INS MANUEL VÁZQUEZ MONTALBÁN

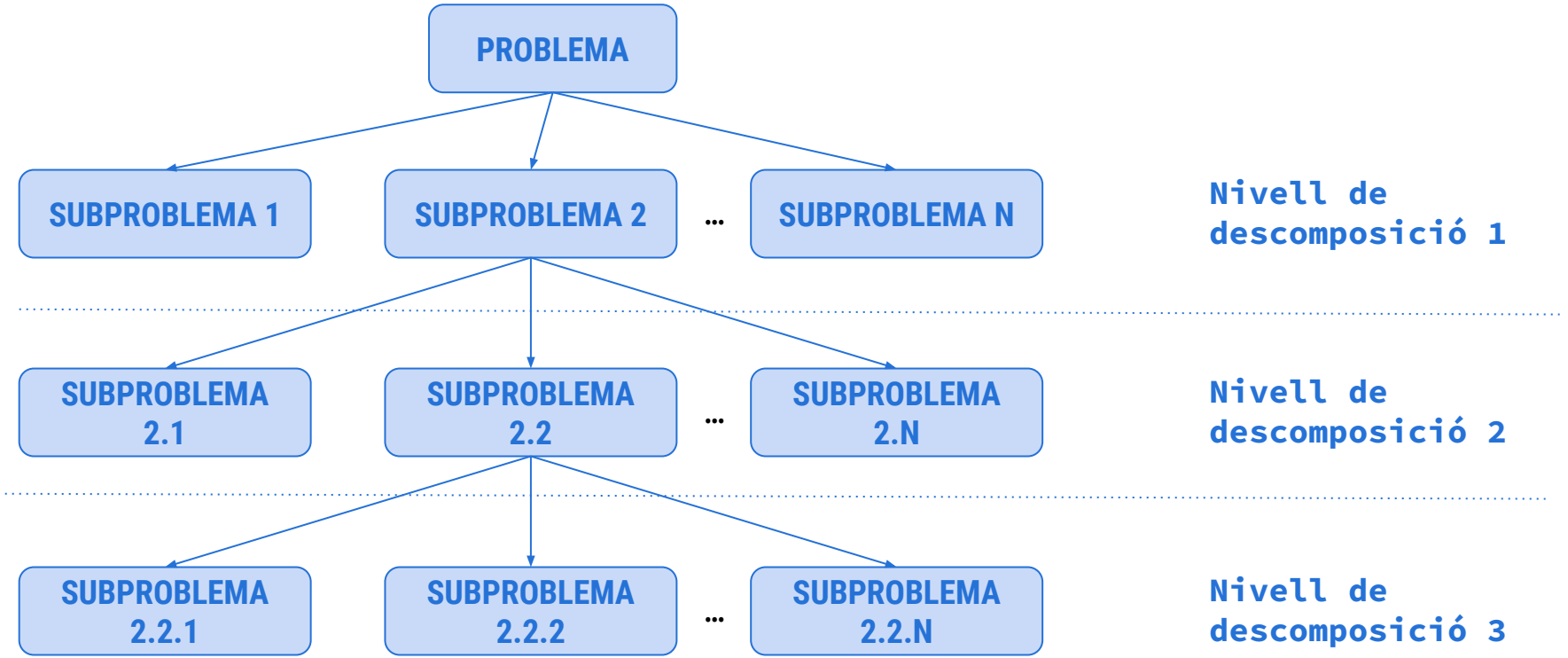
M03. Programació bàsica

UF2. Disseny modular

1. FUNCIONS

Disseny modular. Descomposició del problema

Tal i com vam veure a la UF1, a l'hora de resoldre un problema, apliquem el disseny top-down (descendent).



Què és una funció?

Un **bloc de sentències** que executa una tasca específica i al que fem referència mitjançant un **nom.**

En java, aquesta funcionalitat s'anomena **mètode.**

Què s'ha d'especificar a l'hora d'escriure un mètode?

Nom

El mètode ha de tenir un nom per tal de poder-la cridar

Cos

Conté les operacions (declaracion, condicions, iteracions) que ha d'executar el mètode

Arguments

Paràmetres que ha de rebre (o no) el mètode per tal d'executar les operacions del cos

Retorn

Cal indicar el tipus del resultat que retornarà (si en retorna) o si no retorna res

Implementació del disseny modular

```
function
    var
        integer num1, num2
    endvar
    num1 = validate()
    num2 = validate()
    if num1 > num2 then
       write(“num1 és major”)
    else
       if num1 < num2 then
          write(“num2 és major”)
       else
          write(“Són iguals”)
       endif
    endif
endfunction
```

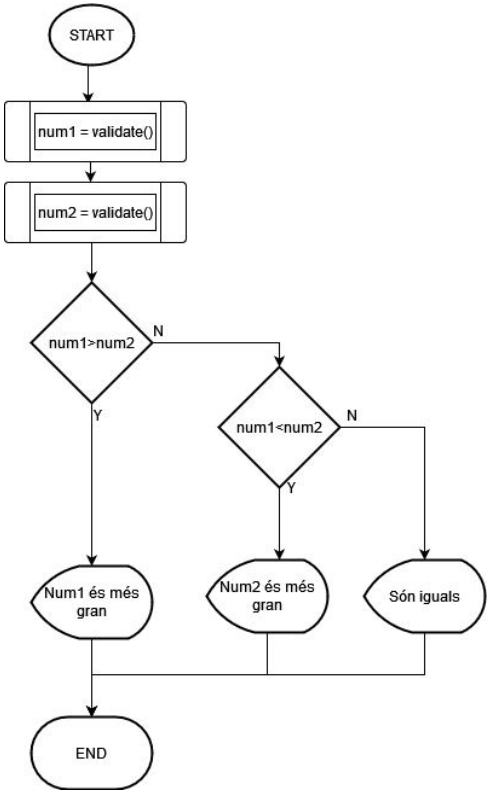
```
function validate()
    var
      integer num
    endvar
    do
      write(“Introdueix un nombre natural”)
      read(num)
    while num<1
    return num
endfunction
```

