hem utilitzat l'assignació **:=** i ja tenim les tres primeres accions de l'algorisme (que finalment seran les instruccions del programa).

L'**assignació** és l'acció més bàsica i fonamental sense la qual no es pot fer cap algorisme.

I també podem escriure l'expressió que comprova si el residu és zero:

```
residu = 0
```

Fins aquí, doncs, ja hem combinat diferents elements de l'algorísmica que coneixem:

- variables,
- expressions,
- acció elemental d'assignació.

També hem utilitzat funcions predefinides d'entrada/sortida per llegir les dades del problema.

Podríem, a més, plantejar la divisió a fer: el que ens interessa és calcular el residu, això ho podem fer amb l'operador **mod** que calcula el mòdul o residu d'una divisió:

```
residu := dividend mod divisor;
```

Hem vist aquest operador a la unitat anterior.

Però ja no podem continuar. Per fer-ho ens cal saber com combinar aquestes accions o sentències, per exemple per poder indicar que, si el dividend és més petit que el divisor, cal intercanviar-los, però si no ho és no, o que cal anar repetint les divisions fins que arribem al residu 0.

Per això ens cal conèixer les estructures algorísmiques: seqüencial, alternativa o iterativa.

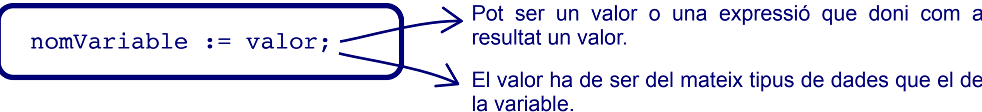
Abans de continuar, fixeu-vos-hi, què ha passat? Hem definit les variables, i hem començat a llegir les dades i fer els càlculs, un darrere l'altre. Però ha arribat un moment que ha calgut una variable auxiliar per intercanviar els dos valors. Llavors ha fet falta recular, anar a la zona de definició de variables i afegir aquesta variable *aux*. Així doncs, els algorismes no s'escriuen des de la primera línia a la darrera, tot seguit, sinó que **cal anar tirant endarrere i endavant per anar refinant i completant el codi.**

Estructures algorísmiques: composició d’accions

L’assignació

Com ja s’ha vist, l’**assignació** és l’acció fonamental en l’algorísmica. Consisteix a assignar (donar, guardar) un valor concret dins d’una variable.

Sintaxi de l’acció fonamental d’assignació



Composició d’accions

Només hi ha tres maneres de combinar les instruccions en un algorisme: en forma de **seqüència**, d’**alternativa** o de **repetició**.

1. **Seqüencial**
2. **Alternativa o condicional**
3. **Iterativa o repetitiva**

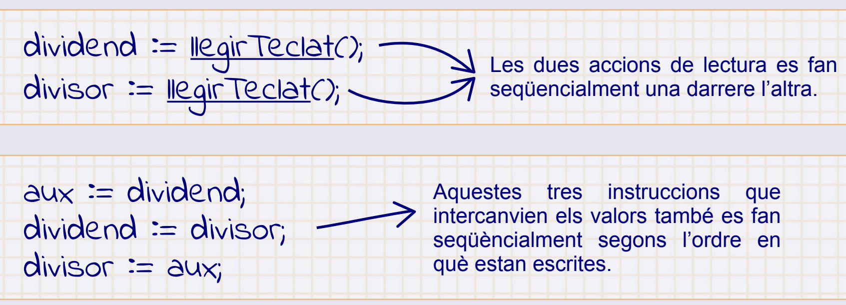
1. Estructura seqüencial

La primera de les opcions, la combinació seqüencial, ja l’hem utilitzada.

Un algorisme és, de fet, una seqüència d’accions (sentències o instruccions elementals). Això és, cada instrucció s’executa una darrere l’altra de forma ordenada. Els passos per resoldre el problema es van fent un darrere l’altre en l’ordre en què estan escrits en l’algorisme. Així que, tot i que pot semblar una estructura molt simple, és primordial en programació.

Exemple

Veiem en aquesta part de l’algorisme, que ja tenim resolta, com s’aplica l’estructura seqüencial.



Per millorar la llegibilitat de l’algorisme, fixeu-vos que **cada acció es posa en una línia diferent** i **s’acaba amb punt i coma**. Són convencions que faciliten l’elaboració i la comprensió de l’algorisme, i que és important respectar. El mateix passa en els llenguatges de programació, com podeu veure en els codis de JavaScript del final de la unitat.

Exemple

Dissenyem un algorisme que, donat el salari brut d'un treballador, en calculi el salari net i els impostos que ha de pagar sabent que paga un 20% d'impostos.

1 Entenem el problema

Tenim dues dades d'entrada: el salari brut i el percentatge que paga d'impostos. I ens demanen dues dades de sortida: per una banda el salari net i per una altra els impostos a pagar.

2 Pensem com resoldre'l

Decidim els càlculs a fer per obtenir el que demana l'enunciat

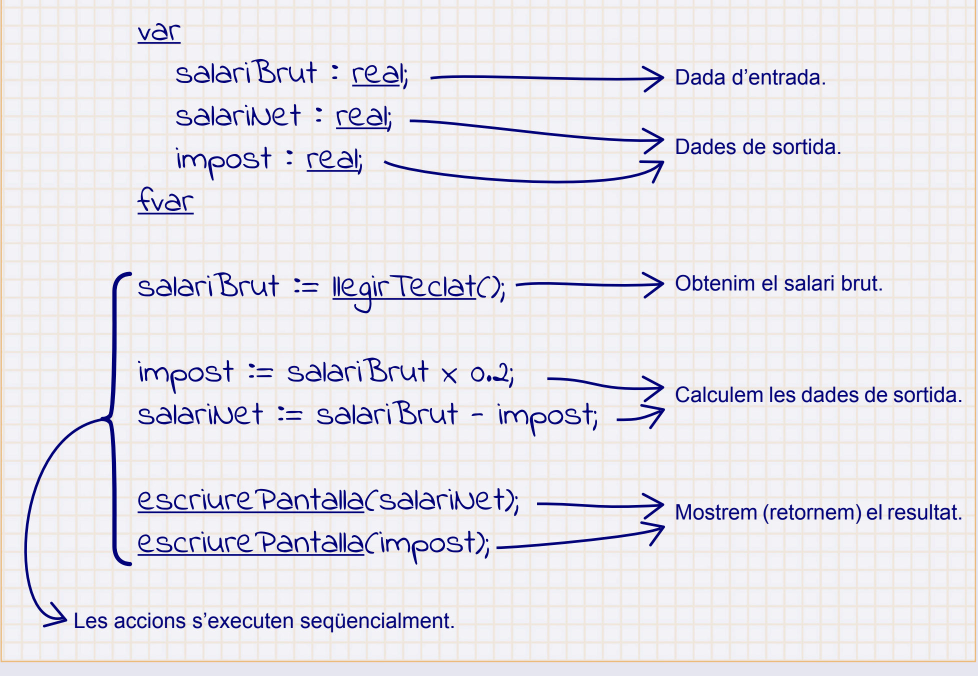
- Salari net = salari brut x 0.8
- Impostos = salari brut x 0.2

Una altra alternativa (estratègia de solució) seria

- Impostos = salari brut x 0.2
- Salari net = salari brut - impostos

3 Fem l'algorisme

Escollim, per exemple la segona opció.



