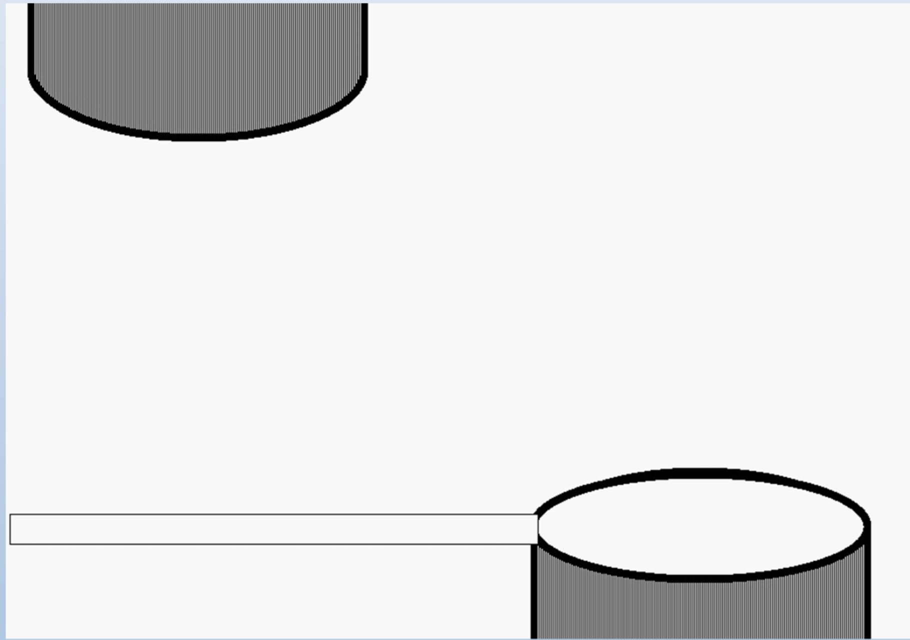
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica etsinf

Tema 1. Recursió



Programació (PRG)

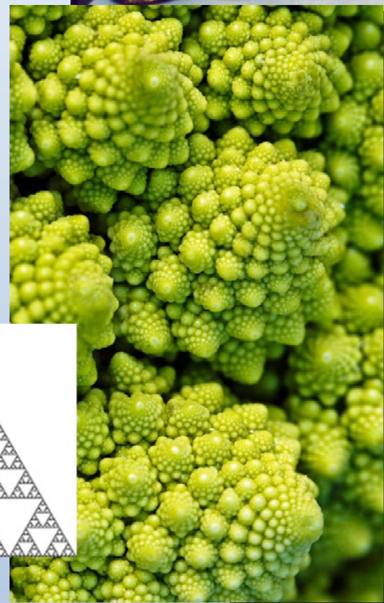
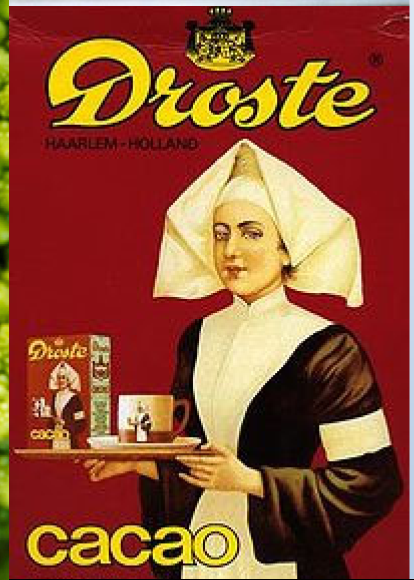
Curs 2019/2020

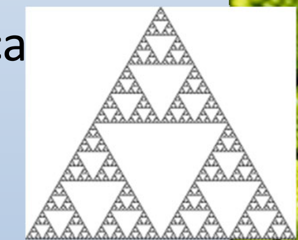
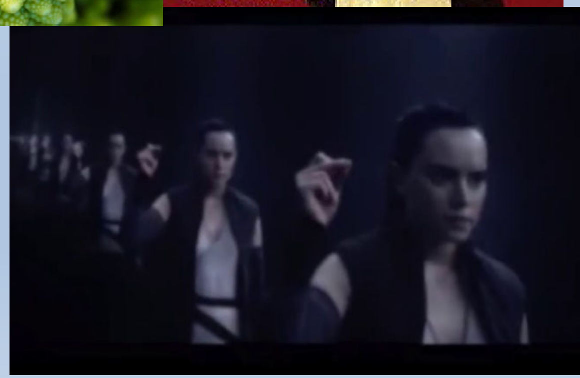
Departament de Sistemes Informàtics i Computació

Continguts

Duració: 6 sessions

1. Introducció
2. Disseny d'un mètode recursiu
3. Tipus de recursió
4. Recursivitat i pila de crides
5. Alguns exemples
6. Recursió amb arrays: recorregut i cerca
   – Esquemes recursius de recorregut
   – Esquemes recursius de cerca
   – Cerca binària recursiva
7. Recursió versus iteració

Introducció

- Recursiu és qualsevol ent (definició, procés, estructura, etc.) que es defineix en funció de si mateix.

- funció recursiva, es la que la seua resolució per a un problema s'obté amb la solució prèvia de la mateixa funció per a un problema més senzill.

- Per exemple, la funció factorial:

$$n! = \begin{cases} 1 & n = 0 \\ \displaystyle\prod_{i=1}^{n} i & n > 0 \end{cases} \qquad n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

Introducció

Conjunt d’instruccions que defineixen com fer la resolució d’un problema en un **temps finit**.

- Un algorisme és recursiu si:
  1. Es un Algorisme
  2. I obté la solució d'un problema amb els resultats que obtessos per a casos més senzills del mateix problema, és a dir, si s’invoca a si mateix amb unes dades mes xicotetes...

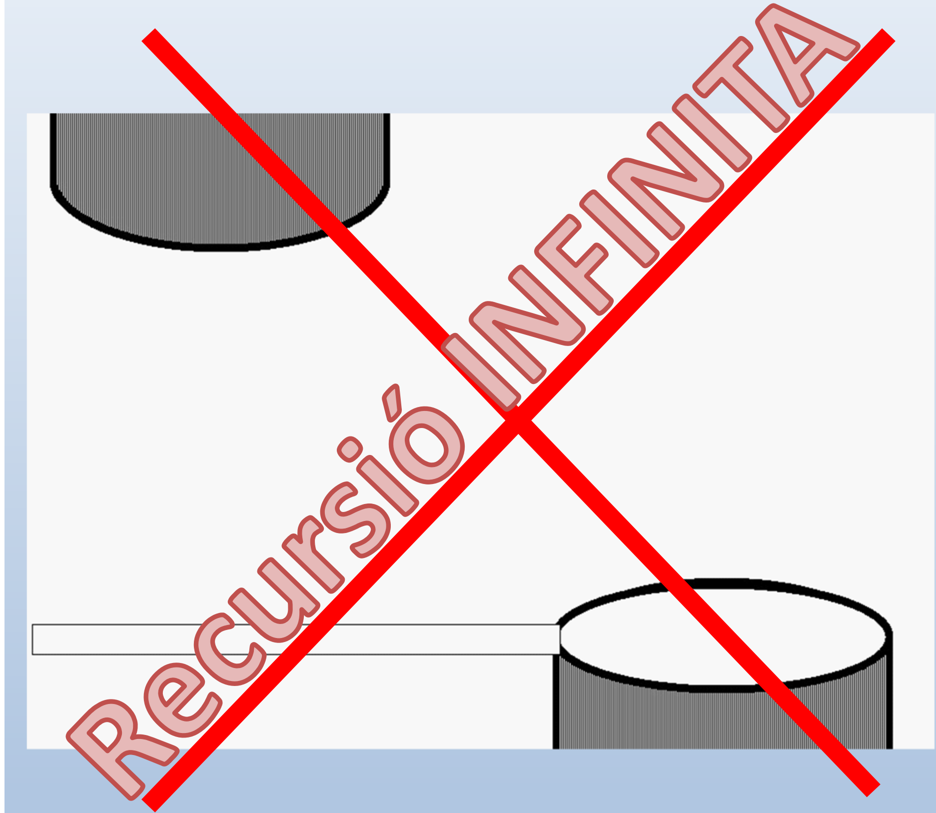
- Així, es descompon el problema en una sèrie de problemes del mateix tipus però més simples

