.

Ampliació d'Informàtica
Apunts de recursivitat

Maria Dolors Ayala Vallespí
Dept. Ciències de la computació
E.T.S.E.I.B.
Universitat Politècnica de Catalunya

Tardor 2023

Índex

1 Recursivitat

1.1 Introducció

La recursivitat és un mètode de disseny de programes on la solució d'un problema depèn de la solució del mateix problema aplicat a instàncies més simples de les dades. El problema no es redueix; el que es simplifica són les dades a les que s'aplica el problema de tal manera que s'arriba a un conjunt de dades per les que la solució del problema és trivial.

En el disseny de funcions recursives, la funció es crida a ella mateixa amb un conjunt de dades més simples (*cas recursiu*) fins que s'arriba al conjunt de dades per les que la solució del problema és trivial (*cas base*).

El concepte de recursivitat es troba en altres camps. Trobem definicions recursives on la definició d'un terme es fa usant el propi terme.

- un ésser humà és aquell que els seus progenitors són éssers humans
- un arbre (estructura de dades) és un node (arrel) i 0 o més arbres units a l'arrel per una aresta
- un directori (carpeta) és una estructura amb fitxers i altres directoris. Els fitxers són el cas base.

Les expressions matemàtiques basades en el raonament inductiu, defineixen un primer element (cas base) i la forma de calcular qualsevol altre element en funció d'un altre (cas recursiu), de forma que al final s'arribi al cas base.

Exemples:

- el zero és un nombre natural i cada nombre natural té un successor que també és un nombre natural. Axioma de G. Peano.
- La següent definició de la funció factorial:

  $0! = 1$
  $n! = (n-1)! * n$

  és una definició recurrent: defineix el factorial de n en funció del factorial de $n-1$ i el cas base és $0! = 1$. Observem que cap de les dues equacions constitueix una definició completa. La primera és el cas base i la segona, el cas recursiu.

La recursivitat permet plasmar directament aquestes definicions recurrents en funcions Python recursives. També es poden dissenyar funcions recursives allà on fins ara hem aplicat iteracions.

Una funció recursiva té un o més casos base, és a dir, casos en què la funció pot retornar un resultat; i un o més casos recursius, és a dir, casos en què la funció simplifica les dades i es crida a ella mateixa. Per tant, una funció recursiva tindrà sempre una composició condicional. El cas base acaba amb la seqüència de crides recursives.

2 Metodologia

Abans d'escriure el codi d'una funció recursiva cal:

- definir el cas (o casos) que té una solució trivial: cas (o casos) base
- definir la relació de recurrència: l'equació que defineix la solució del problema a partir de l'aplicació del mateix problema a dades més simples

A continuació, cal aplicar el patró següent:

```
def f_recursiva (paràmetres):
    if cas_base:
        solució cas trivial
    else:
        solució a partir del resultat d'una o més crides a f_recursiva
```

La recursivitat és una alternativa a la iteració. La iteració amb la sentència `for` és segura ja que aquesta sentència controla els elements bàsics de la iteració: inici, passar a la següent iteració i la condició d'acabament. La iteració amb la sentència `while` és més complexa ja que és la persona que fa el disseny la responsable de definir bé aquests tres elements. Doncs bé, en la recursivitat també cal definir bé aquests tres elements: el pas al següent està relacionat amb les crides recursives i l'inici o la condició d'acabament amb el o els casos base.

Exemple 1: factorial

Recordem la definició de la funció factorial:

$0! = 1$
$n! = (n-1)! * n$

Dissenya la funció recursiva `fact` que a partir d'un nombre enter, n retorna el valor de $n!$.

En aquest exemple, la funció recursiva és la "traducció"a Python de l'anterior definició recurrent:

```
def fact (n):
    if n ==0:
        return 1
    else:
        return fact(n-1)*n
```

Per entendre el funcionament d'una funció recursiva també pot ser útil simular-ne l'execució.

La figura 1 mostra el funcionament del procés recursiu per l'exemple del factorial. A la part de dalt es mostra el procés recursiu cap endavant que consisteix en les successives crides recursives i on necessita recordar els valors dels paràmetres a cada crida. Després realitza el mateix nombre de passos enrere, on es recuperen els valors de cada crida que s'usen per calcular el resultat.

Per tal de gestionar aquestes procés, normalment s'usa una pila per guardar els successius valors de les variables a cada crida recursiva.

En aquest exemple, la pila contindria els successius valors de `n`.

Vegem amb detall els passos del procés recursiu cap endavant:

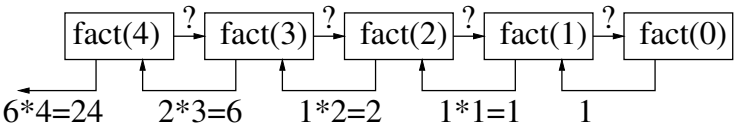

Figura 1: Simulació de l'execució de la funció recursiva `fact` que calcula el factorial d'un nombre.

- Suposem que fem la crida `fact(4)`: 1a crida recursiva o crida inicial
- com que $n = 3$ ($n \neq 0$), el control va a la branca `else` de la composició condicional i fa la crida `fact(2)`: 2a crida recursiva
- com que $n = 2$ ($n \neq 0$), el control va a la branca `else` de la composició condicional i fa la crida `fact(1)`: 3a crida recursiva
- com que $n = 1$ ($n \neq 0$), el control va a la branca `else` de la composició condicional i fa la crida `fact(0)`: 4a crida recursiva
- com que $n = 0$, el control va a la branca `if` de la composició condicional i la funció retorna 1: s'acaba la seqüència de crides recursives

Aquests són els passos del procés recursiu cap enrere:

- en aquest moment hi ha 4 crides a la funció que estan esperant resposta i el valor 1 retornat, retorna a la darrera d'aquestes crides (la 4a) on $n = 1$; aleshores es fa el producte 1*1 i retorna 1
- aquest 1 retorna a la 3a crida on $n = 2$; aleshores es fa el producte 1*2 i retorna 2
- aquest 2 retorna a la 2a crida on $n = 3$; aleshores es fa el producte 2*3 i retorna 6
- aquest 6 retorna a la 1a crida on $n = 4$; aleshores es fa el producte 6*4 i retorna 24
- com que aquesta era la crida inicial, aquí s'acaba el procés recursiu.