2. Estructura alternativa

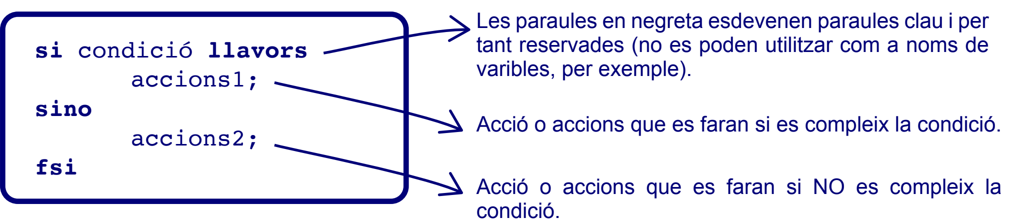
L'estructura seqüencial ens permet definir (seguir) un camí (un flux d'execució) per a resoldre el problema, en el qual una instrucció s'executa darrere l'altra, un camí únic, podríem dir. A vegades, però, necessitem que un algorisme faci determinades accions només si es compleix una certa condició, o que faci unes accions si es compleix una condició i, si no és compleix, que en faci unes altres. Per això s'**utilitza l'estructura alternativa o condicional**.

Aquest concepte d'accions condicionades s'il·lustra molt bé amb l'algorisme que seguim per creuar un pas amb semàfor: si està en verd, creuem, si està en vermell, esperem. I, així, podríem pensar en molts altres exemples.

Farem servir la composició alternativa quan vulguem executar una acció o una altra depenent de si es compleix una determinada condició o no (per això és diu també composició condicional), és a dir, quan vulguem variar el flux d'execució del nostre algorisme per definir diversos camins alternatius.

Sintaxi de l'estructura alternativa o condicional

Algorísmicament, l'**estructura alternativa** (escollir entre FER UNA COSA o FER-NE UNA ALTRA) es representa d'aquesta manera:



I la seva variant simple (escollir entre FER o NO FER):

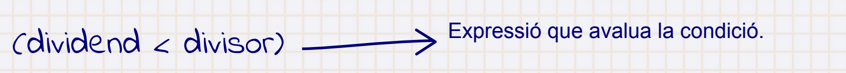
```
si condició llavors
      accions;
fsi
```

Exemple

Ara que coneixem la sintaxi de l'estructura condicional podem seguir avançant en el nostre algorisme d'Euclides per calcular el mcd entre dos números.

El primer pas del càlcul diu que cal comprovar si el dividend és més petit que el divisor i si és així intercanviar-los abans de continuar amb la resta d'operacions.

Ja tenim l'expressió per fer la comprovació:



I les accions per fer l'intercanvi en cas que calgués:

```
aux := dividend;
dividend := divisor;
divisor := aux;
```

→ Accions a fer si es compleix la condició.

Ara només ho hem de composar tot en una estructura iterativa:

```
si (dividend < divisor) llavors
    aux := dividend;
    dividend := divisor;
    divisor := aux;
fsi
```

→ Expressió que avalua la **condició**.

→ Accions a fer si es compleix la condició.

Ja teníem la composició alternativa feta! Fixeu-vos que és una versió simplificada de l'estructura perquè no ens cal, en aquest cas, la part alternativa de ja que, si el dividend ja és més gran que el divisor, no cal fer res. Per això prescindim d'aquesta part de l'estructura.

L'estructura condicional representa una part de l'algorisme (de codi, diríem, si parléssim d'un llenguatge de programació) que necessita comprovar si alguna condició és cert o falsa abans d'executar les accions d'aquesta part. Ens permet seleccionar quina acció cal fer.

Les decisions en els algorismes es prenen fent comparacions entre variables, números, caràcters o utilitzant expressions booleanes, en definitiva expressions que donen com a resultat cert o fals.

Recordem els operadors de comparació que hem vist anteriorment: **=** (igual), **≠** (diferent), **<** (menor), **>** (major), **≤** (menor o igual) y **≥** (major o igual), operadors que s'apliquen majoritàriament als nombres però que també es poden aplicar a caràcters i booleans, com ja hem vist anteriorment. Són operadors que sempre donen com a resultat un booleà: **cert** o **fals**.

Les **expressions booleanes** s'utilitzen per determinar quin flux d'execució (quina "ruta") ha de seguir l'algorisme depenent de si el resultat de l'expressió és cert o fals. D'aquí que tant les expressions algebraiques com el tipus booleà siguin tan importants en l'algorísmica.

Exemple

Han començat les rebaixes a la nostra botiga de roba i cal ajustar el sistema de càlcul del preu final de cada peça. Una feinada! Sort que podem fer un algorisme senzill que ens simplificarà la feina. La nostra promoció és la següent: a tots els articles els apliquem un 25% de descompte, però si es compren més de tres peces, llavors s'aplica un 50% a totes elles. Una ganga!

1 Entenem el problema

Tenim dues dades d'entrada: el preu de la compra i el nombre d'articles totals de la compra. I, com a sortida, ens demanen que calculem l'import després d'aplicar les rebaixes.

2 Pensem com resoldre'l

Decidim els càlculs a fer per obtenir el que demana l'enunciat:

- El tipus de descompte és 25% si el nombre de productes és d'1 a 3 i del 50% si és de més de 3.
- Import descomptat = import x tipus descompte.
- Import final = import - import descomptat.

3 Fem l'algorisme

Detallem els passos a seguir en notació algorísmica:

Què passaria si...

...no haguéssim definit els dos tipus de descompte com a constants? El càlcul del descompte hauria hagut de ser:

```
si numPeces ≤ 3 llavors
    importDescomptat := importInicial x (25.0/100.0);
sino
    importDescomptat := importInicial x (50.0/100.0);
fsi
```

Estructura alternativa que aplica un descompte o un altre segons el nombre de peces venudes.

Aquesta opció és correcta, però no facilita la modificació posterior de l'algorisme. Perquè, i si després d'uns dies decidim millorar l'oferta, i que els descomptes siguin del 30% i del 70% respectivament? Si tenim definits els descomptes com a constants, només cal que canviem el valor de les constants, sense revisar cap més línea de codi. Altrament, caldrà que revisem amb cura tot l'algorisme. Això pot semblar poc important en aquest cas, perquè l'algorisme té molt poques instruccions, però els programes normalment són de milers i milers de línies. Així que aquesta mesura es converteix en una bona pràctica quasi imprescindible si volem treballar de manera eficient.

Per tant, un algorisme més ben fet encara seria:

```
const
    tipusDescompte1 : enter := 25;
    tipusDescompte2 : enter := 50;
    límitPeces : enter := 3;
fconst

var
importInicial, importDescomptat, importFinal : real;
numPeces : enter;
fvar

importInicial := llegirTeclat();
numPeces := llegirTeclat();

si numPeces ≤ límitPeces llavors
    importDescomptat := importInicial x (enterAReal(tipusDescompte1)/100.0);
sino
    importDescomptat := importInicial x (enterAReal(tipusDescompte2)/100.0);
fsi

importFinal := importInicial - importDescomptat;
escriurePantalla(importFinal);
```

Així, si decidim que ara aplicarem el descompte a partir de 2 peces, canviar-lo serà molt més senzill.