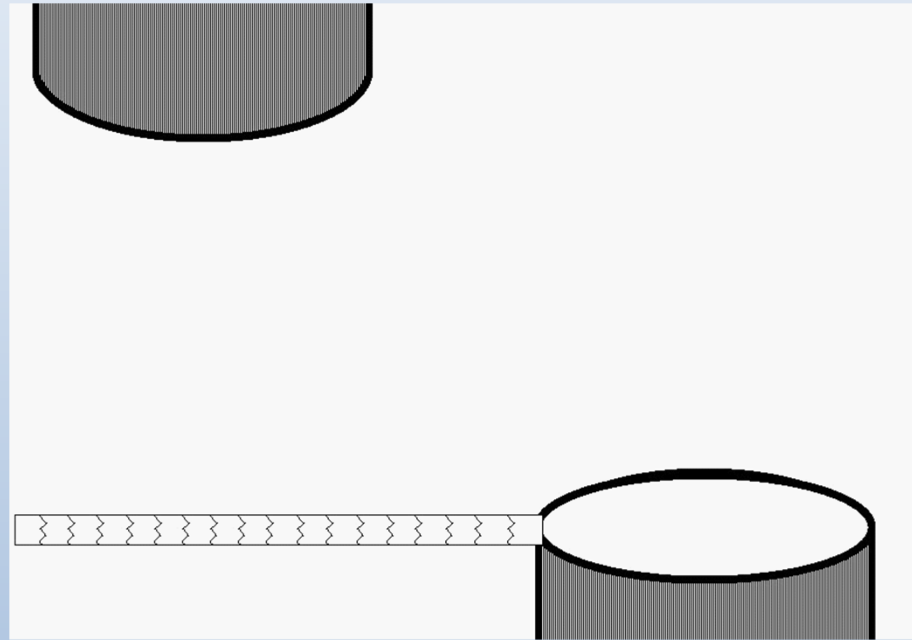
- usant el mateix algorisme, aplicat a problemes més simples

- La solució s’obté amb les solucions dels problemes més simples.

Introducció

Conjunt d'instruccions que defineixen com fer la resolució d'un problema en un **temps finit**.

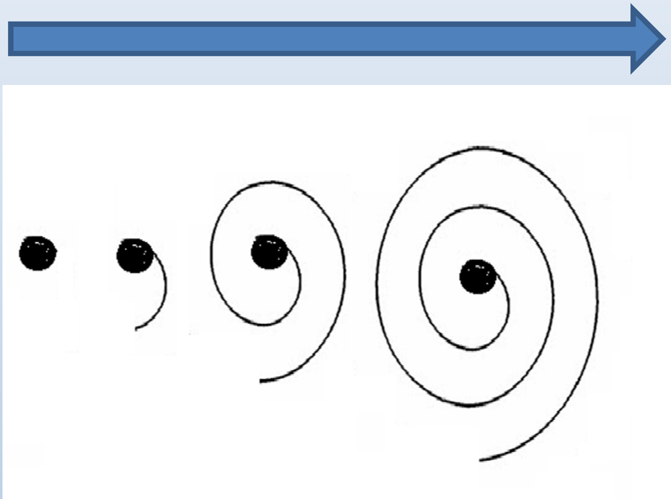




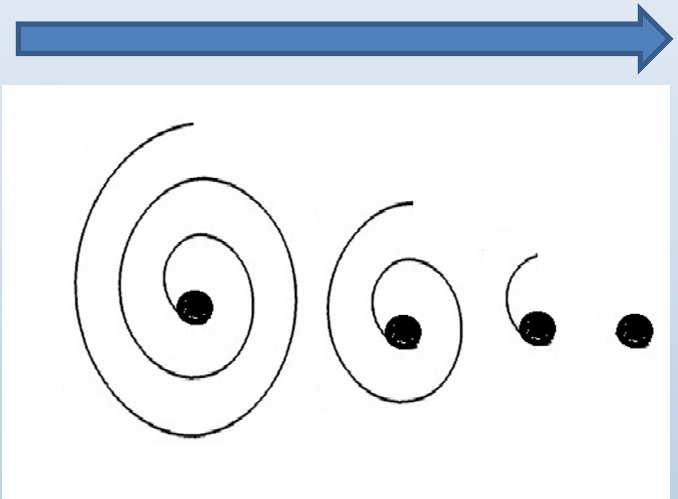
Qual de les dues imatges podríen representar un algorisme recursiu?

Inducció - Recursió

Inducció



Recursió



Introducció

- Hi ha dues situacions diferents a l'hora de resoldre el problema plantejat:
  - Cas base: el problema és lo suficientment simple per a ser resolt de manera trivial.
  - Cas general: el problema requereix l'ús de la resolució d'una instància més simple per trobar la seua solució.

Introducció

- Ja plantejat un algorisme recursiu és necessari verificar:

    1. La terminació de l'algorisme, és a dir, que s'arribarà al cas base siga el que siga el cas inicial.

    2. La correcció de l'algorisme, és a dir, que la solució de l'algorisme és correcta per a qualsevol cas

    (sol requerir demostració matemàtica per inducció sobre algun paràmetre).

Disseny d’un mètode recursiu

- Els algorismes recursius s’implementen fàcilment usant mètodes recursius.

```
/** PRECONDICIÓ: n>=0 */
public static int factorial(int n) {
    if (n == 0) {    // Condició del cas base
        return 1;    // Instruccions del cas base
    }
    else {                              // n>0 Condició del cas general
        return n * factorial(n-1);// Instruccions del cas general
    }
}
```

- L'estructura fonamental: <u>una instrucció condicional</u>
  (cas base o cas general.)
- Crides recursives <u>sempre</u> per a casos estrictament <u>més simples</u>.

Disseny d’un mètode recursiu - Etapes

1. Enunciat del problema.
2. Anàlisi de casos.
3. Transcripció de l'algorisme a un mètode en Java.
4. Validació del disseny.

Disseny d’un mètode recursiu - Etapes

1. Enunciat del problema.

Declarar la capçalera del mètode i establir:
- *les condicions d’entrada (precondició) per a les quals s'haurà d'executar l'algorisme,*
- *el resultat esperat d’aquesta execució.*

2. Anàlisi de casos.

Identificar:
- *cas base i el cas general de la recursió*
- *les instruccions per a resoldre cada cas.*
- *comprovar que els casos són complementaris i excloents (es cobreix qualsevol cas possible).*

3. Transcripció de l'algorisme .

Construir la instrucció condicional,
- en el cas base la resposta
- en el cas general :
  - les instruccions de reducció del problema,
  - la crida o crides recursives i La combinació de les respostes

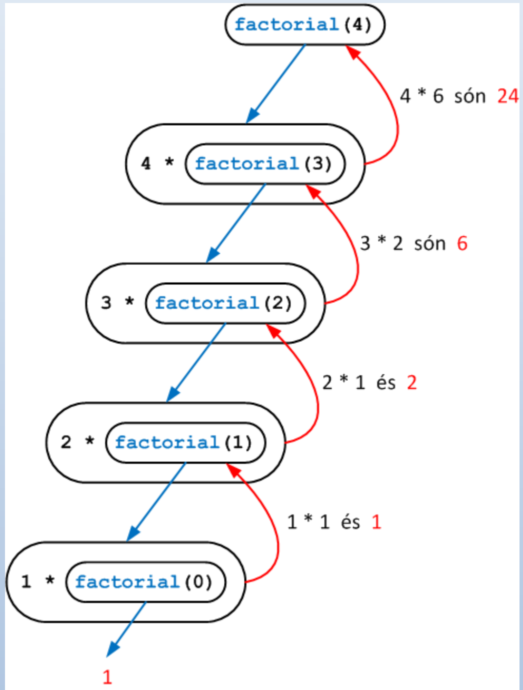
4. Validació del disseny.