La recursivitat pot ser final (*tail recursion*) o no. En la recursivitat final la instrucció que acaba la funció recursiva és una crida recursiva sense cap operació, és a dir, la crida recursiva és la darrera expressió que s'avalua i que sovint és el que es retorna.

Algunes funcions es poden dissenyar amb recursivitat final, que té l'avantatge de poder ser més eficient.

Exemple 2: arrel digital

Donat un nombre natural, n, es defineix la seva arrel digital com el resultat de sumar els seus dígits, tot repetint el procés amb el nou nombre fins arribar a un nombre d'un sol dígit. Aquest dígit s'anomena l'arrel digital de n. Per exemple,

$374 \to 3 + 4 + 7 = 14 \to 1 + 4 = 5 \to \mathit{arrel\_digital}(374) = 5$

Dissenya la funció `arrel_digital` que a partir d'un enter, `n`, retorna la seva arrel digital.

```
def  arrel_digital(num) :
    if num < 10:
        return num
    else:
        suma = suma_dígits (num)
        return arrel_digital(suma)

def suma_dígits (num):
    it = map(int, str(num))
    return sum(it)
```

Observem, en primer lloc, que hem dissenyat la funció auxiliar `suma_digits` que a partir d'un enter retorna la suma dels seus dígits.

La funció recursiva `arrel_digital` té un cas base que és quan `num` té un sol dígit (`num<10`) i un cas recursiu (`num>=10`). En el cas recursiu primer es calcula la suma dels dígits de `num` i després fem la crida recursiva. Observem que la funció retorna directament el resultat de la crida recursiva: és un exemple de recursivitat final.

La funció `fact`, en canvi, no és un exemple de recursivitat final: quan rep el resultat de la crida recursiva, el multiplica per `n`.

La recursivitat pot ser simple (lineal) o múltiple. La recursivitat simple o lineal és aquella en la que cada cas recursiu genera una única crida recursiva. La recursivitat múltiple és aquella en la que cada cas recursiu genera més d'una crida recursiva.

Els dos exemples anteriors apliquen recursivitat simple (lineal): a cada crida es genera una altra crida.

El següent és un exemple senzill de recursivitat múltiple.

Exemple 3: funció d'Ackerman

La funció d'Ackerman, A, ve definida com:

$A(m,n) = n+1, m = 0$
$A(m,n) = A(m-1,1), m > 0, n = 0$
$A(m,n) = A(m-1, A(m,n-1)), m > 0, n > 0$

Dissenya la funció `ackerman` que a partir de dos enters, `m` i `n`, retorna el valor corresponent de la funció d'Ackerman descrita.

```
def ackerman(m, n):
    if m== 0:
       return n+1
    else:
       if n == 0:
          return ackerman(m-1, 1)
       else:
          p =  ackerman(m, n-1)
          return ackerman(m-1, p)
```

La 1a equació permet dissenyar el cas base. Les altres dues donen lloc a dos casos recursius. El primer cas recursiu és lineal i el segon és múltiple (doble) ja que cal cridar dos cops a la funció.

2.1 Límit de la recursivitat

Tots els llenguatges de programació, i Python en particular, estableixen un límit al nombre de crides recursives.

Podem consultar aquest límit fent:

```
>>> import sys
>>> sys.getrecursionlinit()
1000
```

La funció recursiva següent:

```
def fact (n):
    return n*fact(n-1)
```

és incorrecte. És fàcil veure que no hi ha cas base i, per tant, aquesta funció es queda en l'equivalent a un bucle sense fi:

```
>>> fact(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fact
  File "<stdin>", line 2, in fact
  File "<stdin>", line 2, in fact
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

El límit establert fa que després de 1000 crides recursives, el sistema operatiu pari una funció recursiva.

3 Tipologia d'exercicis

En aquest curs aprendrem a dissenyar funcions recursives de complexitat creixent que tractaran les temàtiques següents:

- successions de nombres
- recorregut i cerca en llistes
- funcions recursives modificadores
- creació de llistes
- funcions recursives i classes
- recursivitat múltiple

A les següents seccions s'analitzen exemples de cadascun d'aquests tipus.

4 Successions de nombres

La definició matemàtica de recurrència d'una successió ens porta de forma gairebé directa a la definició dels casos recursius i base necessaris en la definició d'una funció recursiva, tal com s'ha vist a l'exemple del factorial. En aquest apartat veurem altres exemples.

Exemple 4: terme del desenvolupament en sèrie de Taylor de e^x

El desenvolupament en sèrie de Taylor de e^x es defineix com:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

Dissenya la funció recursiva `terme_exp` que donat un real `x` i un enter `i` ($i \geq 0$), retorna el valor del terme t_i de l'anterior desenvolupament en sèrie de Taylor.

El càlcul dels termes de la successió anterior es pot fer usant la següent expressió recurrent, de la que s'obté directament el cas base i el cas recursiu:

$t_0 = 1$ cas base
$t_i = \frac{x^i}{i!} = \frac{x}{i}\frac{x^{i-1}}{(i-1)!} = \frac{x}{i}t_{i-1}$ cas recursiu

I la funció recursiva s'obté directament:

```
def terme_exp(x, i):
    if i == 0:
        return 1
    else:
        return x/i * terme_exp(x, i-1)
```

Exemple 5: aproximació del valor de la funció e^x pel desenvolupament en sèrie de Taylor

Dissenya la funció recursiva `exponencial_1` que donat un real `x` i un enter `n` ($n > 0$), retorna una aproximació de la funció e^x usant el desenvolupament en sèrie de Taylor (vist a l'anterior exemple) fins al terme n, inclòs.

Per dissenyar una funció recursiva que calcula un sumatori també ens podem basar en la definició recurrent d'un sumatori. El sumatori anterior, al que anomenem S, es pot expressar com:

$S_0 = 1$ sumatori fins al terme 0
$S_n = S_{n-1} + t_n$ sumatori fins al terme n

I la funció recursiva és:

```
def exponencial_1(x, n):
    if n == 0:
        return 1
    else:
        return terme_exp(x, n) + exponencial_1(x, n-1)
```

Observem que aquesta funció calcula un terme a cada crida recursiva i si revisem la funció `terme_exp` observem que per cada terme calcula tots els anteriors. Per tant, aquesta estratègia és molt poc eficient.

Atès que per calcular els termes també hem fet un disseny recursiu, ens podem plantejar calcular a la vegada el terme enèsim i la suma dels termes fins a l'enèsim. Un problema que comporta aquesta altra estratègia és que la funció que ens demanen ha de retornar el valor de l'aproximació de l'exponencial, mentre que la que estem especificant hauria de retornar dos valors: el terme i l'aproximació. La forma de resoldre aquest problema, que com es veurà s'aplica a molts altres casos, és dissenyant dues funcions: una funció recursiva (auxiliar) amb els valors de retorn (i els paràmetres, si convé) necessaris per la nostra especificació, i una altra funció (principal) que segueix l'especificació de l'enunciat, que no és recursiva i que bàsicament gestiona la crida i recollida de resultats de la funció auxiliar.

Aplicant aquesta estratègia, dissenyarem una altra versió de la funció anterior, que anomenarem `exponencial_2`. La funció auxiliar, que anomenarem `exponencial_2_r`, és la fusió de les dues funcions anteriors `terme_exp` i `exponencial_1`. La funció principal `exponencial_2` crida a la funció auxiliar i del tuple que aquesta li retorna, només retorna el segon element, corresponent al sumatori.

```
def exponencial_2(x, n):
    return exponencial_2_r(x, n)[1]

def exponencial_2_r(x, n):
    if n == 0:
        return 1, 1
    else:
        t, s = exponencial_2_r(x, n-1)
        t = t * x/n
        s = s + t
        return t, s
```

Revisant totes les funcions d'aquest exercici, observem que cap d'elles és recursiva final. Totes elles, després de la crida recursiva, realitzen altres operacions.

Exemple 6: nombres triangulars

Un nombre triangular es defineix com el resultat de sumar els n primers nombres naturals:

$T_n = \sum_{k=1}^{n} k = 1 + 2 + 3 + 4 + ... + n, n > 0$

Dissenya la funció recursiva `triangulars` que a partir d'un enter `n`, retorna l'enèsim nombre triangular.

Si expressem l'anterior definició de forma recurrent, ja tenim l'esquema de la funció recursiva:

$T_1 = 1$ cas base
$T_n = T_{n-1} + n$ cas recursiu

```
def triangulars(n):
    if n == 1:
        return 1
    else:
        return triangulars(n-1)+n
```

Exemple 7: màxim comú divisor

Dissenya la funció recursiva `mcd` que calcula el màxim comú divisor (mcd) de dos nombres donats, aplicant el mètode d'Euclides, que es basa en les següents propietats:

1. mcd(a, b) = mcd(b, a)
2. si a > b → mcd (a, b) = mcd (a - b, b)
   si a < b → mcd (a, b) = mcd (a, b - a)
3. si a = b, mcd(a, b) = a = b

El cas base ve donat per la tercera propietat i els dos casos recursius per la segona.

```
def mcd (a, b):
    if a == b:
        return a
    elif a > b:
        return mcd (a-b, b)
    else:
        return mcd (a, b-a)
```

Aquest és un exemple de recursivitat final.

5 Recorregut i cerca en llistes

Per aplicar un esquema de recorregut o cerca en una llista cal aplicar l'estratègia següent:

- cas recursiu: es selecciona un element de la llista i es parteix la llista en dues: la que té un element (el seleccionat) i la que té la resta dels elements. L'element individual es tracta i es fa la crida recursiva amb l'altra llista. Aquest element pot ser teòricament qualsevol. Però el més pràctic és seleccionar el primer.
- cas base: el cas trivial que se suposa que hem de saber resoldre és el d'una llista amb un sol element o una llista buida.

Exemple 8: sumar els elements d'una llista

Donada una llista d'enters, dissenya la funció recursiva `suma_llista_1` que retorni la suma dels seus elements.

- cas recursiu: seleccionar el primer element (posició 0) de la llista actual i fer la crida recursiva amb la llista que va des de l'element a la posició 1 fins al final
- cas base: llista buida: retorna 0

```
def suma_llista_1 (llista):
    if len(llista) ==0:
        return 0
    else:
        return llista[0]+ suma_llista(llista[1:])
```

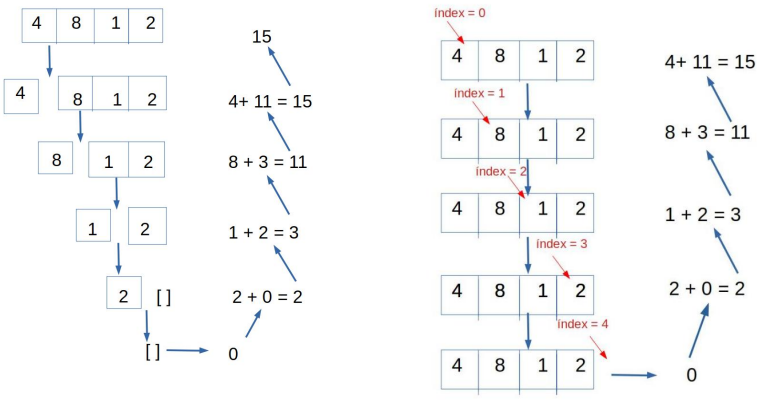

Figura 2: Simulació de l'execució de la funció recursiva per sumar els elements d'una llista. (esquerra): partint la llista explícitament, (dreta): usant índexs.

A la figura 2 (esquerra) es mostra un gràfic amb la simulació d'aquest procés recursiu per una llista de 4 elements.

En aquest disseny, la partició de la llista es fa literalment usant l'operador llesca (`[:]`). Això implica que a cada crida recursiva es fa una còpia d'aquesta part de la llista i, per tant, és un disseny ineficient pel que fa a la despesa de memòria.

Aquest problema es pot solucionar fent la partició de la llista de forma implícita, és a dir, usant un índex en lloc de partir literalment la llista, com es veu en el següent exemple.

Exemple 9: sumar els elements d'una llista, usant l'índex

Donada una llista d'enters, dissenya la funció recursiva `suma_llista_2` que retorni la suma dels seus elements (sense fer còpies de llistes).

Aplicarem la següent estratègia:

- cas recursiu: es selecciona un valor de l'índex i es consideren dues parts de la llista: la que conté l'element individual corresponent a l'índex i la que conté la resta dels elements. Es tracta l'element individual i es fa la crida recursiva amb l'altra llista, modificant adequadament l'índex. L'índex pot ser teòricament qualsevol. Però el més pràctic és començar amb l´index 0 i anar avançant d'un en un.
- cas base: el cas trivial que se suposa que hem de saber resoldre és el d'una llista amb un sol element o una llista buida. En aquest cas implica que l'índex sigui el del darrer element (llista amb un sol element), `idx = len(llista)-1` o més gran que el del darrer element (llista buida), `idx == len(llista)`

Per poder fer aquesta partició usant l'índex, la funció recursiva necessita tenir dos paràmetres: la llista i l'índex. De la mateixa manera com hem resolt a l'exemple 4 la funció `exponencial_2`, definirem una funció auxiliar recursiva amb aquests dos paràmetres i la funció principal que tindrà només el paràmetre corresponent a la llista:

- la funció principal fa la 1a crida recursiva inicialitzant l'índex a 0.
- cas recursiu: a cada crida recursiva l'índex s'incrementa en una unitat. Així la llista que es considera és la que comença en aquell valor de l'índex
- cas base: llista buida: `idx == len(llista)`