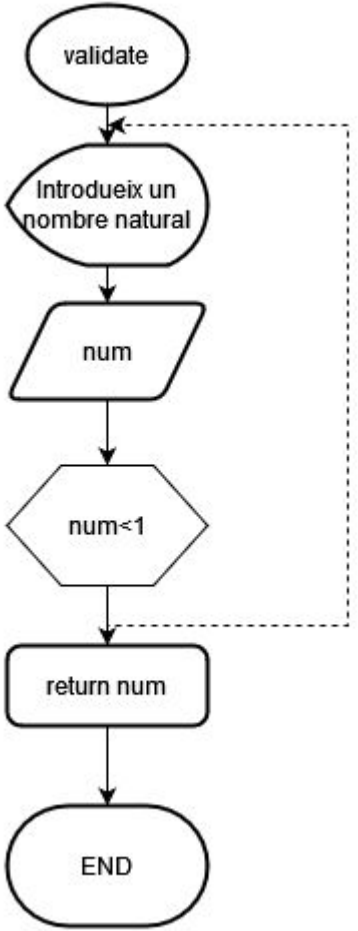
Àmbits d'una variable

Una variable és **global** quan aquesta es declara en un programa fora de qualsevol bloc (és accessible des del seu punt de definició o declaració fins al final del codi font).
És a dir, existeix i té valor des del començament fins al final de l'execució del programa.

Una variable és **local** quan la seva declaració es fa dins d'un bloc (l'accés a aquesta variable queda limitat a aquest bloc i als blocs continguts dins d'aquest per sota del seu punt de declaració).
És a dir, és accessible només dins del bloc al qual pertany.

Implementació del disseny modular





Mètodes en Java

Definició d'un mètode

```
modificador tipus nomMetode (llistaArguments){
    (declaracions de variables locals);
    operacions (amb els paràmetres i variables locals);
    (retorn)
}
```

Les variables declarades en el cos de la funció són **locals** (només són accessibles dins el mètode).
El paràmetre *tipus* indica de quin tipus serà el valor retornat pel mètode (int, float, char, objectes,...)

Definició d'un mètode

`modificador tipus nomMetode (llistaArguments)`

```
public void nomMetode ()
public void nomMetode (int)
public int nomMetode ()
public int nomMetode (int)
```

La **declaració d'un mètode** (**prototipus**) indica:

- el nom del mètode
- quants arguments té i de quin tipus són
- el tipus del valor retornat

Important! Podem declarar un mètode amb el mateix nom i diferent tipus d'arguments (overloading, OOP).

Invocació de mètodes. Exemple

```
public class Metodes {
    public int val1 = 3, val2 = 4;

    public static void main(String[] args) {
        Metodes method = new Metodes();
        System.out.println(method.sumar());
    }

    public int sumar(){
        return val1+val2;
    }
}
```

Invocació de mètodes. Exemple II

```java
//mètode que retorna la suma dels dos arguments
public int sumar(int num1, int num2){
    return num1+num2;
}

//mètode que retorna la suma dels dos valors globals
public int sumar(){
    return val1+val2;
}
```

Invocació de mètodes

Tenint en compte els exemples anteriors, quina de les opcions és millor a l'hora de declarar un mètode?

Cal tenir en compte els següents punts:

- **Àmbit**: els valors amb els que treballaran els mètodes seran locals o globals?
- **Retorn**: el resultat de les operacions executades dins del mètode ha de retornar al programa principal o a un altre mètode o no?

Un cop resolts els punts anteriors, podem defnir els nostres mètodes segons una de les 4 combinacions possibles.

Exercicis I

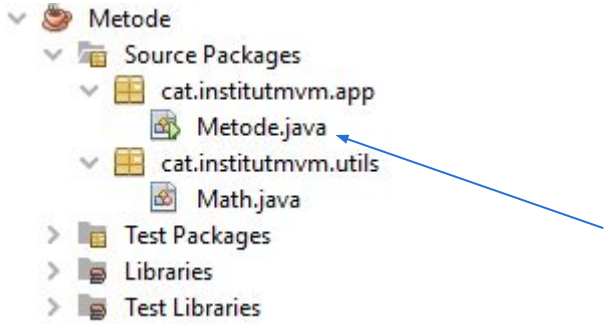
Exercici 1

Crea un programa que demani un nombre per teclat i validi si aquest és natural.

Exercici 2

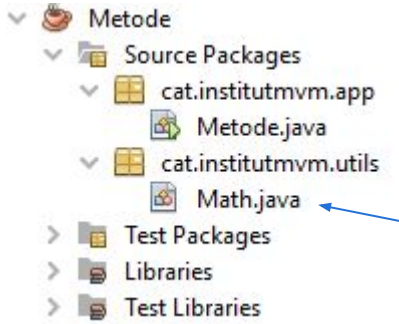
Crea un programa que demani per teclat un el radi d'una circumferència (nombre enter) i retorni la seva longitud.