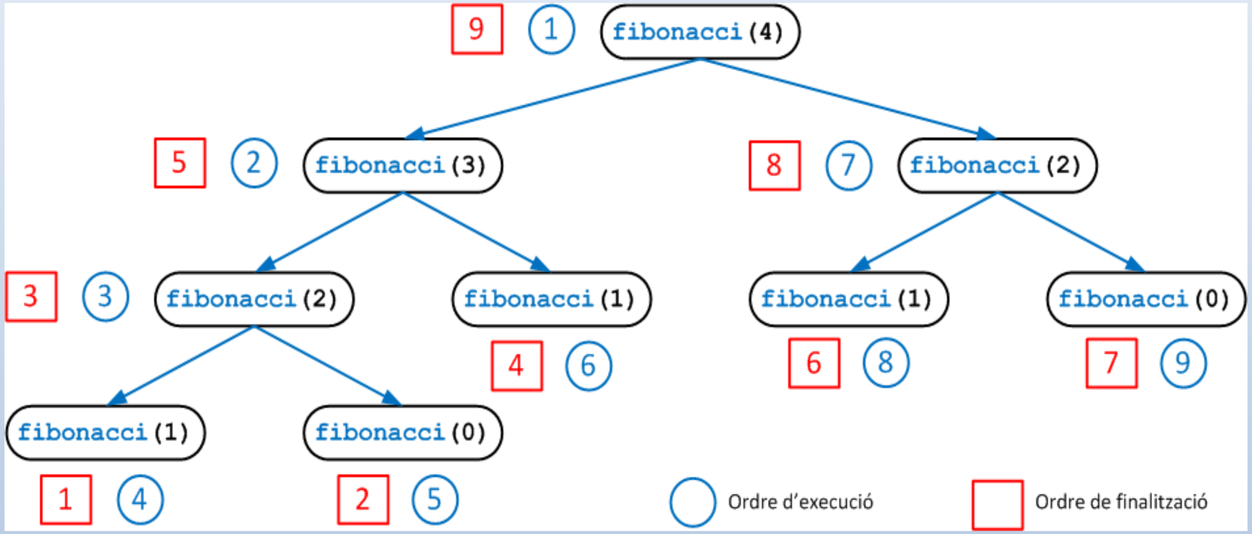
En cada crida recursiva es compleix que:
- El nou problema és estrictament menor (més proper al cas base que el original)
- Els paràmetres de la crida segueixen la precondició.

Tipus de recursió

- Recursió Lineal: una sola crida recursiva en cada cas recursiu.
  - Final: el resultat de la crida recursiva és el resultat del problema.
  - No final: el resultat de la crida recursiva s'utilitza per calcular el resultat del problema però no es directament la solució.
- Recursió Múltiple: més d'una crida recursiva en cada cas recursiu
  - els seus resultats s'han de combinar per obtenir la solució del cas actual. Es genera un arbre de crides en lloc d'una seqüencia.

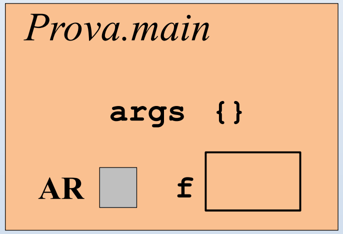
Exemples de lineal y múltiple





Recursivitat i pila de crides

- Cada invocació a un mètode (inclós el main) genera un registre d'activació que conté:
  - Una copia dels paràmetres del mètode com variables locals
  - Les variables locals definides dintre del mètode
  - El resultat del mètode
  - L'adreça de retorn, on anirà el programa en acabar-se.