...si calgués definir un sistema més granulat de rebaixa com ara, “entre una i dues peces aplicar un 10%, entre 3 i 5 peces aplicar un 20% i per a més de 6 peces aplicar un 30%”, seria:

```
si numPeces ≤ 2 llavors
    importDescomptat := importInicial x (10/100);
sino
        si (numPeces ≤ 5) llavors
            importDescomptat := importInicial x (20/100);
        sino
            importDescomptat := importInicial x (30/100);
        fsi
fsi
```

Com ha calgut fer en aquest exemple, les estructures es poden incloure una dins l’altra tantes vegades com calgui.

En aquest cas, per facilitar la llegibilitat de l’algorisme és molt important identar correctament les estructures, altrament costaria d’entendre què fan. Això pot semblar poc rellevant però és molt important. Per això els editors dels llenguatges de programació respecten sempre els colors que indiquen les paraules clau i la identació de les estructures.

O encara millor, si fem servir servir constants pels diferents tipus de descomptes i així els podrem modificar fàcilment si cal més endavant variar les condicions de les rebaixes:

```
const
    tipusDescompte1 : enter := 10;
    tipusDescompte2 : enteo := 20;
    tipusDescompte3 : enter := 30;
    limitPeces1 : enter := 2;
    limitPeces2 : enter := 5;
fconst

var
importInicial, importDescomptat, importFinal : real;
numPeces : enter;
fvar

importInicial := llegirTeclat();
numPeces := llegirTeclat();

si numPeces ≤ limitPeces1 llavors
    importDescomptat := importInicial x (enterAReal(tipusDescompte1)/100.0);
sino

si numPeces ≤ limitPeces2 llavors
    importDescomptat := importInicial x (enterAReal(tipusDescompte2)/100.0);
sino
    importDescomptat := importInicial x (enterAReal(tipusDescompte3)/100.0);
fsi
fsi

importFinal := importInicial - importDescomptat;
escriurePantalla(importFinal);
```

Estructures alternatives més elaborades

Hem vist dues variacions de l'estructura alternativa:

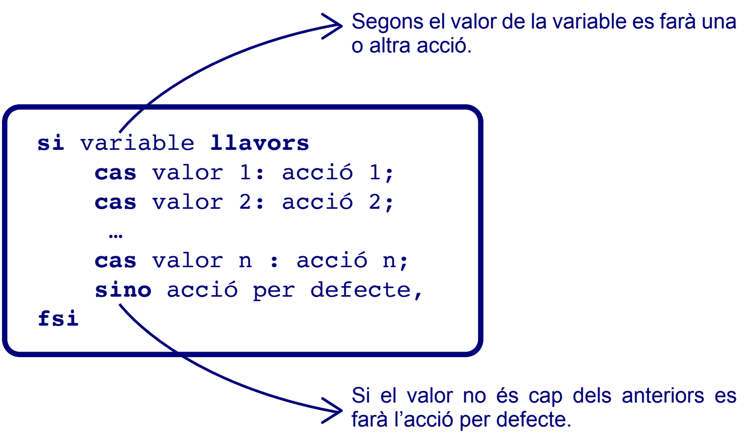
- Fer o no fer (la versió simplificada que ens permet fer una desviació en el camí): **Si.**
- Fer una cosa o fer-ne una altra (la versió completa que ens permet dos camins alternatius) **SI-SINO.**

També, que l' estructura alternativa dona joc a moltes combinacions de l'execució del flux d'un programa quan fer una acció depèn d'una determinada condició, com ja hem vist. A vegades caldrà, fins i tot, utilitzar una (o diverses) estructures alternatives una dins l'altra. És el que es coneix com a estructures **imbricades** o **niuades** que ens permeten definir diversos camins alternatius.

Existeix, però, una altra variant que serveix només en casos molt concrets en els quals es pot estalviar haver d'imbricar diverses estructures alternatives, com seria el cas de l'exemple anterior. Es tracta d'escollir entre diferents alternatives possibles depenent del valor d'una variable de tipus enter o caràcter:

- Fer una d'aquestes accions segons el cas (selecció entre diferentes opcions): **SI-CAS-SINO.**

La sintaxi d'aquesta estructura és la següent:



Exemple

Volem fer un algorisme que simuli una calculadora que fa les operacions aritmètiques elementals (suma, resta, multiplicació i divisió) a partir de dos números donats a i b enters positius.

```
var
    a, b : enter
    r : enter;
    operació: caràcter;
fvar

a := llegirTeclat();
b := llegirTeclat();
operació := llegirTeclat();

si (operació) llavors
    cas 'S': r = a + b;
    cas 'R': r = a - b;
    cas 'M': r = a x b;
    cas 'D': r = a div b;
    sino
fsi
```

Segons el valor de la variable es farà una o altra acció.

Si l'operació que ens demanen no és cap d'aquestes, no cal fer res.

Aquesta estructura "busca" quina acció cal fer entre les diferents opcions (casos) que es descriuen. L'opció **sino** indica què fer si el valor de la variable no coincideix amb cap dels casos detallats.

Quan s'utilitza aquesta estructura, el flux del programa deixa de comprovar més opcions de seguida que troba una resposta positiva. És important doncs relacionar els casos en l'ordre més adient perquè l'algorisme sigui el més eficient possible.

3. Estructura iterativa

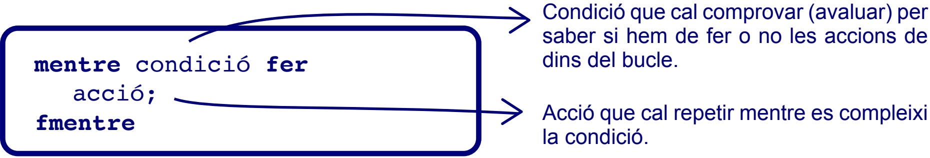
Amb l'estructura condicional ja es pot resoldre un nombre més gran de problemes, però encara ens cal anar més enllà. Sovint, cal que un algorisme repeteixi una acció (o un conjunt d'accions) unes quantes vegades fins que es compleixi una determinada condició. Per a això, s'utilitza l'**estructura iterativa o repetitiva**.

Aquest concepte d'accions repetides es representa amb la idea de **bucle**. I s'il·lustra molt bé amb una pila d'algorismes que seguim de manera quotidiana: beure aigua mentre tinguem sed, fer la PAC fins que l'hàgim acabada, pujar la muntanya fins que arribem al cim, repetir la sèrie d'abdominals tres cops cada dia, córrer fins arribar a la meta, baixar esglaons fins que arribem a peu d'escala; i així, molts d'altres.

Farem servir la composició iterativa quan vulguem repetir una acció que ja s'ha fet (és a dir, recular a una acció anterior) i tornar-la a fer, per això és diu també composició repetitiva. O, dit d'una altra manera, quan vulguem variar el flux d'execució del nostre algorisme per poder tirar enrere i executar diversos cops (**iteracions**) una part del codi.