Estructura del projecte




```java
package cat.institutmvm.app;

import java.util.Scanner;
import cat.institutmvm.utils.Math;

public class Metode {
    //variables globals
    private int val1 = 3, val2 = 4;
    private static final String MSG_1 = "Introdueix dos nombres: ";
    private static final String MSG_2 = "Resultat amb valors globals: ";
    private static final String MSG_3 = "Resultat amb pas de paràmetres: ";

    public static void main(String[] args) {
        int num1, num2;
        Scanner sc = new Scanner(System.in);
        Metode method = new Metode();
        Math mt = new Math();
        System.out.println(MSG_1);
        num1 = sc.nextInt();
        num2 = sc.nextInt();
        System.out.println(MSG_3 + mt.sumar(num1, num2));
        System.out.println(MSG_2 + method.sumar());
    }

    //mètode que retorna la suma dels dos valors globals
    public int sumar(){
        return val1+val2;
    }
}
```

Estructura del projecte




```java
package cat.institutmvm.utils;

public class Math {
        //mètode que retorna la suma dels dos valors globals
    public int sumar(int num1, int num2){
        return num1+num2;
    }
}
```

2. LLIBRERIES

Llibreries (biblioteques)

Les llibreries o biblioteques són una recopilació de rutines que implementen operacions.

Als nostres programes podem incloure les llibreries pròpies del llenguatge (lectura de dades, operacions aritmètiques,...). També podem recopilar els nostres mètodes en llibreries i incloure-les en les nostres aplicacions.

En el cas de Java, aquests mètodes es defineixen dins de classes, que es classifiquen en diferents packages (segons la tipologia dels mètodes). Les classes pròpies de l'arquitectura Java estan dins l'**API de Java**.

Java API

Alguns dels packages més importants són els següents:

- java.lang: conté les classes fonamentals per a la programació en Java
- java.util: conté diferents utilitats per a la conversió de cadenes i lectura de dades i el framework *collections*, entre d'altres.
- java.nio.file: defineix les classes per a accedir a fitxers i fitxers del sistema
- java.time: conté les classes principals per dates, temps i durades

2. Recursivitat

Recursivitat

Recursió és la tècnica consistent a definir una funció en termes de si mateixa.

- a C, Java i Python una funció pot cridar a altres funcions -> una funció també pot cridar-se a si mateixa

S'anomena **recursivitat** a un procés mitjançant el qual una funció es crida a si mateixa de forma repetida, fins que se satisfà alguna determinada condició. El procés s'utilitza per a computacions repetides en les quals cada acció es determina mitjançant un resultat anterior.

Molts problemes recursius es poden resoldre de manera iterativa.

Recursivitat

Igual que les *lleis de la robòtica d'Asimov*, tots els algorismes recursius han d'obeir tres lleis importants:

- Un algorisme recursiu ha de tenir un cas base.
- Un algorisme recursiu ha de cridar-se a si mateix, recursivament.
- Un algorisme recursiu ha de canviar el seu estat i moure's cap al cas base.

La **recursivitat** és una forma de descriure un procés per resoldre un problema de manera que, al llarg d'aquesta descripció, s'invoca el procés mateix que s'està descrivint, però aplicat a un cas més simple.

Recursivitat

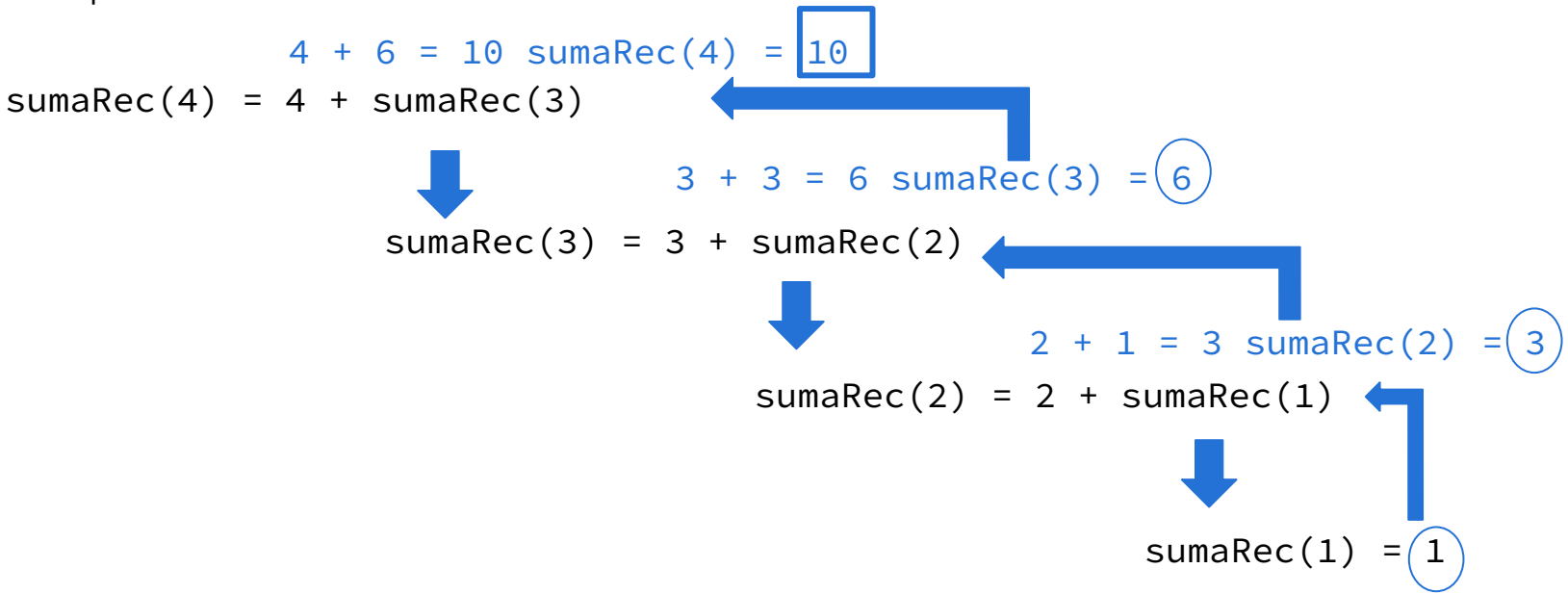
Exemple: calcular la suma de tots els números fins el nombre indicat

```
public int sumaIt(int n){
     int num = 0;
     for(int i=1;i<=n;i++){
          num +=i;
     }
     return num;
}


public static void main(String[] args) {
     Method met = new Method();
     System.out.println(met.sumaIt(4));
     }
}
```

```
public int sumaRec(int n){
     if (n==1){
          return 1;
     }
     else{
          return n + sumaRec(n-1);
     }
}

public static void main(String[] args){
     Method met = new Method();
     System.out.println(met.sumaRec(4));
     }
}
```