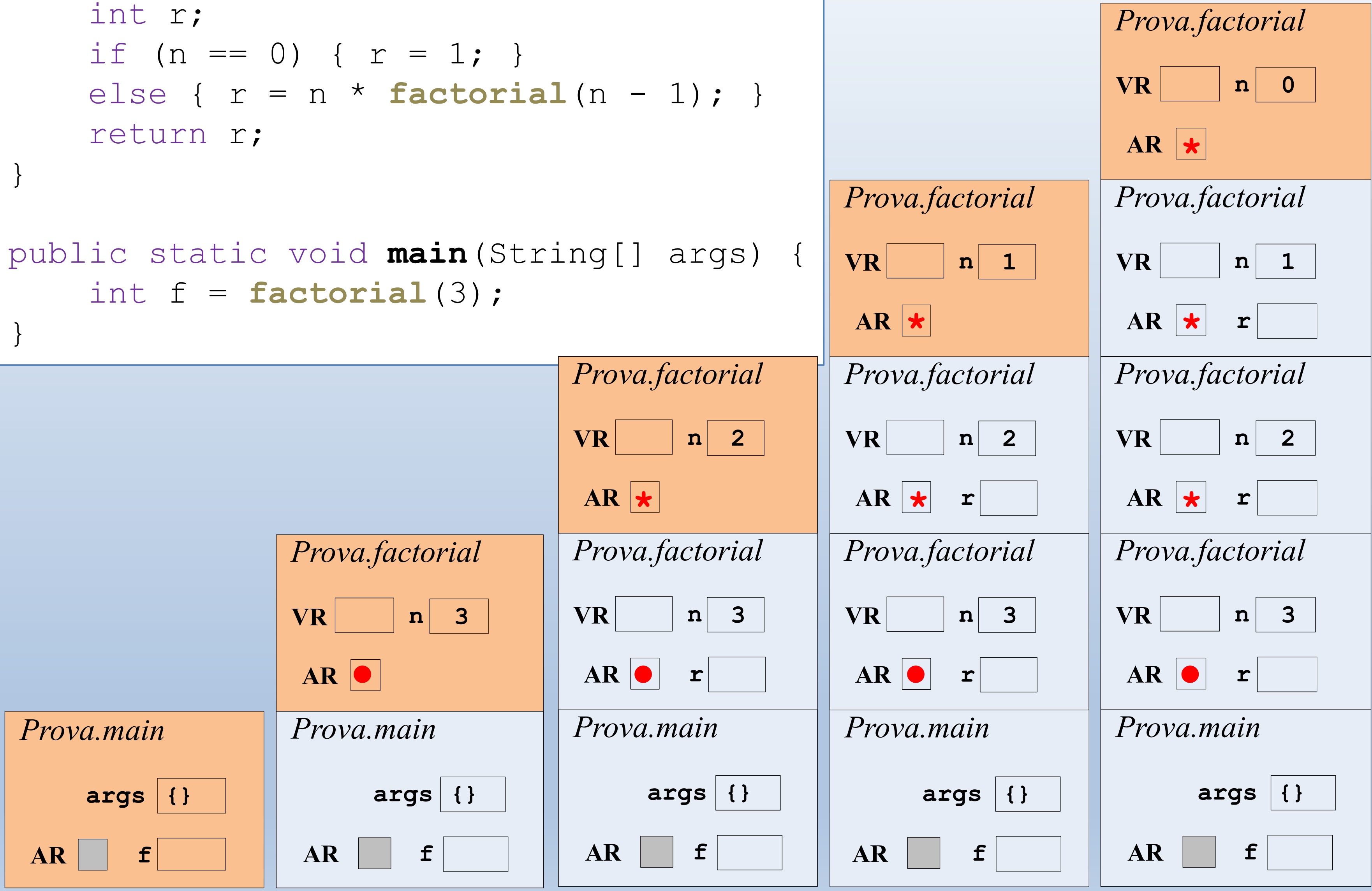


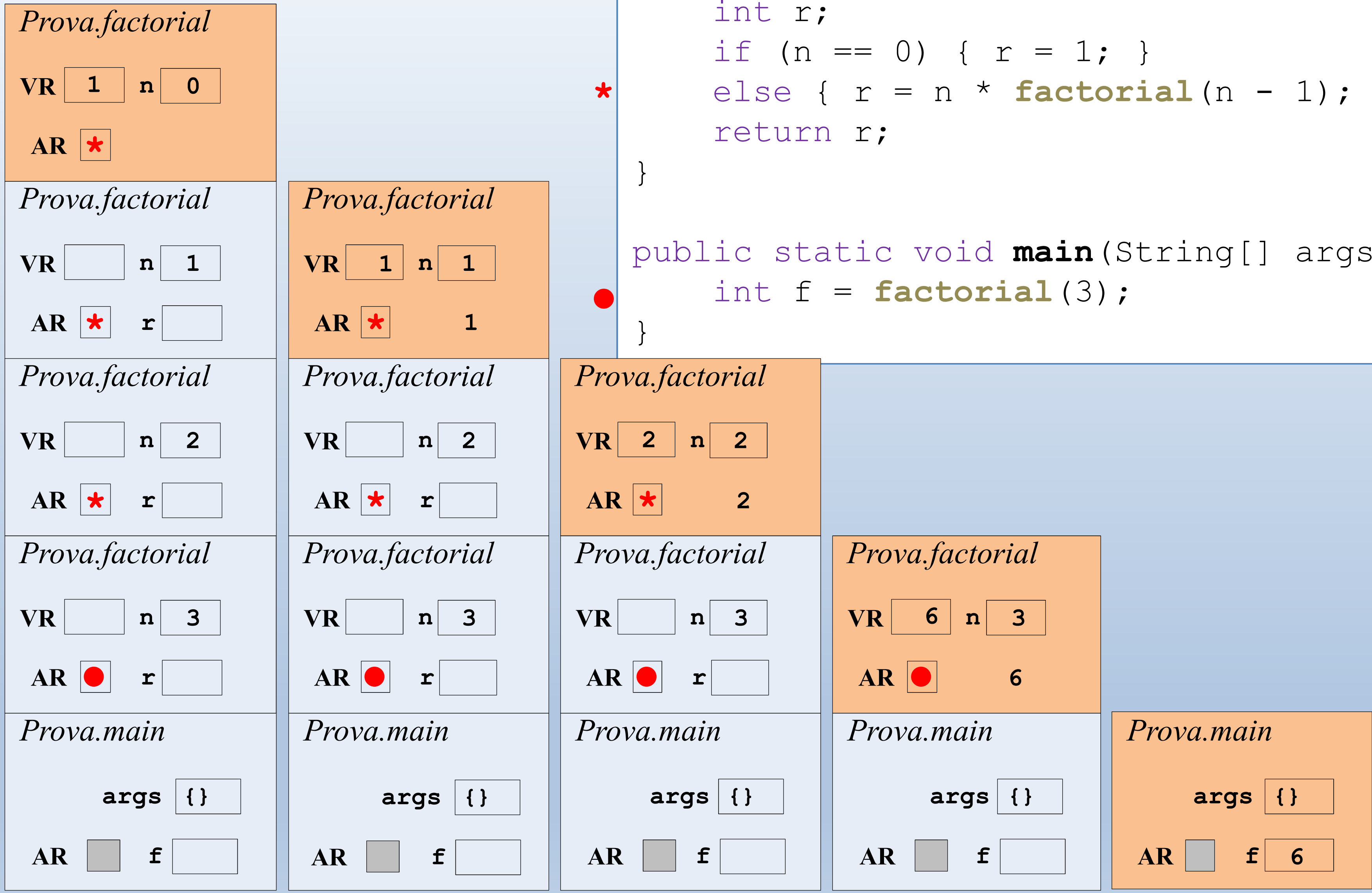
- Cada crida recursiva a un mateix mètode te el seu registre d'activació propi que s'empila a la pila de crides.
- Hi ha tants registres d'activació com crides pendents.
- De tots ells, només està actiu el que està en el cim de la pila.
- Quan acaba mètode (`return`), el seu registre d'activació es desempila.
- En eixe cas s'activa el registre que queda més amunt en la pila.
- La recursiu pot esgotar la memòria, produint-se desbordament de la pila.
- Una causa del desbordament de la pila és la recursió infinita,

StackOverflowError

```java
/** PRECONDICIÓ: n>=0 */
public static int factorial(int n) {
    int r;
    if (n == 0) { r = 1; }
    else { r = n * factorial(n - 1); }
    return r;
}

public static void main(String[] args) {
    int f = factorial(3);
}
```






```
/** PRECONDICIÓ: n>=0 */
public static int factorial(int n) {
    int r;
    if (n == 0) { r = 1; }
*   else { r = n * factorial(n - 1); }
    return r;
}

public static void main(String[] args) {
●   int f = factorial(3);
}
```

Recursivitat i pila de crides

- L'ús de la pila de la crida `factorial(n)` és més gran en el cas recursiu que en l'iteratiu.

- En la versió iterativa de `factorial` coexisteixen simultàniament en memòria, el registre d'activació de `factorial` i el del `main`.

- En la versió recursiva de `factorial` poden arribar a coexistir simultàniament n+1 registres d'activació de `factorial` més el del `main`.

- El consum de memòria de `factorial` iteratiu és sempre el mateix mentre que el del `factorial` recursiu depèn de n.

Exercici... ¿Qué fa aquest mètode?

```
public class Raro {

    /** precondició n >= 0. */
    public static void escribeRaro(int n) {
        if (n > 0) {
            System.out.print(n);
            escribeRaro (n-1 );
            System.out.print(n);
        } else { System.out.print(0); }
    }

    public static void main(String[] args ) {
        escribeRaro(5);
}
```

- Executa el mètode `main`.
- Que mostra por pantalla?
- Que fa si ho executes passant com argument un negatiu?
- Si canviem la condició del `if` per `n != 0`: es correcte l'algorisme?

Alguns exemples – Potència n-èsima

1. Enunciat del problema.

   Donat un número natural $n \geq 0$ i un número real $a \neq 0$, calcular la potència a^n.

```
/** PRECONDICIÓ: n>=0 i a!=0 */
public static double potencia(double a, int n)
```

2. Anàlisi de casos.
   - Cas base: Si $n=0$, aleshores $a^n = 1$.
   - Cas general: Si $n>0$, aleshores $a^n = a^{n-1} * a$.
3. Transcripció de l'algorisme a un mètode en Java.

```
/** PRECONDICIÓ: n>=0 i a!=0 */
public static double potencia(double a, int n) {
    if (n == 0) { return 1; }
    else { return potencia(a, n - 1) * a; }
}
```

recursió lineal no final

4. Validació del disseny.
   - En el cas general, en cada crida el valor del segon paràmetre decreix en 1, així, en algun moment arribarà a ser 0, aconseguint el cas base i finalitzant l'algorisme.
   - A més, sempre es compleix que el valor de $n \geq 0$ i $a \neq 0$.

Alguns exemples – Residu de la divisió entera

1. Enunciat del problema.

   Donats dos números naturals a≥0 i b>0, calcular el residu de la seua divisió entera a/b.

```
/** PRECONDICIÓ: a>=0 i b>0 */
public static int residu(int a, int b)
```

2. Anàlisi de casos.
   - Cas base: Si a<b, aleshores el residu de a/b és a.
   - Cas general: En un altre cas, el residu de a/b és el residu de (a-b)/b.

3. Transcripció de l'algorisme a un mètode en Java.

```
/** PRECONDICIÓ: a>=0 i b>0 */
public static int residu(int a, int b) {
    if (a < b) { return a; }
    else { return residu(a - b, b); }
}
```

recursió lineal final

4. Validació del disseny.
   – En el cas general, en cada crida el valor del primer paràmetre va decreixent, en algun moment serà inferior al segon paràmetre, aconseguint el cas base i finalitzant l'algorisme.
   – A més, sempre es compleix que a≥0 i b>0.

Alguns exemples – Successió de Fibonacci

1. Enunciat del problema.

   Calcular el terme n-èsim de la successió de Fibonacci (el terme 0-èsim és el primer).

```
/** PRECONDICIÓ: n>=0 */
public static int fibonacci(int n)
```

2. Anàlisi de casos.
   - Cas base: Si n≤1, el valor del terme n-èsim és n.
   - Cas general: En un altre cas, és la suma dels termes `(n-1)`-èsim i `(n-2)`-èsim.