Sintaxi de l'estructura iterativa o repetitiva

Algorísmicament, l'**estructura iterativa** (repetir diverses vegades una part del codi) es representa d'aquesta manera:

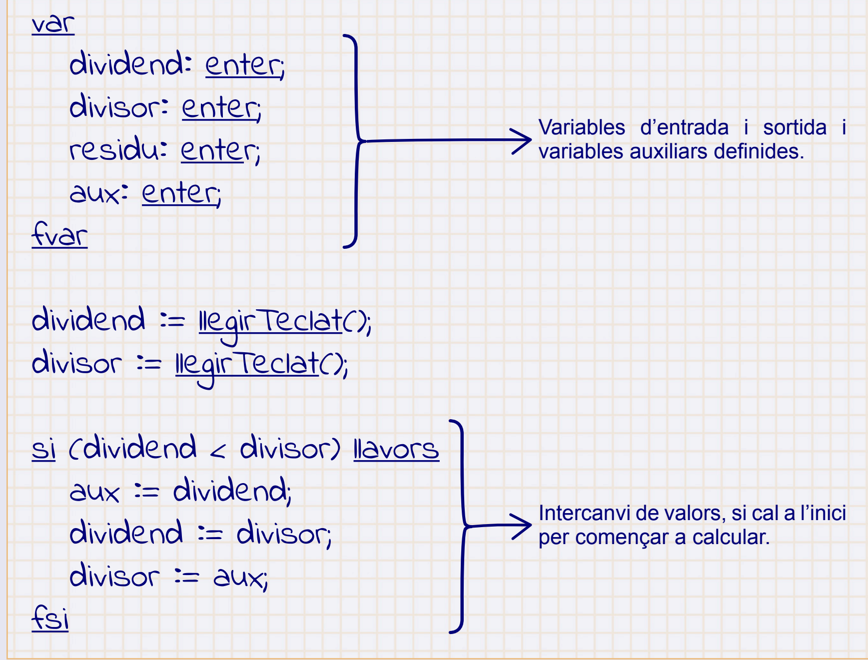


Cada "volta" que fem en el bucle l'anomenem **iteració**. L'estructura iterativa permet repetir l'execució del bucle mentre es compleix la condició lògica. Aquesta condició es verifica al principi de cada iteració. Si la primera vegada ja no es compleix la condició, no s'executa cap iteració i es manté el flux de l'algorisme per a la següent acció després del bucle en el cas de l'exemple, l'acció **escriurePantalla**. Un cop es deixa de complir la condició "se surt" del bucle i es continua el flux de l'algorisme per l'acció que hi ha després del bucle (a continuació del **fmentre**).

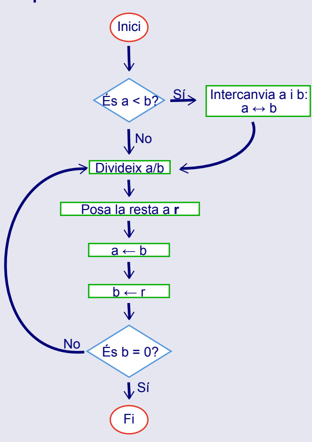
Exemple

Ara que coneixem la sintaxi de l'estructura iterativa podem, finalment, acabar de dissenyar l' algorisme d'Euclides per calcular el mcd entre dos números.

Ens havíem quedat en aquest punt:



Ens queda fer el càlcul per obtenir el mcd. Recordem com es fa:



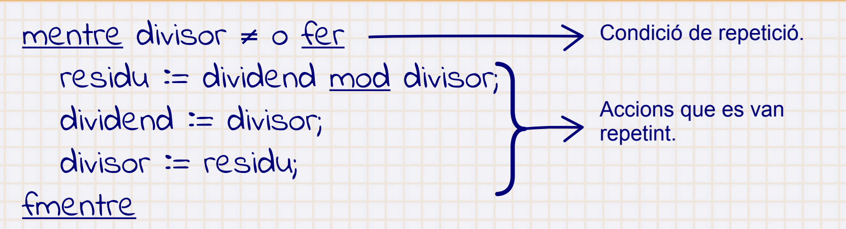
Fem els càlculs que caldrà repetir:

```
residu := dividend mod divisor;
dividend := divisor;
divisor := residu;
```

I l'expressió que regularà el número de cops a repetir:

```
divisor ≠ 0
```

Així que ja podem completar el bucle:

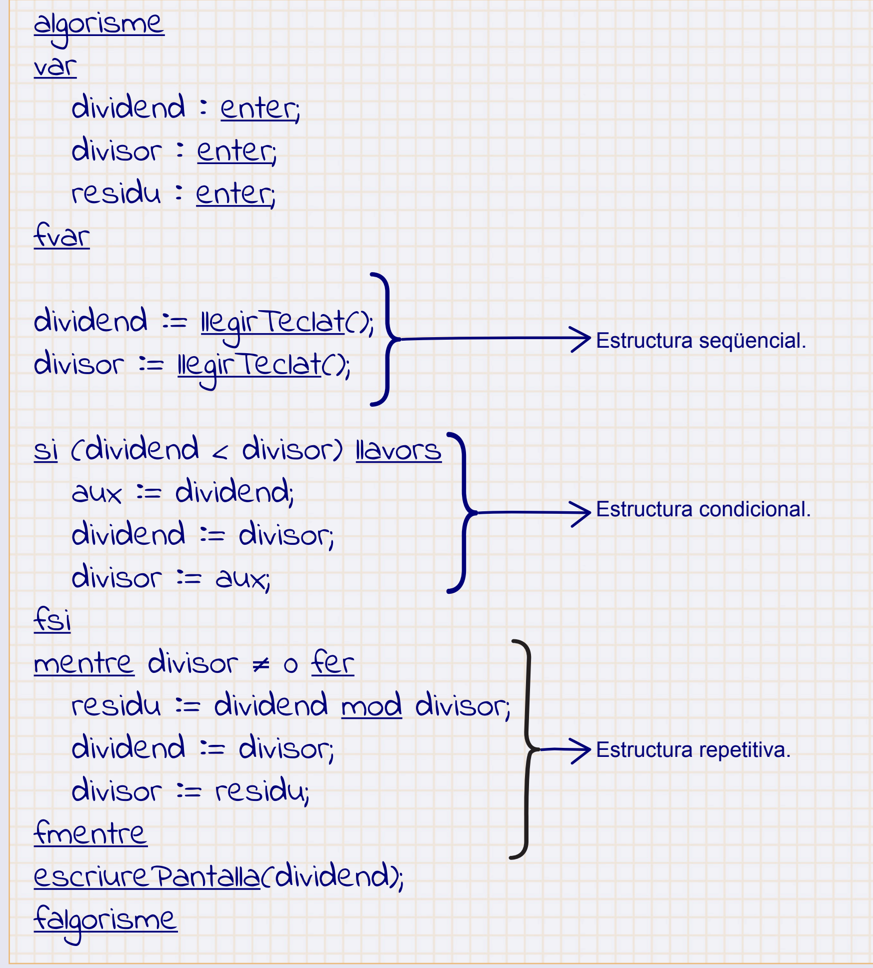


Fixeu-vos que el nombre d'iteracions que farà el bucle depenen del valor dels dos números dels quals hem de calcular el mcd.

Quan el divisor sigui 0 ja no cal seguir dividint, el mcd haurà quedat a la variable *dividend*. Aquest serà el resultat que mostrarem:

```
escriurePantalla(dividend);
```

I, ara sí, tindrem l'algorisme complet per calcular el mcd entre dos nombres enters positius:



L'estructura iterativa representa una part de l'algorisme (de codi, diríem, si estem parlant d'un llenguatge de programació) que cal anar repetint mentre una condició sigui certa o bé un nombre determinat de vegades.