Recursivitat

Exemple: calcular la suma de tots els números fins el nombre indicat. Com ho fa?



Recursivitat

Exemple: calcular la potència d'un nombre

```
public int powRec(int m, int n){
     if (n==0){
          return 1;
     }
     else{
          if(n==1){
               return m;
          }
          else{
               return m * powRec(m, n-1);
          }
     }
}
```

```
public static void main(String[] args){
     Method met = new Method();
     System.out.println(met.powRec(2,3));
}
```

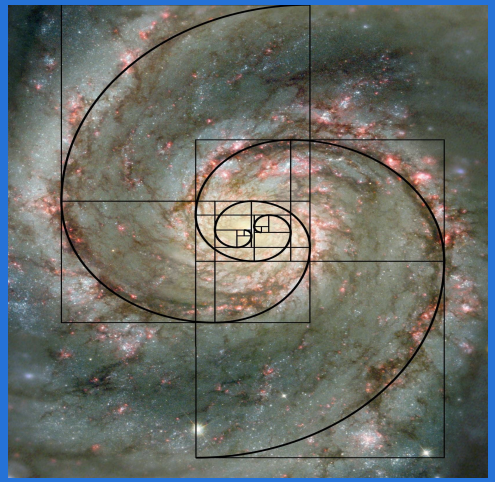
Recursivitat

Exemple: calcular el factorial d'un nombre

```
public int factorialIt(int n){
     int count = 1, val = 1;
     while (val <= n){
          count *= val;
          val++;
     }
     return count;
}

public static void main(String[] args){
     Method met = new Method();
     System.out.println(met.factorialIt(4));
     System.out.println(met.factorialRec(4));
}
```

=

```
public int factorialRec(int n){
     if (n==0 || n==1){
          return 1;
     }
     else{
          return n * factorialRec(n-1);
     }
}
```



La Successió de Fibonacci

Recursivitat. Fibonacci

La successió de Fibonacci és una successió matemàtica de nombres naturals tal que cada un dels seus termes és igual a la suma dels dos anteriors. Descrita per primera vegada per Leonardo de Pisa Fibonacci, cadascun dels seus termes rep el nom de nombre de Fibonacci.
Exemple: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Si es pren una successió de nombres naturals de tal forma que els dos primers termes siguin

F(0) = 0
F(1) = 1

i cadascun dels següents termes és la suma dels dos anteriors:

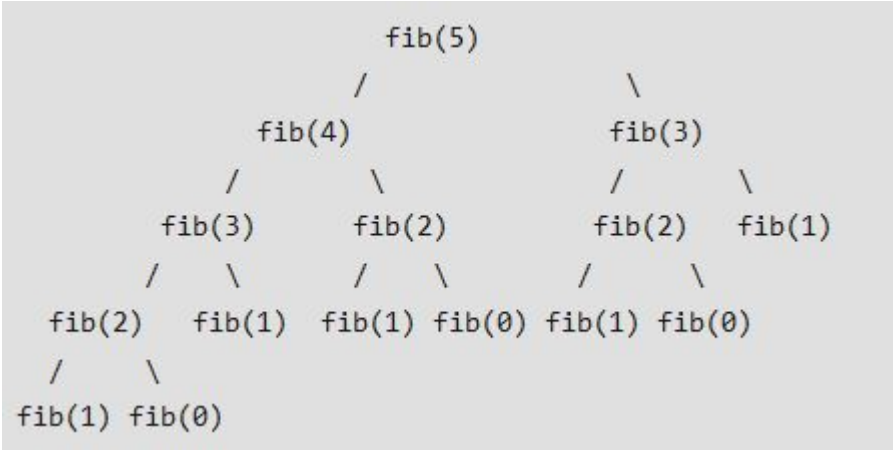
F(n) = F(n-2) + F(n-1)

Aquesta successió és definida per recursivitat com:

$$F(n) = \begin{cases} 0, & \text{si } n = 0; \\ 1, & \text{si } n = 1; \\ F(n-1) + F(n-2) & \text{altrament.} \end{cases}$$

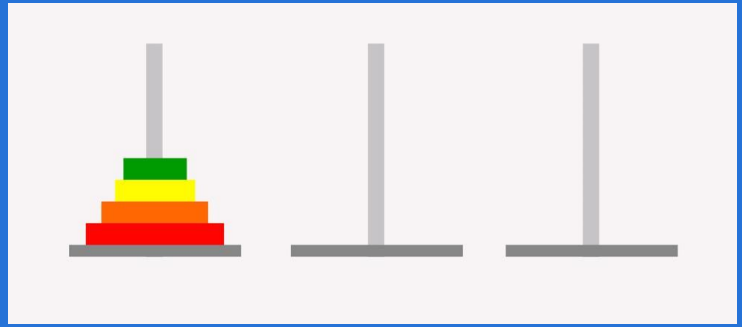
Recursivitat. Fibonacci

Per tal de calcular la successió, hem de calcular els valors dels nombres anteriors (en sentit descendent, top-down).