3. Transcripció de l'algorisme a un mètode en Java.

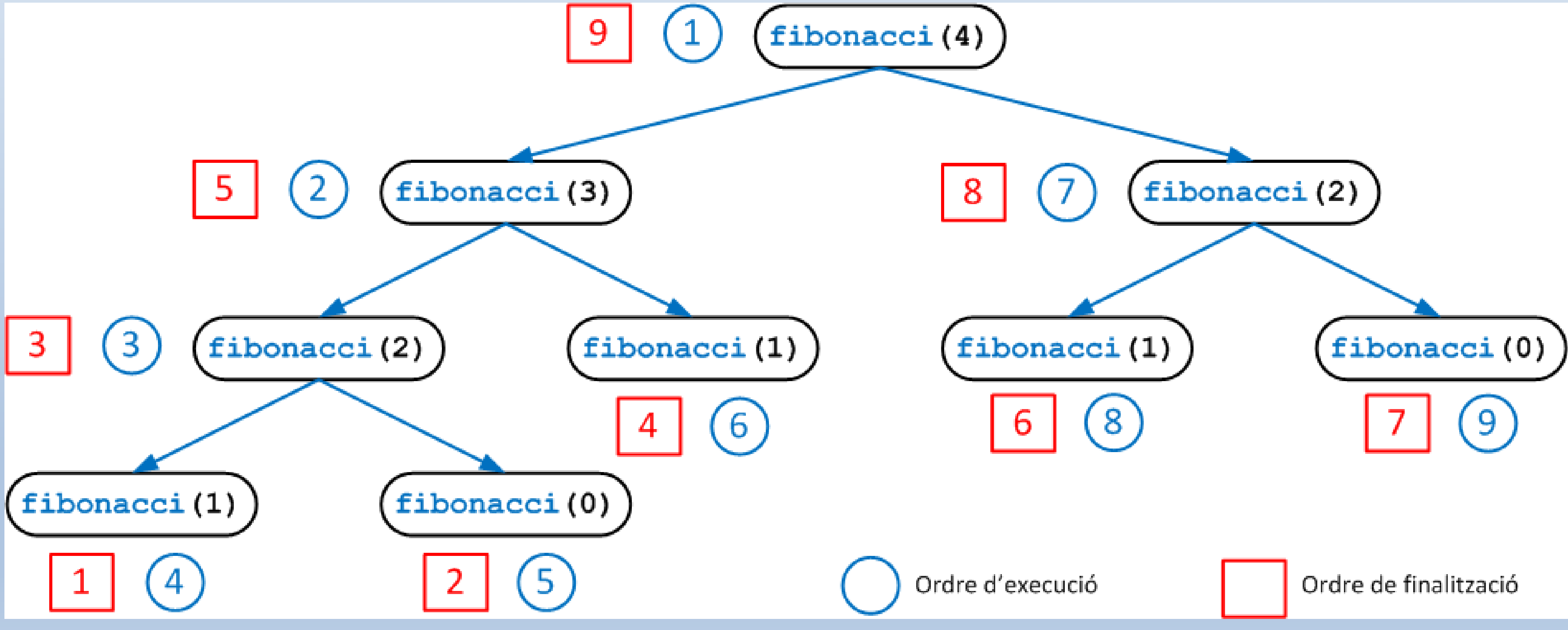
```
/** PRECONDICIÓ: n>=0 */
public static int fibonacci(int n) {
    if (n <= 1) { return n; }
    else { return fibonacci(n - 1) + fibonacci(n - 2); }
}
```

recursió múltiple

4. Validació del disseny.
   - En el cas general, en cada crida es va tendint a termes inferiors de la successió, en algun moment es complirà la condició del cas base, finalitzant l'algorisme.
   - A més, sempre es compleix que n>=0.

Alguns exemples – Successió de Fibonacci

- El mètode `fibonacci` és un mètode recursiu múltiple (en el cas general es realitzen dues crides recursives i el resultat es construeix a partir de la suma dels resultats de les dues crides).
- Per a `fibonacci(4)` es genera l'arbre de crides de la figura.



Alguns exemples – Algorisme d'Euclides I

1. Enunciat del problema.

   Donats dos números naturals a>0 i b>0, calcular el màxim comú divisor (l'algorisme d'Euclides (1))

```
/** PRECONDICIÓ: a>0 i b>0 */
public static int mcd(int a, int b)
```

2. Anàlisi de casos.
   - Cas base: Si a=b, el m.c.d. és b.
   - Cas general: Si a>b, el m.c.d. és el m.c.d. de a–b i b.

     Si a<b, el m.c.d. és el m.c.d. de a i b–a.

3. Transcripció de l'algorisme a un mètode en Java.

```
/** PRECONDICIÓ: a>0 i b>0 */
public static int mcd(int a, int b) {
    if (a == b) { return b; }
    else if (a > b) { return mcd(a - b, b); }
    else { return mcd(a, b - a); }
}
```

recursió lineal final

4. Validació del disseny.
   - En el cas general, en cada crida el valor de un dels paràmetres va decreixent, per tant arribaran a ser iguals i, en aquest cas, s'arribarà al cas base, finalitzant l'algorisme.
   - A més, sempre es compleix que a>0 i b>0.

Alguns exemples – Algorisme d'Euclides II

1. Enunciat del problema.

Donats dos números naturals a>0 i b>0, calcular el màxim comú divisor (l'algorisme d'Euclides (2))

```
/** PRECONDICIÓ: a>0 i b>0 */
public static int euclides(int a, int b)
```

2. Anàlisi de casos.

– Cas base: Si el residu de a/b és 0, el m.c.d. és b.
– Cas general: En un altre cas, el m.c.d. és el m.c.d. de b i el residu de a/b.

3. Transcripció de l'algorisme a un mètode en Java.

```
/** PRECONDICIÓ: a>0 i b>0 */
public static int euclides(int a, int b) {
    if (a % b == 0) { return b; }
    else { return euclides(b, a % b); }
}
```

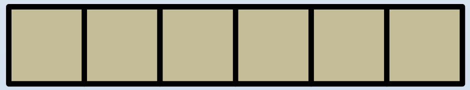
recursió lineal final

4. Validació del disseny.

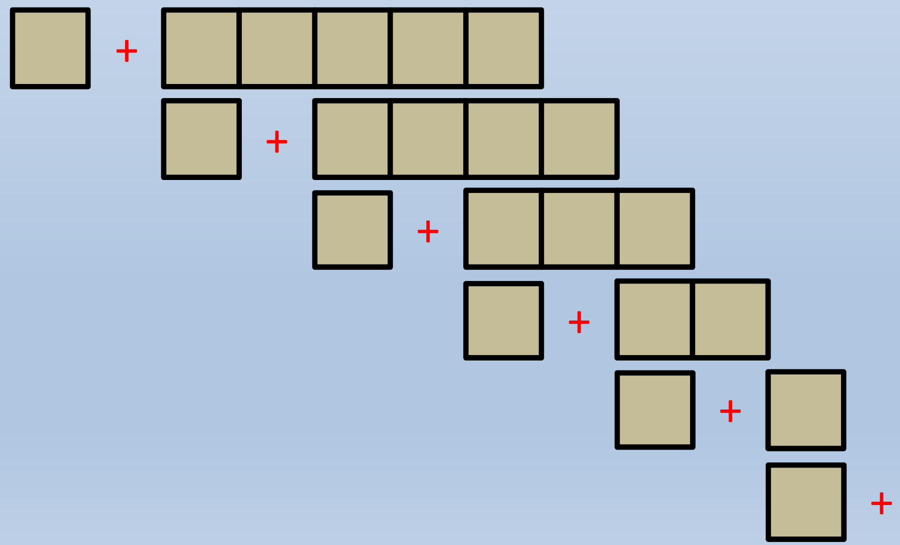
– En el cas general, en cada crida el valor del segon paràmetre va decreixent; per tant arribarà a ser el m.c.d. dels valors originals i, en aquest cas, s'arribarà al cas base, finalitzant l'algorisme.
– A més, sempre es compleix que a>0 i b>0.

Recursió amb arrays

Un array es una seqüencia homogènia de elements accessibles per un índex ... per això:

- Donada la declaració d'un array a de num elements de tipus `tipusBase`:
  `tipusBase[] a = new tipusBase[num];`
  

- Aquesta seqüència es pot definir recursivament com la seqüència formada pel primer element i la subseqüència (subarray) definida per la resta de components de a.