Amb l'estructura iterativa **mentre** podem representar qualsevol repetició que ens calgui en un algorisme, és l'estructura iterativa per antonomàsia.

El format s'assembla al de l'estructura condicional simple. I, igualment, també necessita avaluar primerament una condició lògica, això és una expressió que necessàriament ha de donar com a resultat un valor booleà. Si el valor és cert, s'executen les accions de dins del bucle, si no, es continua a partir de la següent acció després del **fmentre**. De nou, es constata la importància de les expressions algebraiques en les estructures algorísmiques.

Exemple

Un altre "joc" matemàtic és comptar les xifres d'un nombre enter. Així sabrem, per exemple, quants caràcters (quant d'espai) caldran si el volem escriure en un formulari:

1 Entenem el problema

La dada d'entrada és el número enter a escriure. La sortida és el nombre de xifres (caràcters) que ocuparà quan l'escrivim per pantalla.

2 Pensem com resoldre'l

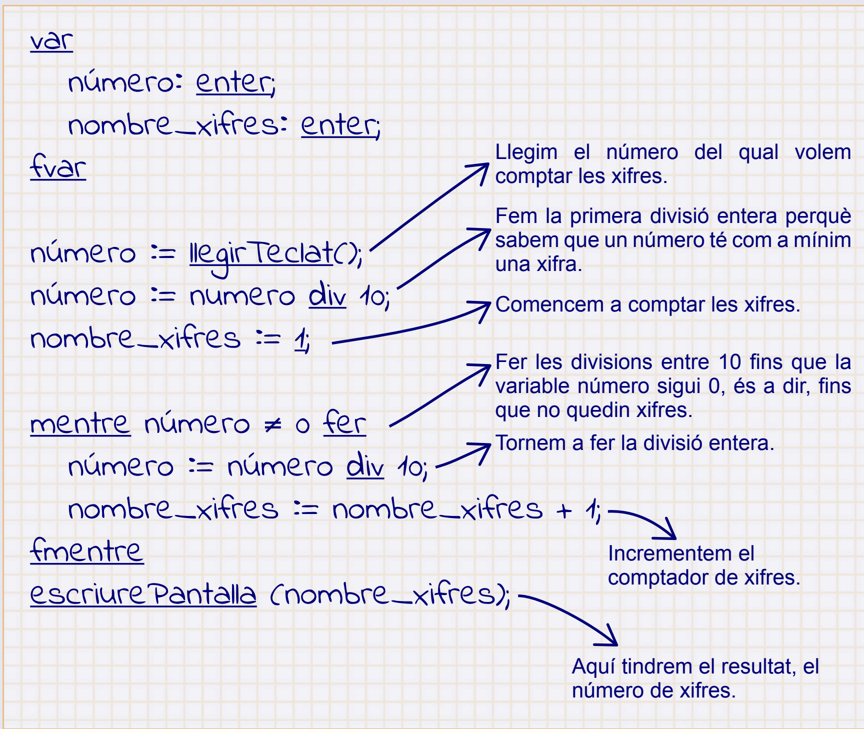
Decidim els càlculs a fer per obtenir el que demana l'enunciat.

La divisió entera d'un número enter qualsevol entre 10 dona el mateix nombre sense la xifra de les unitats (la de més a la dreta). Per exemple: 3943 div 10 = 394.

Aquest fet ens permet obtenir la solució repetint les divisions fins que el resultat sigui 0. En efecte, arribarà un moment en què només tindrem la divisió 3 div 10 = 0.

3 Fem l'algorisme

Com que hem de repetir la divisió entera diverses vegades fins que el número sigui zero, farem servir la composició iterativa **mentre**.



Simulant l'algorisme pel número 3984, es pot comprovar que fem en total 10 accions, ja que les dues accions del bucle es repetiran 3 cops cadascuna.

En aquest exemple s'observa alguns aspectes interessants de l'estructura iterativa. I el fet és que, en molts casos, cal una variable que fa de **comptador** (en el nostre exemple és la variable *nombre_xifres*). Si és el cas, convé tenir presents aquests aspectes:

- El **nom de la variable comptador**, procurar que sigui adient i clar.
- Cal **inicialitzar el comptador**, normalment abans de començar l'estructura iterativa.
- Cal **anar incrementant el comptador** dins l'estructura iterativa.

Què passaria si...

…la condició del bucle mai no fos certa?
En aquest cas, les accions de dins del bucle no es farien mai, i això no té cap mena de sentit. Seria com dibuixar un camí pel qual no hi ha cap manera de passar-hi.
Per tant, doncs, és importantíssim assegurar que almenys la condició del bucle es cumplirà un cop!

I si la condició del bucle mai no fos falsa?
En aquest cas, tindríem un bucle infinit i l'algorisme no acabaria mai. Així que, compte! Quan construïm composicions iteratives és imprescindible comprovar que el bucle acabarà el algun moment. Assegurar la **condició de final de la iteració** és un aspecte que porta molts maldecaps als programadors pels efectes tan nefastos que té si no es gestiona bé, i que poden provocar que el programa "es pengi".

Una altra estructura iterativa

L'estructura **mentre** permet organitzar pràcticament qualsevol conjunt d'instruccions que calgui repetir, però no sempre és l'estructura més adient.

- Repetir unes accions mentre es compleixi una certa condició (bucle controlat per una condició): **MENTRE.**

En alguns casos, és més òptim fer servir una altra estructura iterativa anomenada **PER**:

- Repetir unes accions un nombre determinat de cops (bucle controlat per un comptador): **PER.**

Exemple

El professor de matemàtiques està repartint els exàmens del segon semestre que ja ha corregit als estudiants de la classe. Són 25, per tant, sap del cert que haurà de repartir 25 exàmens.

Es podria resoldre amb una estructura **mentre**:

```
i := 1
mentre i ≤ 25 fer
   llegir el nom de la posició i de la llista;
   cridar l'estudiant amb aquell nom;
   donar-li el seu examen;
   i := i+1;
fmentre
```

Però atès que en aquest cas sabem el nombre d'iteracions que caldrà fer, és més simple utilitzar l'estructura **per** ja que no cal inicialitzar ni anar incrementant el comptador.

```
per i := 1 fins 25 fer
   llegir el nom de la posició i de la llista;
   cridar l'estudiant amb aquell nom;
   donar-li el seu examen;
fper
```

El bucle es va repetint des de i=1 fins a i=25, el comptador i, s'incrementa automàticament.

S'utilitza la construcció **per** quan es coneix el nombre de vegades que s'ha de repetir el codi. És una estructura molt útil per treballar amb seqüències de dades, com es veurà més endavant.

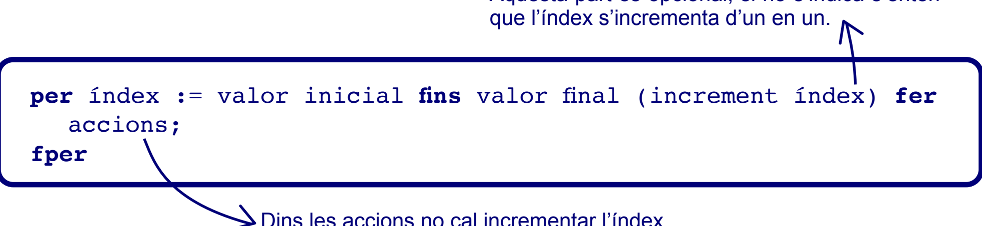
Una construcció **per**, sempre es pot fer utilitzant una composició **mentre**, però a l'inrevés no passa. Per això diem que la construcció **mentre** és més general que la construcció **per**.