Recursivitat. Fibonacci

```
public int fibonacci(int num){
     //condició base
     if (num == 0 || num == 1){
          return num;
     }
     else{
           return fibonacci(num - 1) + fibonacci(num - 2);
     }
}
public static void main(String[] args){
    Method met = new Method();
    for (int n = 0; n <= 20; n++){
     System.out.println(met.fibonacci(n));
    }
}
```



Les Torres de Hanoi

Recursivitat. Les Torres de Hanoi

Les **Torres de Hanoi** (també anomenades la Torre de Brahma o la Torre de Lucas, i a vegades pluralizada com Torres) és un joc o trencaclosques matemàtic. Consisteix en tres barres i un nombre de discos de diferents grandàries, que poden lliscar-se en qualsevol barra. El trencaclosques comença amb els discos en una pila ordenada en ordre ascendent de grandària en una vareta, la més petita en la part superior, fent així una forma de piràmide.

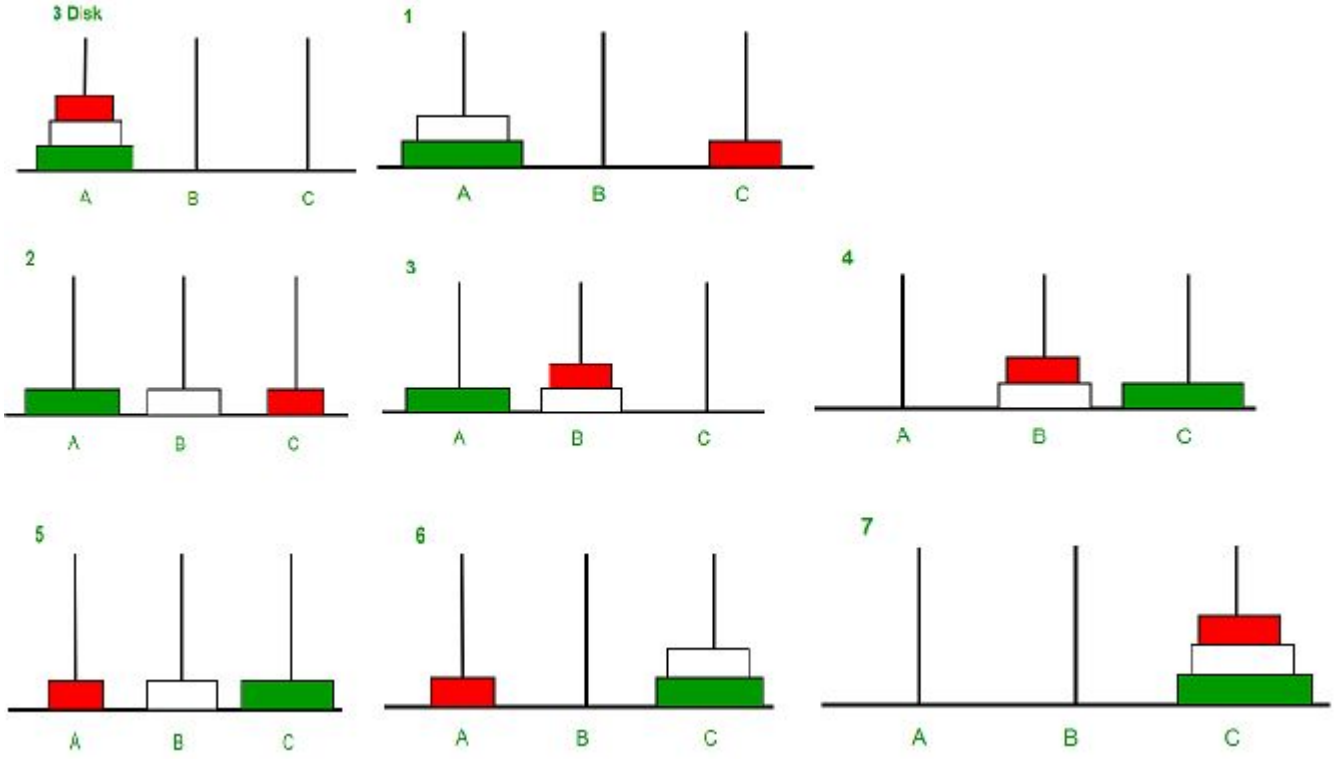
L'objectiu del trencaclosques és moure tota la pila a una altra vareta, obeint les següents regles simples:

- Només es pot moure un disc alhora.
- Cada moviment consisteix a prendre el disc superior d'una de les piles i col·locar-lo sobre una altra pila o sobre una vareta buida.
- No es pot col·locar un disc més gran damunt d'un disc més petit.