Recursió amb arrays

Recursió amb *arrays*: La grandària del problema -> paràmetres,

- Però... si el paràmetre d'entrada es un *array*, i els *arrays* son de grandària constant
- com podem reduir la grandària del problema?

➡ Afegint paràmetres amb els límits.

El recorregut i la cerca d'un array de manera recursiva es prou semblant a les versions iteratives, les diferències més obvies son, en general:

- Paràmetres afegits (límits de la part tractada i la no tractada)
- Absència de instrucció de iteració (bucle) i Aparició de instrucció condicional (el cas base / cas recursiu)
- En les cerques: segona instrucció condicional per a parar la recursió en cas de trobar-ho. (*no en la mateixa condició que abans)*
- Normalment hi ha que especificar quina es la crida inicial per a que l'algorisme siga equivalent al iteratiu
- MOLT IMPORTANT: En el cas de la cerca **NOMÉS FER LA CRIDA RECURSIVA SI NO S'HA TROBAT i, òbviament, no fer-la més vegades de les necessàries.**

Recursió amb arrays: recorregut

- En un recorregut d'un array a s'han de tractar tots els elements de `a[ini..fi]`, amb `ini` i `fi` dins d'un determinat rang de `a`. Per tant, la seua resolució recursiva sol contemplar la següent anàlisi de casos complementaris i excloents:

  - Cas base: `a[ini..fi]` la grandària és prou menuda com per a que la solució siga directa.

    `ini == fi` o `ini > fi` o `ini >= fi`

  - Cas general: `a[ini..fi]` la grandària no es prou menuda; cal tractar un element i fer un *subrecorregut* sobre una part de l'array més menuda, amb menys components:
    - `tractar(a[ini])` i fer recursió amb `a[ini+1..fi]` (recorregut ascendent),
    - `tractar(a[fi])` i fer recursió amb `a[ini..fi-1]` (descendent),
    - `tractar(a[fi]),tractar(a[fi])` i fer recursió amb `a[ini+1..fi-1]` (combinat),
    - i altres variants específiques de cada problema.

Els índex que canvien amb les crides hauran de ser paràmetres del mètode

`tractar( xxx )` és l'operació a realitzar amb l'element que ocupa l'element `xxx` de l'array.

Els índex sempre han de complir la precondició (estar dins del rang corresponent).

Recorregut recursiu en arrays

```java
/** Precondició a.length > 0
  * Torna la posició del màxim de l'array
  */
public static int posMaxIteratiu(double[] a) {

    int posMax = 0;

    for (int i = 1; i < a.length; i++) {

        if (a[i] > a[posMax]) {

            posMax = i;

        }
    }

    return posMax;

}
```

Es compara la posició i amb la del màxim des de 0 fins a i-1

```java
/** Precondició a.length > 0 ini <= a.length
  * Torna la posició del màxim de l'array des de
  * la posició ini endavant
  *      crida inicial posMaxRecursiu(a, 0);
  */
public static int posMaxRecursiu(double[] a, int ini) {

    int posMax;

    if (ini == a.length -1) { posMax = ini; }

    else {

        posMax = posMaxRecursiu(a, ini + 1);

        if (a[ini] > a[posMax]) {posMax = ini; }

    }
    return posMax;
}
```

Es compara la posició ini amb la del màxim des de ini + 1 fins a a.length

Alguns exemples - Recorregut

Recorregut recursiu ascendent

1. Enunciat del problema.

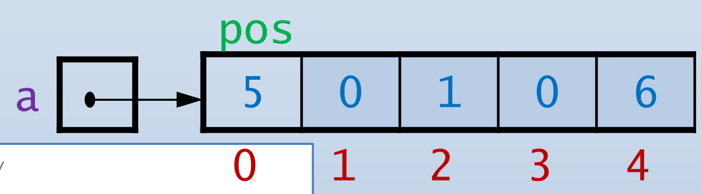
Determinar la suma de tots els elements d'un array d'enters `a[0..a.length-1]` (`a.length` ≥ `0`).

```
/** PRECONDICIÓ: 0 <= pos <= a.length */
public static int sumaRecAsc(int[] a, int pos)
```

2. Anàlisi de casos.

- **Cas base**: Si `pos == a.length`, aleshores la suma de 0 elements és 0.
- **Cas general**: Si no, la suma serà la suma de `a[pos]` i la de `a[pos+1..a.length-1]`.

3. Transcripció de l'algorisme a un mètode en Java.




```
/** PRECONDICIÓ: 0 <= pos <= a.length */
public static int sumaRecAsc(int[] a, int pos) {
    if (pos == a.length) { return 0; }
    else { return a[pos] + sumaRecAsc(a, pos + 1); }
}
```

- La crida inicial ha de ser: `int suma = sumaRecAsc(a, 0);`

4. Validació del disseny.

- En el cas general, en cada crida la talla del problema decreix en 1 perquè pos s'incrementa en 1; així, en algún moment pos arribarà a ser `a.length`, aconseguint el cas base i finalitzant l'algorisme.
- En qualsevol dels dos casos, el valor de pos sempre compleix la precondició.

Alguns exemples - Recorregut

Recorregut recursiu descendent

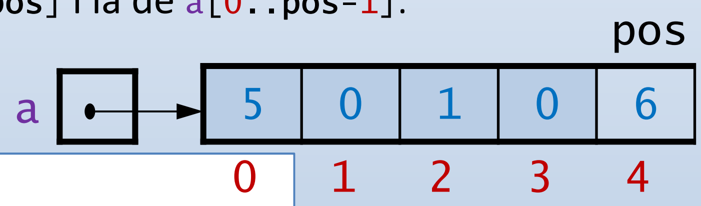
1. Enunciat del problema.

   Determinar la suma de tots els elements d'un array d'enters `a[0..a.length-1]` (`a.length` ≥ `0`).

```
/** PRECONDICIÓ: -1 <= pos < a.length */
public static int sumaRecAsc(int[] a, int pos)
```

2. Anàlisi de casos.
   - Cas base: Si `pos == -1`, aleshores la suma de 0 elements és 0.
   - Cas general: En un altre cas, la suma serà la suma de `a[pos]` i la de `a[0..pos-1]`.

3. Transcripció de l'algorisme a un mètode en Java.




```
/** PRECONDICIÓ: -1 <= pos < a.length */
public static int sumaRecDesc(int[] a, int pos) {
    if (pos == -1) { return 0; }
    else { return a[pos] + sumaRecDesc(a, pos - 1); }
}
```

- La crida inicial ha de ser: `int suma = sumaRecDesc(a, a.length - 1);`

1. Validació del disseny.
   - En el cas general, en cada crida la talla del problema decreix en 1 perquè **pos** es decrementa en 1; per tant **pos** arribarà a ser -1, aconseguint el cas base i finalitzant l'algorisme.
   - En qualsevol dels dos casos, el valor de **pos** sempre compleix la precondició.

Alguns exemples - Recorregut

1. Enunciat del problema.

Recorregut recursiu ascendent

Comptar les aparicions d'un enter `x` en un array d'enters `a[0..a.length-1]` (`a.length` ≥ `0`).