La forma que hem vist de la composició iterativa **per** (que en cada volta incrementa el comptador en una unitat) és la més habitual. Però no respon a totes les situacions amb els quals ens podem trobar. Per exemple, si volem simular un compte enrere des d'un valor determinat fins a 0 o fer la suma de tots els nombres parells entre dos nombres inicials.

És clar que ambdós problemes els podríem resoldre amb una estructura **mentre** perquè per poder-ho fer amb una estructura **per**, malgrat saber la quantitat de voltes que cal realitzar, cal poder indicar a la composició que, si fem un compte enrere, ens interessaria anar baixant el valor del comptador i no pujar-lo i, si volem sumar parells, aniria bé poder incrementar el comptador de dues en dues unitats.

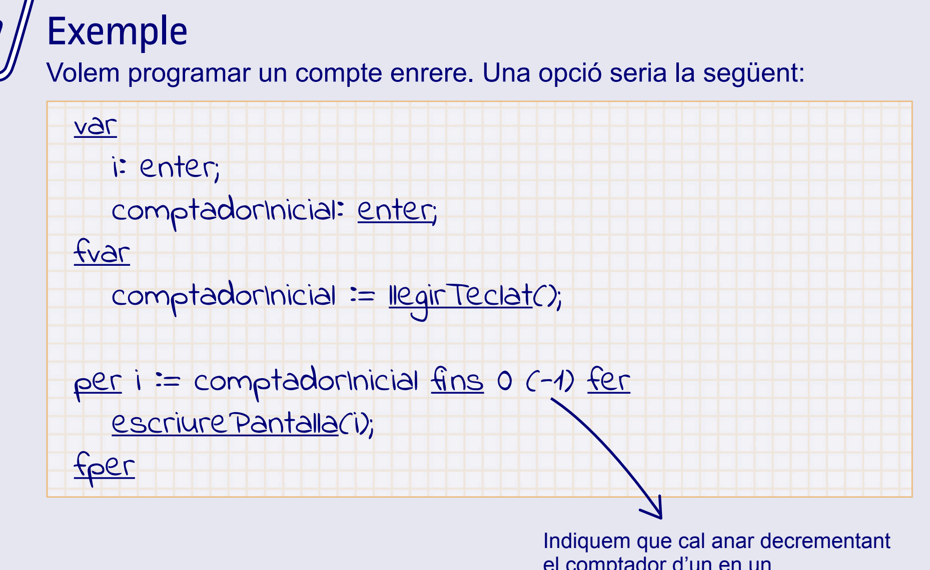
La composició **per** permet també indicar com volem que varii el comptador a cada iteració.

La sintaxi de l'estructura completa **PER** és la següent:



Exemple

Volem programar un compte enrere. Una opció seria la següent:

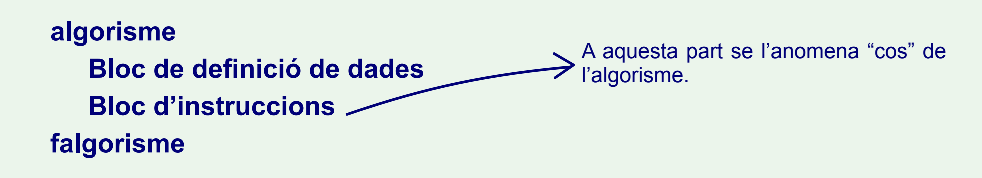


Per últim, les estructures iteratives (tal com passava amb les condicionals) es poden combinar amb d'altres estructures iteratives i condicionals, utilitzant una (o diverses) estructures una dins l'altra. És el que es coneix com a estructures **imbricades** o **niuades** que ens permeten definir moltíssimes combinacions de fluxos d'execució.

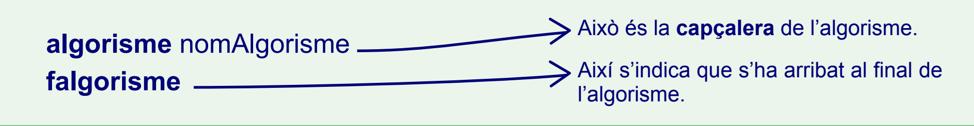
Estructura d'un algorisme

Un algorisme (i un programa també) segueix una estructura predeterminada per anar escrivint tots els elements d'una manera ordenada, assignant un lloc concret a cadascun d'ells. Per tant, tots segueixen la mateixa estructura i això els fa més fàcilment comprensibles. Els elements que inclou un algorisme ja sabem que són de dos tipus, bàsicament: relacionats amb les dades i relacionats amb les instruccions.

En primer lloc, lògicament, sempre es descriu la part relacionada amb les dades (declaració de variables i constants). D'aquesta manera, l'algorisme sap quines dades podrà utilitzar a les instruccions, que es detallen sempre posteriorment i que formen el **cos** de l'algorisme:

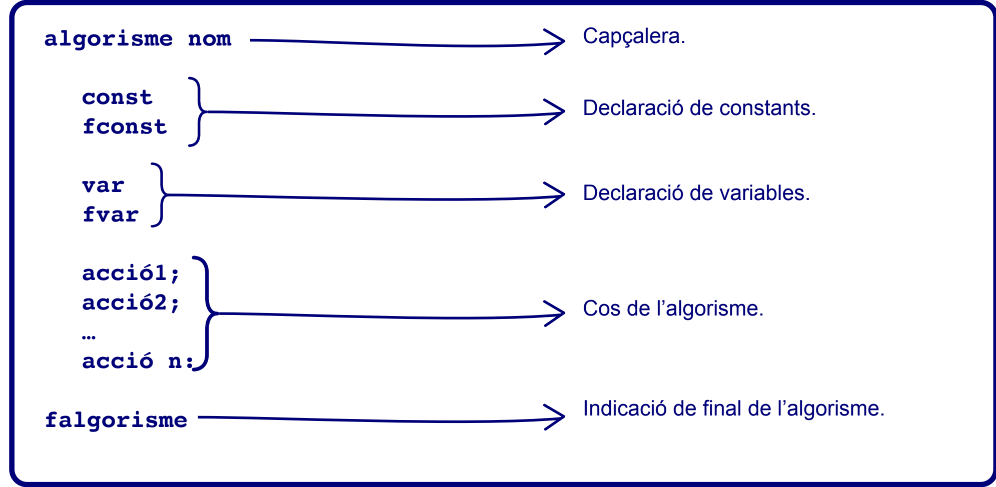


A més, per indicar l'inici i el final d'un bloc, la notació algorísmica ho denota amb determinades paraules clau. En el cas de l'inici i final d'un algorisme ho denota així:



És important que el nom que donem a l'algorisme sigui significatiu de què fa. Així, es facilita la seva comprensió i posterior modificació si cal. Es una bona pràctica recomanable aplicar les mateixes recomanacions de la nomenclatura de les variables.