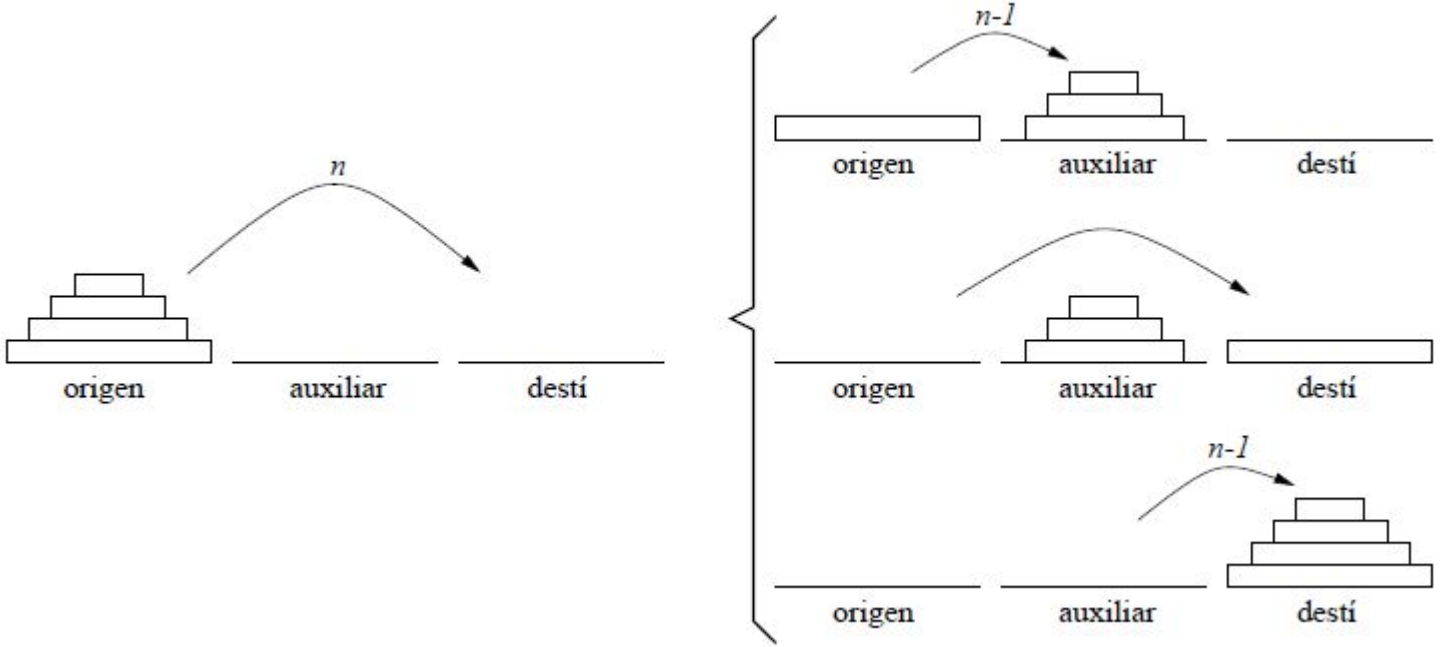
Amb 3 discos, el trencaclosques pot ser resolt en 7 moviments. El nombre mínim de moviments requerits per a resoldre un trencaclosques de la Torre d'Hanoi és $2^{\mathbf{n}-1}$, on **n** és el nombre de discos.

Recursivitat. Les Torres de Hanoi

Recursivitat. Les Torres de Hanoi



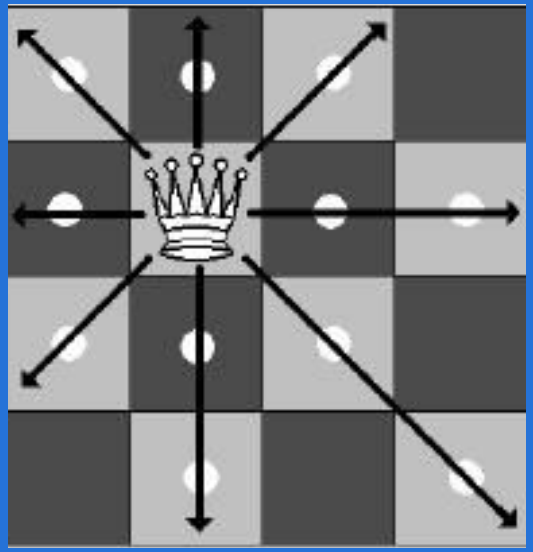
Recursivitat. Les Torres de Hanoi



Recursivitat. Les Torres de Hanoi

```
public void hanoiTower(int n, char from_disc, char to_disc, char aux){
    if(n == 1){
          System.out.format("Mou el disc 1 des de %c fins a %c\n",from_disc, to_disc);
    }

    else{
          hanoiTower(n - 1, from_disc, aux, to_disc);
          System.out.format("Mou el disc %d des de %c fins a %c\n", n, from_disc, to_disc);
          hanoiTower(n - 1, aux, to_disc, from_disc);
    }
}

public static void main(String[] args) {
    int n=3;
    Method met = new Method();
    met.hanoiTower(n, 'A', 'C', 'B');
}
```

Les 8 reines



Recursivitat. Les 8 reines

El problema de les vuit reines consisteix en col·locar en un tauler d'escacs (que té 8x8=64 caselles) vuit reines de forma que cap d'elles amenaci a cap altra.