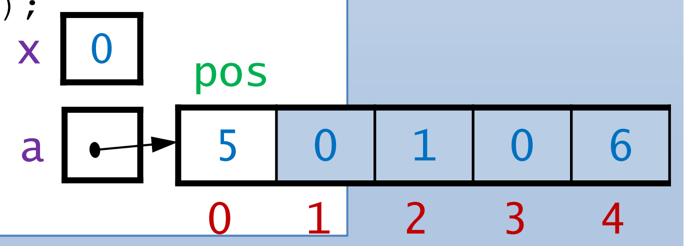
```
/** PRECONDICIÓ: 0 <= pos <= a.length */
public static int comptarRecAsc(int[] a, int x, int pos)
```

2. Anàlisi de casos.

- Cas base: Si **`pos`** `== a.length`, aleshores no hi ha elements i torna 0.
- Cas general: En un altre cas, a les aparicions de `x` en `a[`**`pos`**`+1 .. a.length-1]` sumar `1` si `a[`**`pos`**`]==x`

3. Transcripció de l'algorisme a un mètode en Java.

```
/** PRECONDICIÓ: 0 <= pos <= a.length */
public static int comptarRecAsc(int[] a, int x, int pos) {
    if (pos == a.length) { return 0; }
    else {
        int cont = comptarRecAsc(a, x, pos + 1);
        if (a[pos] != x) { return cont; }
        else { return 1 + cont; }
    }
}
```



- La crida inicial ha de ser: `int num = comptarRecAsc(a, x, 0);`

4. Validació del disseny. Similar a la validació del recorregut ascendent anterior.

Alguns exemples - Recorregut

Recorregut recursiu descendent

1. Enunciat del problema.

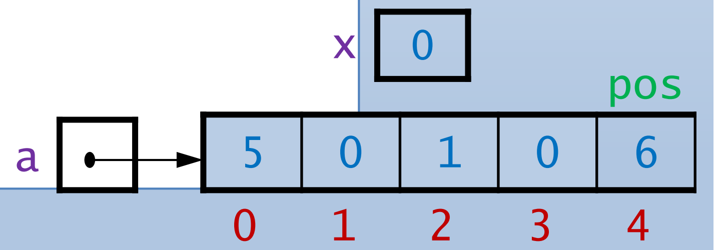
Comptar les aparicions d'un enter `x` en un array d'enters `a[0..a.length-1]` (`a.length` ≥ `0`).

```
/** PRECONDICIÓ: -1 <= pos < a.length */
public static int comptarRecDesc(int[] a, int x, int pos)
```

2. Anàlisi de casos.
   - Cas base: Si **`pos`** `== -1`, aleshores no hi ha elements i torna `0`.
   - Cas general: En un altre cas, a les aparicions de `x` en `a[0..pos-1]` sumar `1` si `a[`**`pos`**`]==x`

3. Transcripció de l'algorisme a un mètode en Java.

```
/** PRECONDICIÓ: -1 <= pos < a.length */
public static int comptarRecDesc(int[] a, int x, int pos) {
    if (pos == -1) { return 0; }
    else {
        int cont = comptarRecDesc(a, x, pos - 1);
        if (a[pos] != x) { return cont; }
        else { return 1 + cont; }
    }
}
```



- La crida inicial ha de ser: `int num = comptarRecDesc(a, x, a.length-1);`

4. Validació del disseny. Similar a la validació del recorregut descendent anterior.

Recursió amb arrays: cerca lineal o seqüencial

- En una cerca lineal recursiva per determinar si es compleix certa propietat en un array `a[ini..fi]`, amb `ini` i `fi` dins d'un determinat rang de `a`, igual que en un recorregut recursiu, la zona de l'array on fer la cerca es va reduint a subarrays més menuts.

- L'anàlisi de casos serà la següent (de manera genèrica):
  - Cas base: subarray reduït al més menut possible on la cerca no pot tindre èxit.
  - Cas general: queden elements suficients per comprovar si es compleix o no la propietat a cercar, de manera que:
    - cerca ascendent,
      - si `propietat(a[ini])` no és veritat, cercar en `a[ini+1..fi]`,
      - si sí és veritat, la cerca finalitza.
    - cerca descendent,
      - si `propietat(a[fi])` no és veritat, cercar en `a[ini..fi-1]`,
      - si sí és veritat, la cerca finalitza.
    - i altres variants específiques de cada problema.

Els índex que canvien amb les crides hauran de ser paràmetres del mètode

On `propietat(a[x])` es veritat si l'element que ocupa la posició `x` de l'array compleix la propietat.

Recursió amb arrays: cerca lineal o seqüencial

- Igual que en els recorreguts, es sol definir un mètode públic homònim, anomenat **guia** o **llançadora**, que realitza la crida inicial, per tal d'ocultar l'estructura recursiva de l'array a que mostren els paràmetres `ini` i `fi` de la capçalera del mètode recursiu que ara es defineix privat.
- Per exemple, en el cas d'una cerca recursiva ascendent que determina si algun element d'un array a compleix certa propietat:

```
/** PRECONDICIÓ: 0 <= ini <= a.length */
private static boolean cercar(tipusBase[] a, int ini) {
    if (ini < a.length) {
        if (propietat(a[ini])) { return true; }
        else { return cercar(a, ini + 1); }
    }
    else { return false; }
}
```

```
public static boolean cercar(tipusBase[] a) {
    return cercar(a, 0);
}
```

- Crida inicial: `boolean res = cercar(a);`

Cerca recursiva en arrays

```java
/** Precondició no hi ha
  *  torna la posició del primer negatiu
  *     i si no hi ha torna  -1
  */
public static int posNegIteratiu(double[] a) {

    int i = 0;

    while (i < a.length && a[i] >= 0) {

        i++;

    }
    if (i == a.length) {

        i = -1;

    }

    return i;

}
```

En acabant si hem arribat a la fi,
no ho hem trobat i tornem -1

```java
/** Precondició a.length > 0 ini <= a.length
  *  torna la posició del primer negatiu des de la
  *   posició ini endavant i si no hi ha, torna  -1
  *  crida inicial  posNegRecursiu(a, 0)
  */
public static int posNegRecursiu(double[] a, int ini) {

    int pos;

    if (ini == a.length) { pos = -1; }

    else if (a[ini] < 0) { pos = ini; }

    else { pos = posNegRecursiu (a, ini + 1); }

    return pos;
}
```

El cas base es arribar a la fi,
no ho hem trobat i tornem -1

Alguns exemples - Cerca

Cerca lineal recursiva ascendent

1. Enunciat del problema.

Obtenir la posició del primer element distint de zero d'un array d'enters `a[0..a.length-1]` (`a.length ≥ 0`).

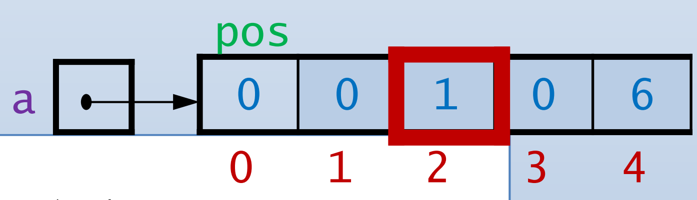
```
/** PRECONDICIÓ: 0 <= pos <= a.length */
public static int trobarRecAsc(int[] a, int pos)
```

2. Anàlisi de casos.

   – Cas base: Si `pos == a.length`, aleshores no es troba i torna -1.

   – Cas general: En un altre cas, bé `a[pos]` és el primer distint de zero i torna `pos` o bé no ho és i la cerca continua en `a[pos+1..a.length-1]`.

3. Transcripció de l'algorisme a un mètode en Java.




```
/** PRECONDICIÓ: 0 <= pos <= a.length */
public static int trobarRecAsc(int[] a, int pos) {
    if (pos == a.length) { return -1; }
    else if (a[pos] != 0) { return pos; }
    else { return trobarRecAsc(a, pos + 1); }
}
```

4. Validació del disseny.

   – En el cas general, en cada crida la grandària del problema decreix en 1 perquè `pos` s'incrementa en 1; així, en algún moment `pos` arribarà a ser `a.length`, aconseguint el cas base i finalitzant l'algorisme.

   – En qualsevol dels dos casos, el valor de `pos` sempre compleix la precondició.

Alguns exemples - Cerca

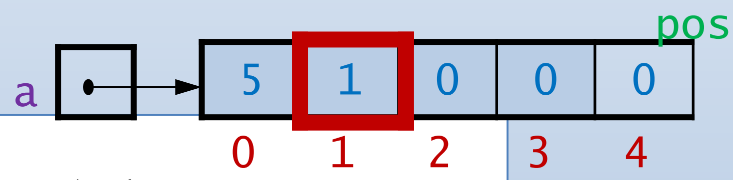
1. Enunciat del problema. Obtenir la posició del darrer element distint de zero d'un array d'enters `a[0..a.length-1]` (`a.length ≥ 0`).

Cerca lineal recursiva descendent

```
/** PRECONDICIÓ: -1 <= pos < a.length */
public static int trobarRecDesc(int[] a, int pos)
```

2. Anàlisi de casos.
   - Cas base: Si `pos == -1`, aleshores no es troba i torna -1.
   - Cas general: En un altre cas, bé `a[pos]` és el darrer distint de zero i torna `pos` o bé no ho és i la cerca continua en `a[0..pos-1]`.