```
def suma_llista_2(llista):
    return suma_llista_2_r(llista, 0)

def suma_llista_2_r(llista, idx):
    if idx == len(llista):
        return 0
    else:
        return llista[idx]+ suma_llista_2_r(llista, idx+1)
```

En aquest cas no es realitzen còpies innecessàries de la llista ja que es treballa sempre amb la mateixa llista: la part de la llista a la que s'adreça cada crida recursiva es representa amb un índex que n'indica l'inici. Vegeu la figura 2 (dreta).

Acabem aquesta secció amb un exemple de cerca en una llista.

Exemple 10: trobar el primer múltiple d'un nombre en una llista

Donada una llista d'enters i un altre enter, `m`, dissenya una funció recursiva que retorni el valor del primer element de la llista que sigui múltiple de `m`. Si a la llista no n'hi ha cap, la funció ha de retornar el valor -1.

El disseny corresponent ha de definir els casos base i recursiu, que són com els vistos en els 2 exemples anteriors. També, com en tots els processos de cerca, cal veure quina és la propietat que s'està cercant: que sigui múltiple de `n`.

A continuació es mostren les funcions `multiple_1` i `multiple_2` que fan la cerca partint explícitament la llista i usant l'índex, respectivament:

```
def multiple_1(llista, m):
    if len(llista) ==0:
        return -1
    else:
        if llista[0]%m == 0:
            return llista[0]
        else:
            return multiple(llista[1:], m)
```

```
def multiple_2(llista, m):
    return multiple_2_r(llista, m, 0)

def multiple_2_r(llista, m, idx):
    if idx == len(llista):
        return -1
    else:
        if llista[idx]%m == 0:
            return llista[idx]
        else:
            return multiple_2_r(llista, m, idx+1)
```

Observem que la funció no només acaba quan s'arriba al cas base sinó quan es troba el primer element que compleix la propietat. Aquesta s'avalua a l'element de la llista seleccionat. A la funció `multiple_1` és sempre l'element a la posició `0` de la part de la llista de cada nivell de recursivitat. A la funció `multiple_2` és l'element a la posició `idx`, ja que la llista és sempre la inicial.

Exemple 11: trobar la posició del primer múltiple d'un nombre en una llista

Donada una llista d'enters i un altre enter `m`, dissenya una funció recursiva que retorni la posició del primer element de la llista que sigui múltiple de `m`. Si a la llista no n'hi ha cap, la funció ha de retornar el valor -1.

Aquest exemple és similar a l'anterior, però com que demanen la posició, la forma més simple d'enforcar-lo és usant l'esquema amb índex:

```
def pos_multiple_index(llista, m):
    return pos_multiple_index_r(llista, m, 0)

def pos_multiple_index_r(llista, m, idx):
    if idx == len(llista):
        return -1
    else:
        if llista[idx]%m == 0:
            return idx
        else:
            return pos_multiple_index_r(llista, m, idx+1)
```

6 Funcions recursives modificadores

En les funcions recursives en les que es demana modificar els seus elements, la millor estratègia passa també per usar l'índex. En aquest cas no és només per evitar còpies de la llista sinó també perquè la modificació d'un element d'una llista requereix referir-lo usant l'índex.

Exemple 12: modificar una llista de paraules

Donada una llista de paraules (strings), `lpar`, i un prefix (string), `prefix`, dissenya la funció recursiva **modificadora** `aplica_prefix` que **modifiqui** la llista de manera que cada paraula vagi precedida pel prefix i hi hagi un guió entre el prefix i la paraula. Per exemple:

```
>>> lpar = ['potent', 'disposat', 'judici']
>>> aplica_prefix(lpar, 'Pre')
>>> lpar
['Pre-potent', 'Pre-disposat', 'Pre-judici']
```

Dissenyarem un parell de funcions: la principal i l'auxiliar recursiva que seran totes dues modificadores:

```
def aplica_prefix(lpar, prefix):
    aplica_prefix_r(lpar, prefix, 0)

def aplica_prefix_r(lpar, prefix, idx):
    if idx < len(lpar):
        lpar[idx] = prefix + '-' + lpar[idx]
        aplica_prefix_r(lpar, prefix, idx+1)
```

La funció principal `aplica_prefix` només fa la crida a la funció recursiva `aplica_prefix_r`. En aquesta, el cas base és `idx == len(lpar)` i com que en aquest cas no s'ha de fer res en particular, no cal incloure la branca de l'`else`. El cas recursiu, que correspon a la condició `idx < len(lpar)`, primer tracta l'element en curs `lpar[idx]` i després fa la crida recursiva incrementant `idx` en 1.

7 Creació de llistes

Les funcions recursives que creen llistes també es poden dissenyar fent còpies de llistes o de manera eficient sense fer-ne.

Exemple 13: creació d'una llista amb els nombres triangulars

Donat un enter `n` dissenya la funció recursiva `llista_triang_1` que retorna una llista amb els `n` primers nombres triangulars.