Sintaxi de l'estructura d'un algorisme



Conceptes clau

Un algorisme és una seqüència d'accions que se succeeixen per resoldre un problema a partir d'unes dades inicials. Aquestes accions s'organitzen de manera **seqüencial** (una després de l'altra), **condicional o alternativa** (unes o unes altres depenent d'una determinada condició), o de manera **iterativa** (es van repetint un nombre concret de vegades o mentre es produeixi una determinada circumstància). Així doncs, tenim tres estructures fonamentals amb les quals podem construir el flux d'execució del programa.

La programació estructurada és un paradigma de programació que es basa en definir un algorisme com un nombre de pasos finit estructurat únicament usant aquestes tres estructures, d'aquí el seu nom.

L'estructura condicional presenta tres formes bàsiques:

- Fer o no fer (la versió simplificada que ens permet fer una desviació en el camí): **Si.**
- Fer una cosa o fer-ne una altra (la versió completa que ens permet dos camins alternatius) **SI-SINO.**
- Fer una d'aquestes accions segons el cas (selecció entre diferentes opcions): **SI-CAS-SINO.**

L'estructura iterativa té una forma per excel.lència que cobreix pràcticament tots els casos:

- Repetir unes accions mentre es compleixi una determinada condició (bucle controlat per una condició): **MENTRE.**

Però, quan coneixem del cert el nombre d'iteracions que caldrà fer, és més pràctic utilitzar aquesta altra variant:

- Repetir unes accions un nombre determinat de vegades (bucle controlat per un comptador): **PER.**

En ambdues estructures (condicional i iterativa) les **expressions** i els valors booleans juguen un paper imprescindible, ja que les usem per definir les condicions d'execució d'una branca o una altra en el cas de l'estructura condicional i les repeticions a fer en el cas de la estructura iterativa.

En el cas de l'estructura iterativa és molt important assegurar que el bucle s'executa al menys una vegada i també que acaba d'alguna manera (**condició de final** de la iteració), altrament entraríem en un bucle infinit i el programa es penjaria. Això s'aconsegueix tenint cura de com es defineix la condició que controla l'execució del bucle.

Amb aquestes estructures podem indicar múltiples maneres d'execució del flux d'un programa perquè, a més, podem incloure unes dins les altres segons convingui (un **mentre** a dins d'un **si**, un **mentre** a dins d'un altre **mentre**, etc.). És el que es coneix com a estructures **imbricades** o **niuades**.

Per facilitar la comprensió dels algorismes, s'identifiquen amb dues paraules clau que marquen l'inici i el final (**algorisme** i **falgorisme**). Així mateix, un algorisme segueix sempre la mateixa **estructura** de blocs; es comença pel bloc de dades: primer es declaren les constants i després les variables. D'aquesta manera, l'algorisme ja "sap" amb quins elements pot operar (les variables i les constants). I, després, s'inclou el cos de l'algorisme amb totes les accions que ha de fer combinades convenientment a partir de les tres estructures. Els llenguatges de programació també segueixen unes pautes de format molt similars. Pautes que, en el cas dels programes, cal seguir estrictament perquè després es puguin executar.

Les accions poden ser assignacions de valors a variables (l'**acció elemental**), operacions amb les dades (operacions aritmètiques en el cas dels nombres, per exemple) o crides a funcions ja existents, com per exemple l'escriptura per pantalla.

Com treballem les estructures en JavaScript?

Un cop s'ha decidit quina serà l'estructura de l'algorisme que resol un problema determinat, sabem ja quines estructures utilitzarem per indicar el flux de l'execució. Caldrà, després, concretar-ho en un llenguatge de programació.

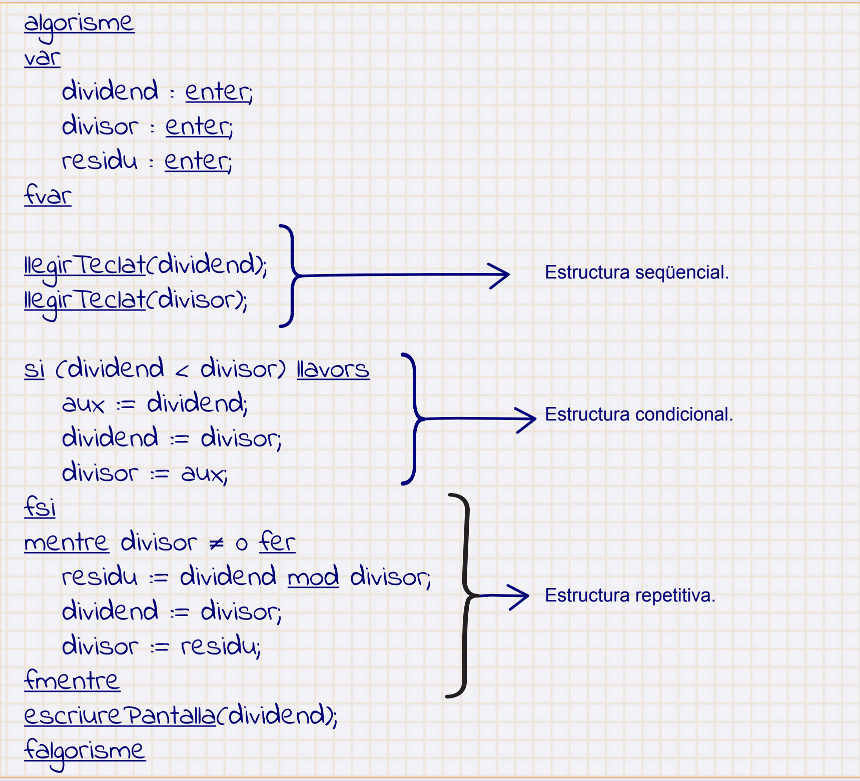
Exemple

Seguim amb el desenvolupament del programa per calcular el mcd. A la pàgina 14 tenim l'algorisme ja dissenyat com a resultat dels tres primers passos:

1 Entendre el problema

2 Pensar com resoldre'l

3 Dissenyar un algorisme que detalli els passos a fer



4 Implementar-ho en llenguatge JavaScript

El llenguatge JavaScript permet representar totes les estructures algorísmiques que hem vist, cadascuna de les quals té la seva pròpia sintaxi que és important conèixer i aplicar correctament en el moment de programar. Tot i així, els principis implicats algorísmicament són plenament vàlids i convé no oblidar-los en cap moment.

El codi JS que seguiria l'estratègia indicada a l'algorisme anterior és el següent:

```
1  // definim le varibles d'entrada i els hi assignem
2  // un valor per provar
3  var dividend = 126;
4  var divisor = 72;
5  var aux, residu;
6  // si el dividend és menor que el divisor,
7  // els intercanviem
8  if (dividend < divisor){
9    aux = dividend;
10   dividend = divisor;
11   divisor = aux;
12   }
13 //calculem el mcd
14 while (divisor != 0) {
15   residu = dividend % divisor;
16   dividend = divisor;
17   divisor = residu;
18   }
19 // mostrem el resultat
20 console.log(dividend);
21
```

L'estructura d'un programa JS també és bo que segueixi aquest ordre: primer declarar les constants i les variables i, a continuació, afegir les instruccions. No hi ha, en aquest cas, cap delimitador d'inici i fi.

Però el llenguatge JS és més flexible i permet definir i donar valor tant a constants com a variables a la mateixa línia de codi. En qualsevol cas, lògicament, és imprescindible que les variables i constants es defineixin abans de ser utilitzades. Altrament, el programa donarà un error d'execució.