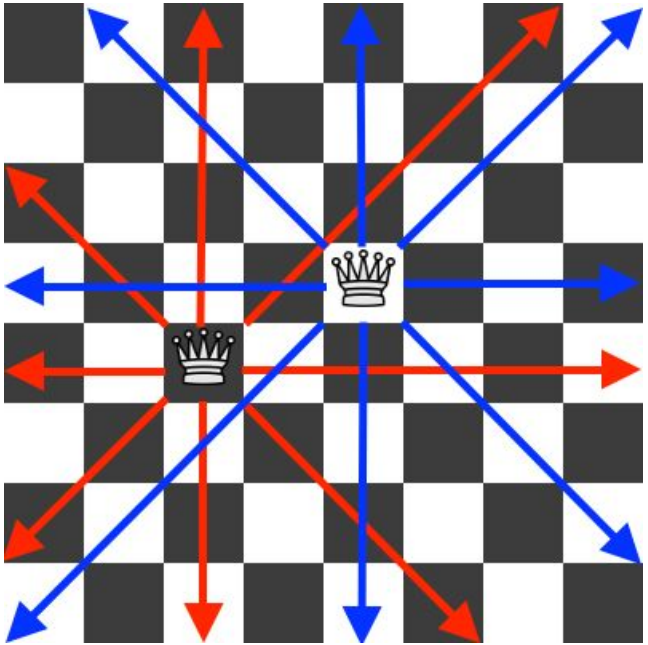
En realitat, però, volem resoldre aquest problema per a qualsevol mida del tauler. Suposem que podem tenir un tauler d'escacs de nxn. Cal posar n reines al tauler sense que n'hi hagi cap parell de reines que es puguin capturar.

Cal dissenyar un programa que llegeixi el nombre n i escrigui l'ubicació de les reines en el cas de que hi hagi una ubicació que compleixi la restricció anterior.

Condicions a tenir en compte:

- Les reines s'han de col·locar de tal manera que no n'hi hagi cap capaç d'amenaçar les altres.
- No pot haver dues reines que comparteixin la mateixa fila, columna o diagonal.
- Les reines es poden moure en horitzontal, vertical i diagonal i qualsevol nombre de caselles.

Recursivitat. Les 8 reines



Podem generar una solució al problema explorant cada fila del tauler i col·locant una reina per columna, tot comprovant a cada pas, que no hi hagi dues reines a la línia d'atac de l'altra.

Un enfocament de la força bruta del problema (backtracking) serà generar totes les combinacions possibles de les vuit reines al tauler d'escacs i rebutjar els estats no vàlids.

Quantes combinacions de 8 reines en un tauler d'escacs de 64 cel·les són possibles?

Recursivitat. Les 8 reines

La fòrmula combinatòria és: que, en el nostre cas, és

$$C(n,k) =_n C_k = \frac{n!}{k! \cdot (n-k)!} \qquad C(64,8) =_{64} C_8 = \frac{64!}{8! \cdot (64-8)!} = 4,426,165,368$$

És evident que l'aproximació de la força bruta no es practica.

Podem reduir encara més el nombre de solucions potencials si observem que una solució vàlida només pot tenir una reina per fila, cosa que significa que podem representar el tauler com una matriu de vuit elements, on cada entrada representa la posició de la columna de la reina a partir de una fila particular. Prenem com a exemple la següent solució del problema:

Les posicions de les reines del tauler anterior es poden representar com les posicions ocupades d'una matriu bidimensional 8x8: [0, 6], [1, 2], [2, 7], [3, 1], [4, 4], [5, 0], [6, 5], [7, 3]. O, com s'ha descrit anteriorment, podem utilitzar una matriu de 8 elements unidimensional: [6, 2, 7, 1, 4, 0, 5, 3].

Recursivitat. Les 8 reines

Si ens fixem atentament en la solució d'exemple [6, 2, 7, 1, 4, 0, 5, 3], observem que es pot construir una solució potencial al trencaclosques de vuit reines generant totes les permutacions possibles d'una matriu de vuit. números, [0, 1, 2, 3, 4, 5, 6, 7] i rebutjar els estats no vàlids (aquells en què dues reines poden atacar-se mútuament). El nombre de totes les permutacions de n objectes únics és n !, que per al nostre cas particular és:

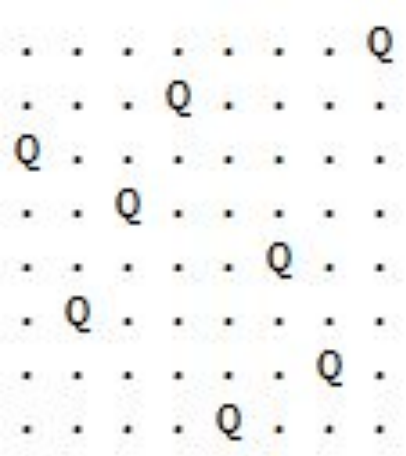
$$n! = 40,320$$

la qual cosa és més raonable que les anteriors 4.426.165.368 situacions per analitzar l'enfocament de la força bruta.

Una solució lleugerament més eficient per al trencaclosques utilitza un enfocament recursiu: suposem que ja hem generat totes les maneres possibles de col·locar k reines a les primeres k files. Per generar les posicions vàlides per a la reina k + 1 col·locem una reina a totes les columnes de la fila k + 1 i rebutgem els estats no vàlids. Fem els passos anteriors fins que totes les vuit reines se situen al tauler. Aquest enfocament generarà les 92 solucions diferents per al trencaclosques de vuit reines.