```
def llista_triang_1(n):
    if n == 1:
        return [1]
    else:
        tr = llista_triang_1(n-1)
        nou = tr[-1] + n
        tr.append(nou)
        return tr
```

El cas base i recursiu són els mateixos que en l'exemple 6 que calcula l'enèsim nombre triangular. El cas base és quan `n == 1`. En aquest cas, la funció retorna una llista amb el primer nombre triangular. En el cas recursiu també fem la crida recursiva per `n-1`. Aquesta ens retorna la llista amb els `n-1` primers nombres triangulars, `tr`. Aleshores hem de calcular l'enèsim nombre triangular sumant a l'anterior nombre triangular, que està a la darrera posició de la llista retornada, el valor `n` del nivell de la recursivitat actual i després afegir aquest valor al final de la llista retornada `tr` i retornar aquesta llista.

En aquest disseny a cada nivell de la recursivitat es crea una nova llista `tr` amb un element més i, per tant, també és ineficient pel que fa a la memòria.

En els processos de recorregut i cerca en llistes s'ha solucionat aquest problemes usant un índex. En el processos de creació de llistes usarem un esquema eficient que utilitzi sempre la mateixa llista i que es basa també en l'ús d'una funció principal i una d'auxiliar.

```
def llista_triang_2(n):
    tr = []
    llista_triang_2_r(n, tr)
    return tr

def llista_triang_2_r(n, tr):
    if n == 1:
        tr.append(1)
    else:
        llista_triang_2_r(n-1, tr)
        tr.append(tr[-1] + n)
```

La funció principal, `llista_triang_2`, inicialitza la llista `tr`; després crida a la funció auxiliar recursiva, la qual va afegint els elements a la llista, i, finalment, retorna la llista `tr`. La funció auxiliar recursiva, `llista_triang_2_r`, tant en el cas base com en el recursiu, afegeix l'element corresponent a la llista `tr` usant el mètode modificador `append`. Notem que la funció `llista_triang_2_r` no retorna res; el que fa és **modificar** la llista (hi afegeix elements), és a dir, és una funció **modificadora**.

Exemple 14: índexs de suma donada (variant examen final quad 2017-18-P)

Dissenya una funció recursiva `sumac` que, donades dues llistes de nombres enters `v1` i `v2` i un valor `s`, retorna una altra llista formada per valors enters que són els índexs `idx` de les llistes `v1` i `v2` tals que la suma dels elements i-èsims de `v1` i `v2` és igual a `s`. Exemples:

```
>>> v1 = [0, 4, 3, 2, 8, 9, 25, 12]
>>> v2 = [7, 5, 6, 5, 4, 0]
>>> sumac(v1, v2, 9)
[1, 2, 5]
>>> sumac(v1, v2, 7)
[0, 3]
```

En aquest exemple cal recórrer dues llistes i crear-ne una altra en paral·lel. Per tant usarem conjuntament els esquemes vistos pel recorregut i creació de llistes. La solució que es mostra a continuació segueix l'esquema eficient i usa un índex `idx` per recórrer les dues llistes donades i una funció principal que inicialitza aquest índex i també la llista que cal crear. La funció recursiva auxiliar, utilitza l'index i modifica la llista a retornar afegint els elements necessaris.

Observacions:

- el recorregut en paral·lel de `v1` i `v2` s'acaba quan s'acaba la llista més curta: condició `idx>=min(len(v1),len(v2))`. Aquest és el cas base.
- la sentència prèvia a la crida recursiva, afegeix un element (`idx`) al final de la llista que es crea només si es compleix la propietat indicada a l'enunciat: `v1[idx] + v2[idx] == s`

```
def sumac(v1, v2, s):
    lind = []
    sumac_r(v1, v2, s, 0, lind)
    return lind

def sumac_r(v1, v2, s, idx, lind):
    if idx >= min(len(v1), len(v2)):
        pass
    else:
        if v1[idx] + v2[idx] == s:
            lind.append(idx)
        sumac_r(v1, v2, s, idx+1, lind)
```

8 Funcions recursives i classes

Tot el vist anteriorment es pot aplicar a elements de qualsevol dels tipus coneguts i també a instàncies de classes prèviament definides.

Exemple 15: llista d'instàncies a la classe `datetime.date`

Donada una llista de dates (instàncies a la classe `datetime.date`) i un enter que representa un any, dissenya la funció `abans` que retorna un tuple amb el dia, mes i any de la primera data de la llista donada que sigui de l'any donat. Retorna el tuple `(0, 0, 0)` si no en troba cap. Exemples:

```
>>> ldates = [(2022, 12, 3), (2021, 4, 3), (2019, 3, 21), (2020, 4, 12),
...   (2019, 5, 31), (2021, 9, 28), (2022, 10, 2), (2018, 8, 5), (2020, 7,
   14)]
>>> for i in range(len(ldates)):
...    ldates[i] =datetime.date(ldates[i][0], ldates[i][1], ldates[i][2])
>>> abans(ldates, 2021)
(3, 4, 2021)
>>> abans(ldates, 2019)
(21, 3, 2019)
>>> abans(ldates, 2022)
(3, 12, 2022)
```

Hem d'aplicar un esquema de cerca com els vistos a l'apartat 5 a una llista de dates.

```
def abans(ldates, anno):
    return abans_r(ldates, anno, 0)

def abans_r(ldates, anno, i):
    if i == len(ldates):
        return (0, 0, 0)
    else:
        if ldates[i].year == anno:
            return (ldates[i].day, ldates[i].month, ldates[i].year)
        else:
            return abans_r(ldates, anno, i+1)
```

Exemple 16: llista d'instàncies a la classe Artropode

En aquest exemple usarem la classe Artropode. Vegeu-ne l'especificació a la figura 3.

Donada una llista d'instàncies a la classe Artropode i un string amb el nom d'un tipus d'apèndix, dissenya la funció `apendix` que retorna una altra llista corresponent als artròpodes de la llista donada que tenen apèndixs del tipus donat. Els elements de la llista retornada son tuples amb tres elements: el nom comú de l'artròpode, el nombre total d'apèndixs que té i el nombre d'apèndixs del tipus indicat que té. Exemple:

```
>>> a1 = Artropode('Latrodectus tredecimguttatus', 'Viuda negra', 2)
>>> a1['ullals'], a1['pedipalps'] = 2, 2
>>> f = Artropode('Formica rufa', 'Formiga roja', 3)
>>> f['potes'], f['antenes'], f['mandibules'] = 6, 2, 2
>>> p = Artropode('Palinurus elephas', 'Llagosta vermella', 2)
>>> p['potes'] , p['antenes'] = 10, 4
>>> lartro = [a1, f, p]
>>> apendix(lartro, 'potes')
[('Formiga roja', 10, 6), ('Llagosta vermella', 14, 10)]
>>> apendix(lartro, 'ullals')
[('Viuda negra', 4, 2)]
>>> apendix(lartro, 'pinces')
[]
```

class artropode.**Artropode**(*n_cientific*, *n_comu*, *n_tagmes*)

Atributs públics:

nom_cientific
Nom científic de l'espècie (string)

nom_comu
Nom comú de l'espècie (string)

nombre_tagmes
Nombre de tagmes del cos (enter positiu)

Mètodes:

num_apendixs()
Retorna el nombre total d'apèndixs de l'artròpode.

Aquesta classe ha de suportar les **operacions** següents:

Operació	Resultat
`a[t]`	Retorna el nombre d'apèndixs de tipus *t* (string) de l'artròpode *a*; 0 si no en té d'aquest tipus.
`a[t] = n`	Estableix que l'artròpode *a* té *n* (enter) apèndixs de tipus *t* (string).

Figura 3: Especificació de la classe Artropode

Com a l'exemple 14 de l'apartat anterior, hem de recórrer una llista, `lartro` i crear-ne una altra, `lapen`:

```
def apendix(lartro, nom_apendix):
    lapen = []
    apendix_r(lartro, nom_apendix, 0, lapen)
    return lapen

def apendix_r(lartro, nom_apendix, idx, lapen):
    if idx < len(lartro):
        bestiola = lartro[idx]
        n_apen = bestiola[nom_apendix]
        if n_apen != 0:
            lapen.append((bestiola.nom_comu, bestiola.num_apendixs(), n_apen)
                )
        apendix_r(lartro, nom_apendix, idx+1, lapen)
```

Per definir els elements de la llista a retornar, s'ha usat l'atribut `nom_comu`, el mètode `num_apendixs` i l'operació d'indexació de la classe Artropode.

Exemple 17: classe amb un mètode recursiu

En aquest exemple usarem les classes Seient i Sala. Vegeu-ne l'especificació a la figura 4.

La classe Seient

La classe **Seient** representa el seient d'una sala

class seient.**Seient**(*lliure=True, tipus='PALA-DRETA'*)

Atributs:

lliure
Booleà que indica l'estat del seient: lliure (`True`) o no (`False`).

tipus
String que indica el tipus de seient. Pot ser: `'PALADRETA'`, `'PALAESQUERRA'` o `'SENSEPALA'`, en funció de si té pala i a quin costat la té.

Mètodes:

tePala()
Retorna un booleà indicant si el seient té pala (`True`) o no (`False`).

tePalaDreta()
Retorna un booleà indicant si el seient té pala dreta (`True`) o no (`False`).

estaLliure()
Retorna un booleà indicant si el seient està lliure (`True`) o no (`False`).

ocupa()
Canvia l'estat del seient a ocupat.

allibera()
Canvia l'estat del seient a lliure.

treuPala()
Canvia l'atribut tipus a `'SENSEPALA'`

posaPala(*pala*)
Canvia l'atribut tipus a pala, que pot ser `'PALADRETA'` o `'PALAESQUERRA'`

La classe **Seient** també suporta la funció **str** que retorna un string que representa un seient que pot ser dels 6 tipus següents: `'PD-L'`, `'PE-L'`, `'SP-L'`, `'PD-O'`, `'PE-O'`, `'SP-O'`, on `'PD'`, `'PE'` i `'PD'` signifiquen pala dreta, esquerra i sense pala respectivament i `'L'` significa seient lliure i `'O'` ocupat.

La classe Sala

Dissenyeu la classe **Sala** que representa una sala amb un nombre de files i un nombre de seients a cada fila (instàncies de la classe **Seient**), que serà sempre el mateix. Tant la numeració de les files com de les columnes comença per 1. La primera fila és la més propera a l'escenari i el seient de més a l'esquerra és el primer seient de cada fila. Deseu aquesta classe al fitxer `sala.py`. L'especificació d'aquesta classe és la següent:

class sala.**Sala**(*nfiles=10, nseients=10*)

Atributs públics:

nfiles
Nombre de files

nseients
Nombre de seients per cada fila (el mateix per totes)

Aquesta classe ha de suportar les **operacions** següents:

Operació	Resultat
`s[pos]`	Retorna la instància de la classe **Seient** que ocup la posició *pos* de la sala *s*
`s[pos] = c`	Posa una instància de la classe **Seient**, *c*, a la posició *pos* de la sala *s*

La posició *pos* ha de ser un tuple (fila, seient) indicant el número de la fila i del seient dins la fila.

La classe **Sala** també suporta la funció **str** que retorna un string que representa la sala amb el format següent: cada seient es representa pel seu string corresponent i, a més, cada seient està separat del seient contigu per un espai en blanc (" ") i entre cada parell de files consecutives hi ha un salt de línia ("\n").

Figura 4: Especificació de les classes Seient i Sala.

En aquest exemple es demana ampliar la classe `Sala` amb el mètode `ocupacio` que retorna el percentatge de seients ocupats a la sala. Aquest mètode usarà un mètode privat recursiu que calculi el nombre de seients ocupats de la sala. Recordem que un mètode privat no es pot cridar des de fora de la classe i per distingir-lo, tal com es fa amb els atributs privats, el nom es precedeix per dos guions baixos. Anomenarem aquest mètode privat `__ocupats`. Exemples:

```
>>> sa = Sala(3, 4)
>>> sa[1,3].ocupa()
>>> round(sa.ocupacio(), 1)
8.3
>>> sa[1,1].ocupa()
>>> sa[1,2].ocupa()
>>> round(sa.ocupacio(), 1)
25.0
```

Vegem la implementació:

```
    def ocupacio(self):
        total = self.nfiles * self.nseients
        plens = self.__ocupats(0, 0)
        return plens/total*100

    def __ocupats(self, i, j):
        if i == self.nfiles:
            return 0
        else:
            i2, j2 = i, j + 1
            if j2 == self.nseients:
                i2, j2 = i + 1, 0
            no = self.__ocupats(i2, j2)
            if not self.__seients[i][j].lliure:
                no = no + 1
            return no
```

El mètode recursiu `__ocupats` fa un recorregut dels seients de la sala i compta els que estan ocupats (no estan lliures). El recorregut seient a seient comença pel seient a la posició `(0, 0)` i segueix per tots els seients de la fila `0`; quan aquesta s'acaba va a la fila següent. El codi per anar al seient següent és:

```
i2, j2 = i, j + 1        # anem al seient següent de la mateixa fila i
if j2 == self.nseients: # si s'han acabat els seients de la fila i
    i2, j2 = i + 1, 0    # anem al seient 0 de la fila següent, i+1
```

I el cas base es detecta quan s'han acabat les files. Tot i que l'estructura de la sala és matricial, aquest recorregut segueix un esquema lineal.

9 Recursivitat múltiple

La recursivitat múltiple és aquella en la que cada cas recursiu genera més d'una crida recursiva.

Exemple 18: suma dels elements d'una llista revisitat

A l'apartat 5 es presenta l'estratègia de partir la llista en dues de manera que una part és una llista amb un sol element i l'altra part és la llista amb la resta d'elements. Si en lloc de fer aquesta partició, partíssim la llista per la meitat, tindríem un esquema recursiu amb el mateix cas base però el cas recursiu donaria lloc a dues crides recursives: una per cada meitat de la llista. Exemple:

Primera versió, partint la llista explícitament: funció `suma_llista_3`:

```
def suma_llista_3(llista):
    if llista == []:
        return 0
    elif len(llista) == 1:
        return llista[0]
    else:
        mig = len(llista)//2
        return suma_llista_3(llista[:mig]) + suma_llista_3(llista[mig:])
```

Segona versió, usant índexs: funció `suma_llista_4`

Per definir cada part de la llista necessitarem dos índexs que indiquin respectivament, el primer i darrer element de la llista, `inici` i `fi`. En ela exercicis vistos fins ara, només calia un índex (el del primer element) perquè el darrer element era sempre el darrer de la llista inicial. El cas base també es pot definir com el d'una llista amb un element (condició `inici == fi` o com el d'una llista buida (condició `inici >= fi`. Aquest darrer és el que s'usa al disseny que es mostra tot seguit

```
def suma_llista_4(llista):
    return suma_llista_4_r(llista, 0, len(llista)-1)

def suma_llista_4_r(llista, inici, fi):
    if inici > fi:
        s = 0
    elif inici == fi:
        s = llista[inici]
    else:
        mig = (fi + inici) // 2
        s = suma_llista_4_r(llista, inici, mig) + \
            suma_llista_4_r(llista, mig+1, fi)
    return s
```

Exemple 19: avaluació d'una expressió

Un arbre binari és una estructura de dades on cada node té com a molt dos fills. Un arbre binari d'expressions és un arbre binari que permet representar expressions aritmètiques. En aquesta estructura els nodes fulla (els que no tenen descendents) representen valors de l'expressió i els altres nodes representen operacions. Vegeu la figura 5.

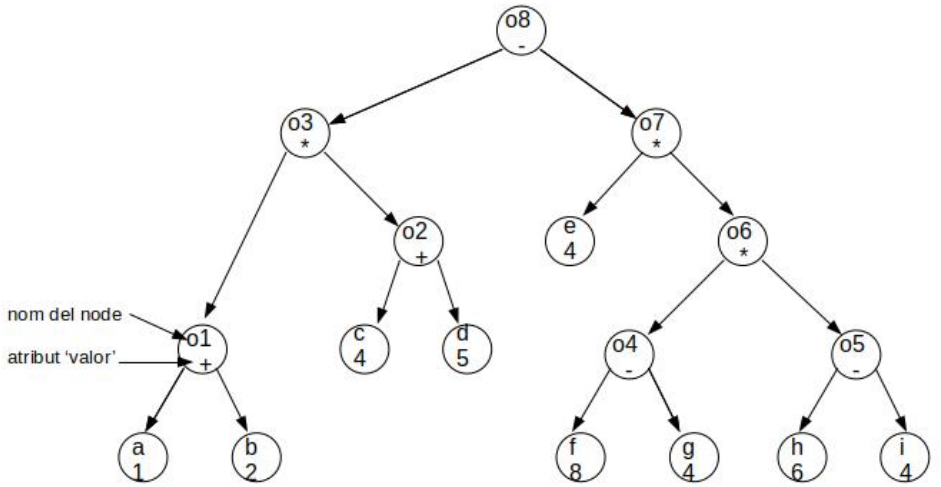

Figura 5: Arbre binari d'expressions. Exemple que representa l'expressió: $(1+2)*(4+5)-4*(8-4)*(6-4)$

Representarem un arbre binari d'expressió amb un graf dirigit (classe `DiGraph` de `networkx`). Els nodes tenen el seu nom i un atribut de node anomenat `'valor'` on hi ha un enter amb el valor numèric en el cas dels nodes fulla o un string amb l'operació en els altres nodes.

Dissenya la funció recursiva `expressio` que a partir d'un arbre binari d'expressió representat amb un `DiGraph` i el nom d'un node, retorni el resultat d'avaluar l'expressió a partir del node donat. Exemples:

```
>>> g = nx.DiGraph()
>>> g.add_node('a', valor = 1)
>>> g.add_node('b', valor = 2)
>>> g.add_node('c', valor = 4)
>>> g.add_node('d', valor = 5)
>>> g.add_nodes_from (['o1', 'o2'], valor = '+')
>>> g.add_node('o3', valor = '*')
>>> g.add_edges_from ([('o1', 'a'), ('o1', 'b'), ('o2', 'c'), ('o2', 'd'), ('
    o3', 'o1'), ('o3', 'o2')])
>>> expressio(g, 'o3')
27
>>> expressio(g, 'o2')
9
>>> expressio(g, 'b')
2
```

El cas base correspon als nodes fulla, és a dir, els que no tenen successors. Quan el node no és una fulla, cal fer una crida recursiva per cadascun dels seus dos descendents: recursivitat múltiple (doble):