3. Transcripció de l'algorisme a un mètode en Java.




```
/** PRECONDICIÓ: -1 <= pos < a.length */
public static int trobarRecDesc(int[] a, int pos) {
    if (pos == -1) { return -1; }
    else if (a[pos] != 0) { return pos; }
    else { return trobarRecDesc(a, pos - 1); }
}
```

•

4. Validació del disseny.
   - En el cas general, en cada crida la grandària del problema decreix en 1 perquè pos es decrementa en 1; així, en algún moment pos arribarà a ser -1, aconseguint el cas base i finalitzant l'algorisme.
   - En qualsevol dels dos casos, el valor de pos sempre compleix la precondició.

Cerca binària: estratègia

- S'estableixen dues estratègies principals a l'hora de fer una cerca en un array:
  - Cerca lineal o seqüencial:
    - es va reduint l'espai de cerca element a element.
    - S'ha de realitzar una cerca exhaustiva (des d'un extrem cap a l'altre) , fins que es trobe o s'arribe al final sense trobar-lo).
    - Es pot aplicar sempre, independentment de si les dades estan ordenades o no a l'array.
  - Cerca binària o dicotòmica:
    - es va reduint l'espai de cerca, eliminant cada vegada la meitat d'elements.
    - Les dades de l'array han d'estar ordenades d'una forma coneguda (per exemple, de menor a major), però és més eficient que la cerca lineal.

- Enunciat del problema.

  Obtenir la posició de x en un array d' enters `a[ini..fi]` ordenat ascendentment, `0<=ini<=fi<a.length`.

- Estratègia de l'algorisme:

  examinar la posició central, `meitat = (ini + fi) / 2`, i decidir segons els tres casos possibles:
  - `a[meitat] == x`, aleshores la cerca acaba amb èxit i torna `meitat`.
  - `a[meitat] > x`, aleshores la cerca continua en `a[ini..meitat-1]`.
  - `a[meitat] < x`, aleshores la cerca continua en `a[meitat+1..fi]`.

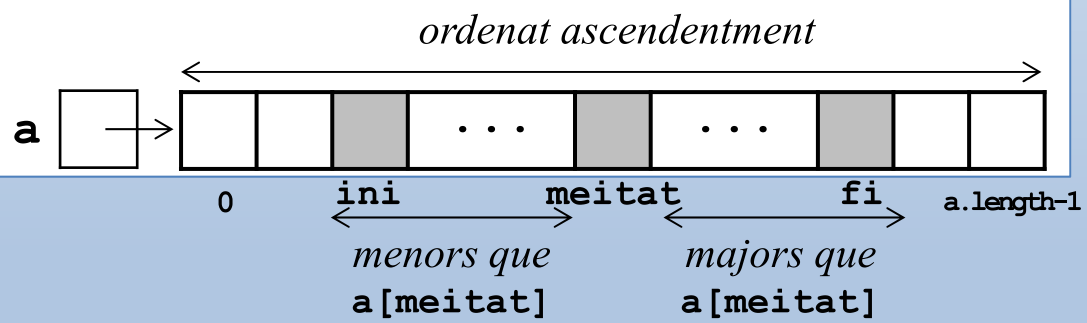
  Si x no es troba, torna -1.

Cerca binària iterativa

1. Enunciat del problema.

Obtenir la posició d'un enter `x` en un array d' enters `a[0..a.length-1]` ordenat ascendentment.

```
/** PRECONDICIÓ: 0 <= ini <= a.length i -1 <= fi < a.length */
public static int cercaBinIter(int[] a, int x, int ini, int fi) {
    int meitat = 0;
    boolean trobat = false;
    while (ini <= fi && !trobat) {
        meitat = (ini + fi) / 2;
        if (x == a[meitat]) { trobat = true; }
        else if (x < a[meitat]) { fi = meitat - 1; }
        else { ini = meitat + 1; }
    }
    if (trobat) { return meitat; }
    else { return -1; }
}
```



- Fica un punt de ruptura en la línia del `while` i observa l'estat de les variables al llarg de l'execució del codi per a l'array `a` = {0, 3, 4, 6, 7, 8, 10, 17}, `ini` = 0, `fi` = 7 i `x` = 20. Quantes iteracions es fan?

Cerca binària recursiva

1. Enunciat del problema.

   Obtenir la posició d'un enter `x` en un array d' enters `a[0..a.length-1]` ordenat ascendentment.

```
/** PRECONDICIÓ: 0 <= ini <= a.length i -1 <= fi < a.length */
public static int cercaBinRec(int[] a, int x, int ini, int fi)
```

2. Anàlisi de casos.
   - Cas base: Si `ini > fi`, aleshores no es troba i torna `-1`.
   - Cas general: Si `ini ≤ fi`, per al subarray `a[ini..fi]`,
     - Determinar la posició de l'element central `meitat = (ini + fi) / 2`.
     - Accedir a l'element central de `a[ini..fi]`, `a[meitat]`, comprovant que si:
       - `a[meitat] == x`, aleshores la cerca acaba amb èxit i torna `meitat`.
       - `a[meitat] > x`, aleshores la cerca continua en `a[ini..meitat-1]`.
       - `a[meitat] < x`, aleshores la cerca continua en `a[meitat+1..fi]`.

Cerca binària recursiva

3. Transcripció de l'algorisme a un mètode en Java.

```
/** PRECONDICIÓ: 0 <= ini <= a.length i -1 <= fi < a.length */
public static int cercaBinRec(int[] a, int x, int ini, int fi) {
    if (ini > fi) { return -1; }
    else {
        int meitat = (ini + fi) / 2;
        if (a[meitat] == x) { return meitat;}
        else if (a[meitat] > x) {
            return cercaBinRec(a, x, ini, meitat - 1);
        }
        else { return cercaBinRec(a, x, meitat + 1, fi); }
    }
}
```

- La crida inicial ha de ser: `int pos = cercaBinRec(a, x, 0, a.length - 1);`

4. Validació del disseny.
   - En el cas general, en cada crida la grandària del problema es divideix per 2 (divisió entera); així, en algún moment `ini > fi`, aconseguint el cas base i finalitzant l'algorisme.
   - En qualsevol dels dos casos, els valors de `ini` i `fi` sempre compleixen la precondició.

- Fica un punt de ruptura en la línia del cas base i en la primera línia del cas general i observa l'estat de les variables al llarg de l'execució del codi per a l'array `a` = {0, 3, 4, 6, 7, 8, 10, 17}, `ini` = 0, `fi` = 7 i `x` = 20. Quantes crides es fan en total?

Recursió versus iteració

- Tant recursió com iteració fan ús d'estructures de control: la recursió fa servir com instrucció principal una instrucció de selecció (condicional) i la iteració fa servir com instrucció principal una instrucció de repetició (bucle).
- Ambdues requereixen una condició de terminació: la condició del cas base en la recursió i la condició de la guarda del bucle en la iteració.
- Ambdues s'aproximen gradualment a la terminació: la iteració a mesura que s'apropa al compliment d'una condició i la recursió conforme es divideix el problema en altres més menuts, apropant-se també al compliment d’una condició, la del cas base.
- Ambdues poden tenir (per error) una execució potencialment infinita.
- Es pot demostrar que la solució algorísmica de qualsevol problema algorísmicament resoluble, es pot expressar recursivament i també iterativament.
- En aquest sentit, es diu que recursió i iteració són equivalents, i per això, alternatius.

Recursió versus iteració

- En general, no és possible afirmar què és el més convenient o senzill.
- És freqüent trobar problemes per als quals la solució iterativa és més senzilla d'estructurar que la recursiva.
- A més, la recursió suposa, en general, més càrrega computacional (espai en memòria) que la iteració.
- En altres casos, la versió recursiva reflecteix de manera més natural, concisa i elegant la solució al problema que la versió iterativa, el que fa que siga més fàcil de depurar i entendre.
- Es pot concloure que recursió i iteració, a mes d'alternatius, són complementaris.