Sintaxi de les estructures alternatives en JS

La sintaxi és molt similar a la que fem servir en notació algorísmica.

Fer o no fer (condicional simple):

```
if  (condició) {
    acció;
};
```

Fer una cosa o una altra (condicional):

```
if  (condició) {
    acció1;
 }
else
{
    acció2;
};
```

Fem servir les claus per delimitar les accions a fer.

Fer una cosa o una altra (condicionals imbricats):

```
if  (condició1) {
    acció1;
 }
else if (condició2)
{
    acció2;
};
```

Escollir entre vàries opcions:

```
switch (condició) {
    case valor :
           acció1;
           break;
    case valor :
           acció2;
           break;
    default:
           accióDefault;
           break;
}
```

Exemple

```
1  const tipusDescompte1 = 25;
2  const tipusDescompte2 = 50;
3  const límitPeces = 3;
4  //variables d'entrada, els hi assignem el valor
5  // directament
6  var importInicial = 1000;
7  var numPeces = 4;
8  //variables intermija i final
9  var importDescomptat, importFinal;
10 if (numPeces <= límitPeces){
11 importDescomptat = importInicial * (tipusDescompte1/100);
12 }
13 else {
14 importDescomptat = importInicial * (tipusDescompte2/100);
15 }
16 importFinal = importInicial - importDescomptat;
17 console.log(importFinal);
18
```

```
1  var a, b;
2  var resultat;
3  var operació;
4  // donem valor a les variables d'entrada
5  var a = 2;
6  var b = 5;
7  var operació = "S";
8  //simulem el funcionament de la calculadora
9  switch (operació){
10 case "S":
11     resultat = a + b;
12     break;
13 case "R":
14     resultat = a - b;
15     break;
16 case "M":
17     resultat = a * b;
18     break;
19 case "D":
20     resultat = a / b;
21     break;
22 default:
23     console.log("Operació no permesa");
24     break;
25 }
26 console.log(resultat);
```

Sintaxi de les estructures iteratives en JS

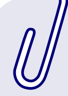
La sintaxi és també molt senzilla.

Trobareu una explicació detallada del funcionament de totes aquestes estructures en JavaScript a la guia de JavaScript de Mozilla.

Repetir mentre es compleixi una determinada condició

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

Aquí les claus també serveixen per delimitar les accions a fer.



Exemple

```
1  var numero; // valor d'entrada
2  var nombre_xifres; //valor resultant
3  numero = 3984; //donem valor a l'entrada
4  // dividim i ens quedem amb la part entera
5  numero = parseInt(numero/10);
6  //Inicialitzem el valor resultant
7  nombre_xifres= 1;
8  while (numero > 0) {
9    numero = parseInt(numero / 10);
10   nombre_xifres = nombre_xifres + 1;
11 }
12 console.log (nombre_xifres);
13
```

Repetir un nombre determinat de vegades

```
for (índex = valor inicial; índex < valor final; increment índex) {
        accions;
 }
```

Exemple

```
1  var i;// índex
2  var comptadorInicial;
3  // donem valor a la variable d'entrada
4  var comptadorInicial = 3;
5  //simulem el comptador enrere
6  for (i=comptadorInicial; i>0; i--){
7    console.log(i);
8  }
9  console.log(i);
10
```

Decrementa el comptador en una unitat.

Provar el codi en JS

El darrer pas que fem quan programem és provar el codi resultant per assegurar que fa el que realment ha de fer.

Per això, és important pensar en una sèrie de **jocs de prova**. Un joc de prova és un conjunt de valors, un per a cada variable inicial, que, executant el programa, dona el resultat final esperat i correcte. Es poden necessitar molts casos de prova per determinar si un programa és correcte o no.

Exemple

5 Provar el codi

Com podem provar si el codi del programa JS que calcula el mcd aplicant l'algorisme d'Euclides és correcte? Si reculem al pas 1 (entendre el problema) recordarem quines eren les variables d'entrada: dos números enters positius *a* i *b*. I com a sortida del programa esperem el resultat de calcular el seu mcd, un altre enter positiu.

Hi ha molts tipus de proves, les que es fan a una part de codi concreta i petita es diuen **proves unitàries**.

Podem fer una taula i pensar alguns valors per provar que funciona el codi. Cal pensar en valors "possibles" d'*a* i *b* segons el que detalla l'enunciat. En aquest cas diu que són enters positius, per tant no cal que provem casos com ara a=-10 o b=0 que ja sabem que no es produiran.

Hem de donar valors diferents a *a* i *b* i calcular quin és el resultat que esperem del codi.

Jocs de prova	a	b	Resultat
Cas 1	126	72	18
Cas 2	72	126	18
Cas 3	60	36	12
Cas 4	36	60	12
...			
Cas n			

El nombre de casos a provar dependrà de la complexitat del codi del programa i han de ser representatius dels casos més importants i extrems que es puguin donar durant l'execució del programa, per així poder verificar el comportament del programa en el màxim número de situacions possibles.

Pot semblar que els casos 1 i 2 són redundants, però no. En el cas 2, *a es menor que b* i per tant s'executarà la sentència alternativa i s'intercanviaran els valors d'*a* i *b*. En el primer cas, això no cal. Així ens assegurem de provar totes les branques (els fluxos d'execució possibles) del codi.

Quan alguna prova falla (no dona el resultat que s'espera) es pot executar el codi amb el PythonTutor, i anar executant línia a línia fins a localitzar on és l'error i corregir-lo.

Podeu veure un exemple en aquest video.

A vegades l'error pot ser sintàctic (hem escrit un "wile" enlloc de "while", o hem oblidat un claudàtor per tancar una sentència *if*), altres semàntic (hem usat el símbol **=** per comparar en comptes del **==**, per exemple).

Les proves, tot i que puguin semblar una qüestió menor, senzilla o ja final i més rutinària, són una part fonamental de la programació. Si una prova falla, cal recular als passos anteriors i veure on és el problema: està mal codificat?, l'algorisme està mal pensat i per això no fa el que ha de fer?, no s'ha entès el problema inicial? Com més calgui recular per solucionar l'error, més costós serà corregir-lo. Aquest és un altre dels motius pels quals és important fer tots els passos, un a un quan estem programant: saltar-se'n algun (codificar sense pensar l'estratègia de solució o sense entendre el problema, per exemple) sol donar mals resultats al final.

A mesura que els programes es van fent grans i tenen més i més línies de codi, les proves no podran descobrir tots els possibles errors del codi desenvolupat.

Què passaria si...

...feu les mateixes proves amb el següent codi?

```
1  // definim le varibles d'entrada i els hi assignem
2  // un valor per provar
3  var dividend = 72;
4  var divisor = 126;
5  var aux, residu;
6  //calculem el mcd
7  while (divisor != 0) {
8    residu = dividend % divisor;
9    dividend = divisor;
10   divisor = residu;
11   }
12 // mostrem el resultat
13 console.log(dividend);
14 
```

Que donaria els mateixos resultats! De fet, l'algorisme d'Euclides es pot simplificar, ja que si el dividend és més petit que el divisor, la primera iteració del bucle ja serveix per intercanviar-ne els valors. Però la primera versió del codi era més clara i entenedora, oi? Aquesta segona opció seria més eficient. A vegades cal triar entre claredat i eficiència.

Podeu provar aquests codis a la web de PythonTutor.