```
def expressio(g, node):
    if g.out_degree(node) == 0:
        return g.nodes[node]['valor']
    else:
        n1, n2 = g.successors(node)
        r1 = expressio(g, n1)
        r2 = expressio(g, n2)
        oper = g.nodes[node]['valor']
        return eval(str(r1) + oper + str(r2))
```

L'expressió `g.nodes[node]['valor']` és el valor del node: un enter pels node fulla i un string amb l'operació pels altres nodes. La funció `eval` permet avaluar una expressió representada mitjançant un string.

Exemples de recursivitat múltiple

Altres exemples de recursivitat múltiple els podem trobar a la col·lecció d'exercicis. Per exemple:

- Catifa de Sierpiński: recursivitat de grau 8
- Quadtree: recursivitat de grau 4
- Triangle de Sierpiński: recursivitat triple

Exemple 21: recorregut d'un graf

Per acabar, analitzarem el mètode de cerca en profunditat, *depth first search, dfs*, en un graf.

Donat un graf dirigit que representa una determinada relació entre persones (com pot ser *és progenitor* o *és el cap*, ...) i un node, dissenya la funció recursiva `dfs_recursiu` que retorni la llista dels nodes descendents del node aplicant una cerca en profunditat i en preordre.

```
def dfs_recursiu(g, node):
    l = []
    dfs_recursiu_r(g, node, l)
    return l

def dfs_recursiu_r(g, node, l):
    l.append(node)
    for x in g.successors(node):
        if x not in l:
            dfs_recursiu_r(g, x, l)
```

Cal aplicar recursivitat múltiple i el grau és variable ja que cada node pot tenir un nombre variable de successors: per això cal la sentència `for`. El cas base no queda reflectit explícitament; es dóna quan un node no té successors: el `for` no s'executarà i, per tant, no hi haurà crides recursives.

En aquest exemple s'ha usat un graf dirigit que representa un arbre, per facilitar la comprensió. La funció equivalent per un graf no dirigit, només ha de substituir l'expressió `g.successors(node)` per `g.neighbors(node)`.

A continuació es mostra un exemple s'aplicació. A la figura 6 es mostra el graf dirigit `ga` i tot seguit alguns exemples d'execució de la funció:

```
>>> dfs_recursiu(ga, 'robert')
['robert', 'angela', 'andreu', 'pere', 'loli']

>>> dfs_recursiu(ga, 'sarah')
['sarah', 'lluis', 'eric', 'robert', 'angela', 'andreu', 'pere', 'loli', '
    albert', 'laia', 'nuria', 'ernest', 'jordina', 'marga']

>>> dfs_recursiu(ga, 'laia')
['laia', 'nuria', 'ernest', 'jordina']
```

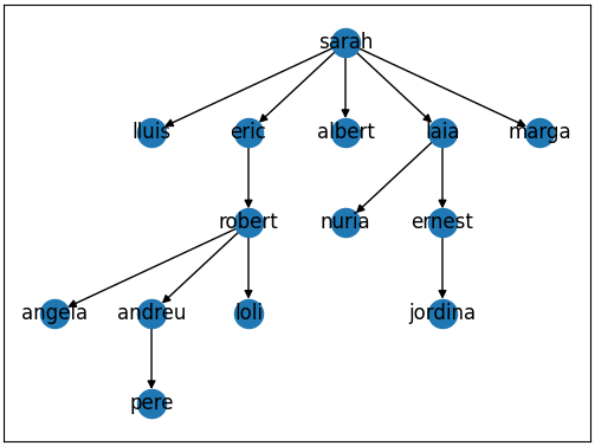

Figura 6: Exemple de graf dirigit, `ga`.

Només a tall de comparació, tot seguit es mostra la versió iterativa d'aquesta funció en la que com es pot veure cal fer ús d'una estructura de dades de tipus pila per tal que els nodes s'obtinguin en l'ordre de cerca en profunditat i preordre.

```
def dfs_iteratiu(g, node):
    l = []
    pila = LifoQueue()
    pila.put(node)
    while not pila.empty():
        v = pila.get()
        l.append(v)
        for x in reversed(list(g.successors(v))):
            if x not in l:
                pila.put(x)
    return l
```

10 Curiositats

Recursivitat i humor:

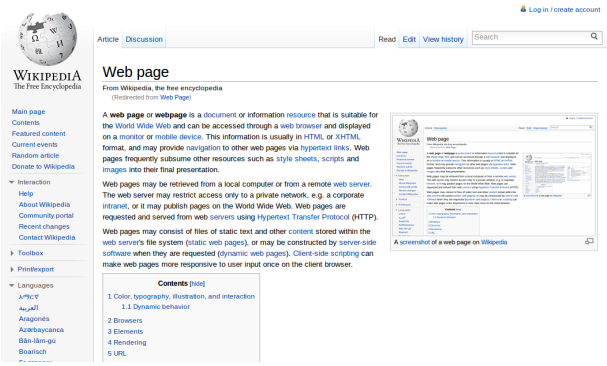

Figura 7: Definició de pàgina web a la wikipedia

- definició d'una pàgina web a la wikipedia. Vegeu la figura 7
- "Recursion, see Recursion."
- "If you already know what recursion is, just remember the answer. Otherwise, find someone who is standing closer to Douglas Hofstadter [1] than you are; then ask him or her what recursion is." Andrew Plotkin
- Acrònims recursius
  - PHP: *PHP Hypertext Preprocessor*
  - GNU: *GNU's not Unix*

Recursivitat i poesia:

Rose is a rose is a rose is a rose. Gertrude Stein 1922.

En aquest exemple no hi ha cas base, però és una poesia!

[1] Douglas Hofstadter es un científic i filòsof que va publicar el llibre *Gödel, Escher, Bach: an Eternal Golden Braid*, que va guanyar el premi Pulitzer d'assaig el 1980.

Recursivitat i art:

En el mon de l'art i el disseny hi podem trobar molts exemples del que es coneix com a *posada en abisme* (*mise en abîme*). Vegeu-ne alguns a les figures 8 i 9.

Figura 8: Exemples de *mise en abîme*:
(esquerra) Llauna de cacau Droste.
(dreta) El matrimoni Arnolfini. Jan van Eyck (1434).

Figura 9: Exemples de *mise en abîme*:
(esquerra) Las meninas. D. Velázquez (1656). Velázquez pintant-se ell mateix.
(dreta) Print Gallery. M. C. Escher (1956). Mostra una ciutat distorsionada que conté una sala d'exposicions que recursivament conté el propi quadro, i així fins a l'infinit.