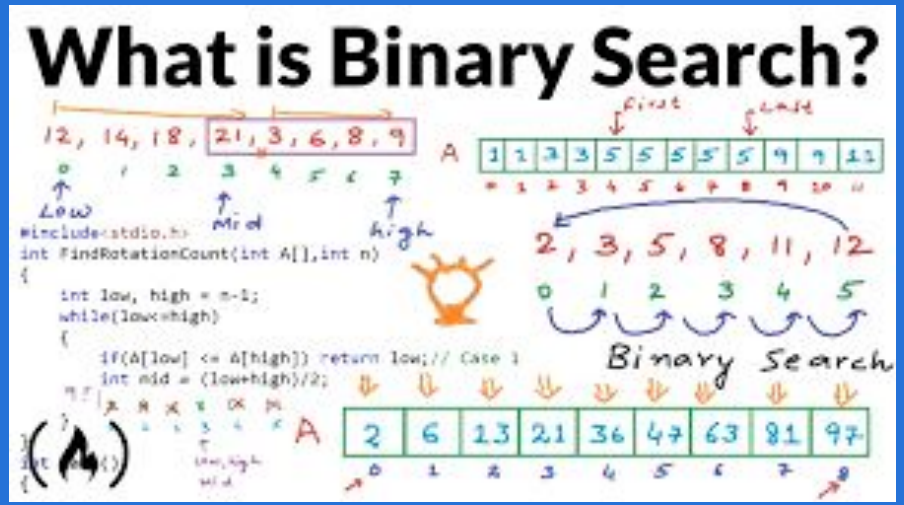
Recursivitat. Les 8 reines

```
boolean solveNQUtil(int board[][], int col) {

        if (col >= N)
            return true;

        for (int i = 0; i < N; i++) {

            if (isSafe(board, i, col)) {

                board[i][col] = 1;


                if (solveNQUtil(board, col + 1) == true)
                    return true;


                board[i][col] = 0; // BACKTRACK
            }
        }

        return false;
}
```



[Exemple de solució en Java]

Binary search



Recursivitat. Binary search

La cerca binària (cerca dicotòmica) és un algorisme recursiu de cerca que troba la posició d'un valor en un array ordenat. Compara el valor amb l'element en el mitjà del array, si no són iguals, la meitat en la qual el valor no pot estar és eliminada i la cerca continua en la meitat restant fins que el valor es trobi.

Suposem que tenim el següent array:

1	2	3	4	5	6	7

i que l'usuari vol trobar el **7**.

Com ho fem?

Recursivitat. Binary search

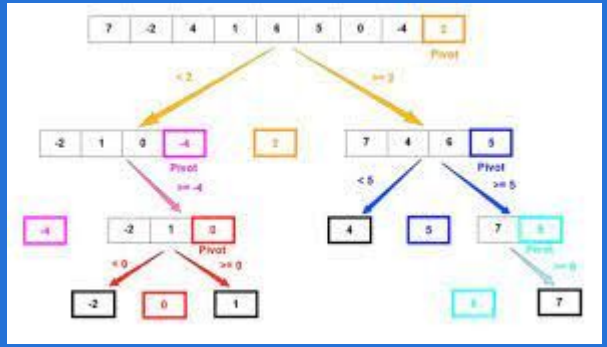
1. Dividim a la meitat el nostre array. Com que no podem escollir posicions decimals, escollim l'entera més propera (anterior o posterior, segons la implementació).
   El fragment seleccionat és [2,3,1]
2. Fem una cerca del nostre element dins d'aquest fragment. Si no es troba, es descarta (s'elimina).
3. Tornem a dividir el fragment restant entre dos i repetim el pas 2. Ara el fragment és [5,4]. Com que no el troba, el descarta.
4. Repetim el pas tres i obtenim el fragment [6,7]. Repetim el pas dos i en aquesta execució, trobem l'element.

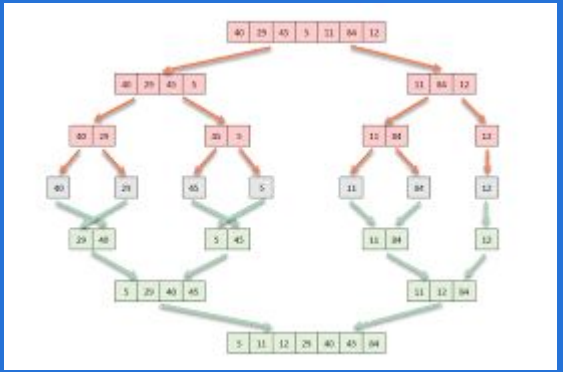
Recursivitat. Binary search

```
public int binarySearch(int arr[], int first, int last, int key){
    if (last>=first){
        int mid = first + (last - first)/2;
        if (arr[mid] == key){
            return mid;
        }
        if (arr[mid] > key){
            return binarySearch(arr, first, mid-1, key);
        }else{
            return binarySearch(arr, mid+1, last, key);
        }
    }
    return -1;
}
```

Exemple en java

Ordenació





Recursivitat. Ordenació

Tipus d'algorismes

- **Simples:** avaluen els valors de cada parell de posicions i els intercanvien.
  - **BubbleSort**
- **Divide and Conquer:** divideixen el problema en problemes més senzill per resoldre un problema complex. S'aplica la recursivitat
  - **MergeSort, QuickSort**

Recursivitat.Resum

- La **Recursivitat** és una estratègia per a solucionar problemes cridant a una funció dins de si mateixa.
- Avantatges:
  - Codis més curts.
  - Codis més legibles
- Quan s'ha d'utilitzar la recursivitat?
  - Quan un problema es pot dividir en subproblemes idèntics, però més petits.
  - Per explorar l'espai en un problema de cerca
- Quines són les 3 normes principals?