

IFT1015 Programmation 1

Récurtivité

Marc Feeley

Récurtivité

Récurtivité

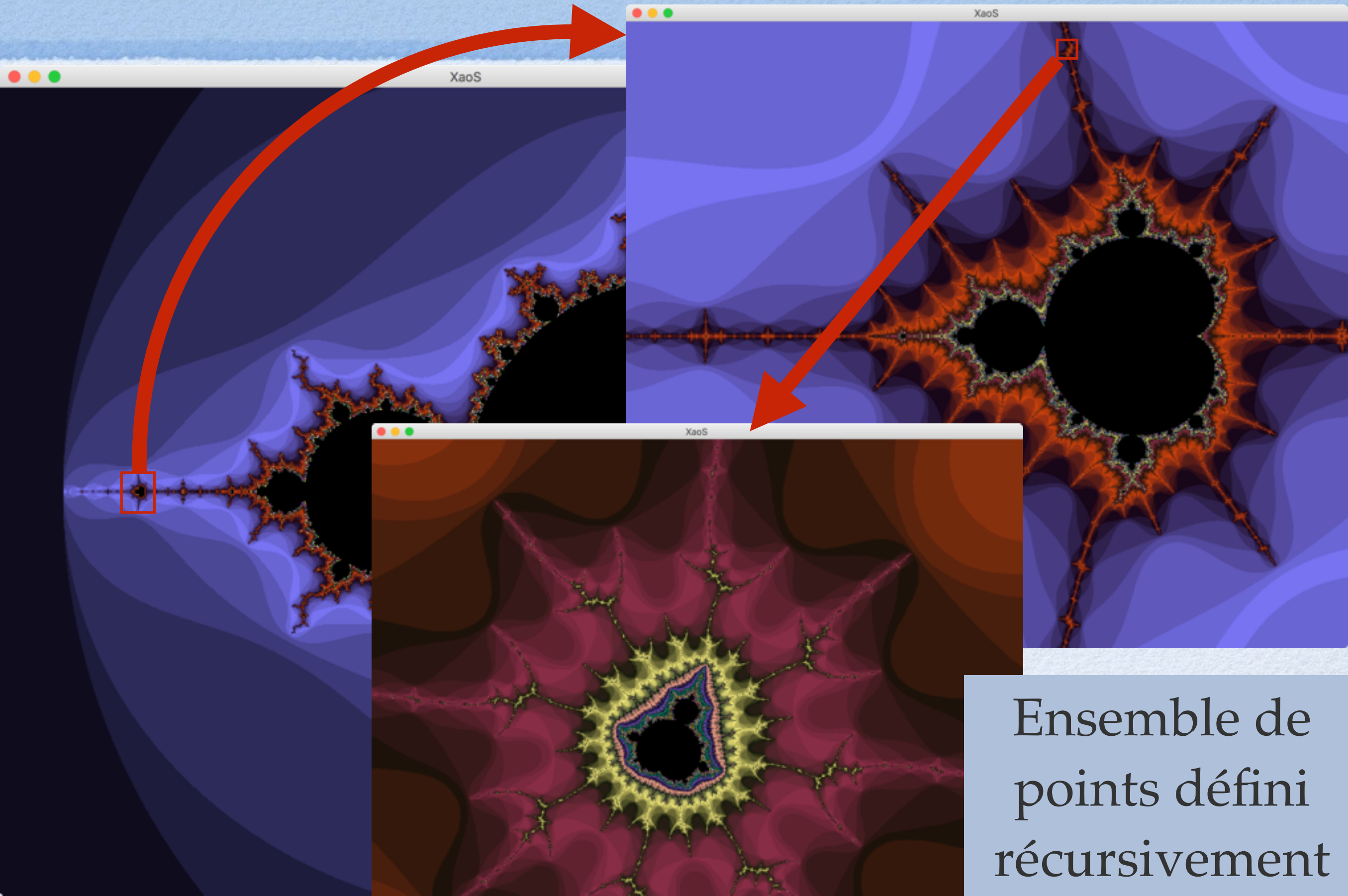
- Les appels de fonction font intervenir une fonction **appelante** et une fonction **appelée**
- Lorsque la fonction appelée est la fonction appelante, on dit que c'est un appel récursif de la fonction et que c'est une **fonction récursive**
- Cette technique permet d'exprimer des traitements complexes en peu de code, par exemple :
 - trier un ensemble de valeurs
 - chercher une valeur dans un tableau trié
 - parcours hiérarchiques (système de fichiers, ...)
 - dessins inspirés de la nature (arbres, flocons, ...)

Récurtivité

- Des exemples de récursivité dans la nature :



L'ensemble de Mandelbrot



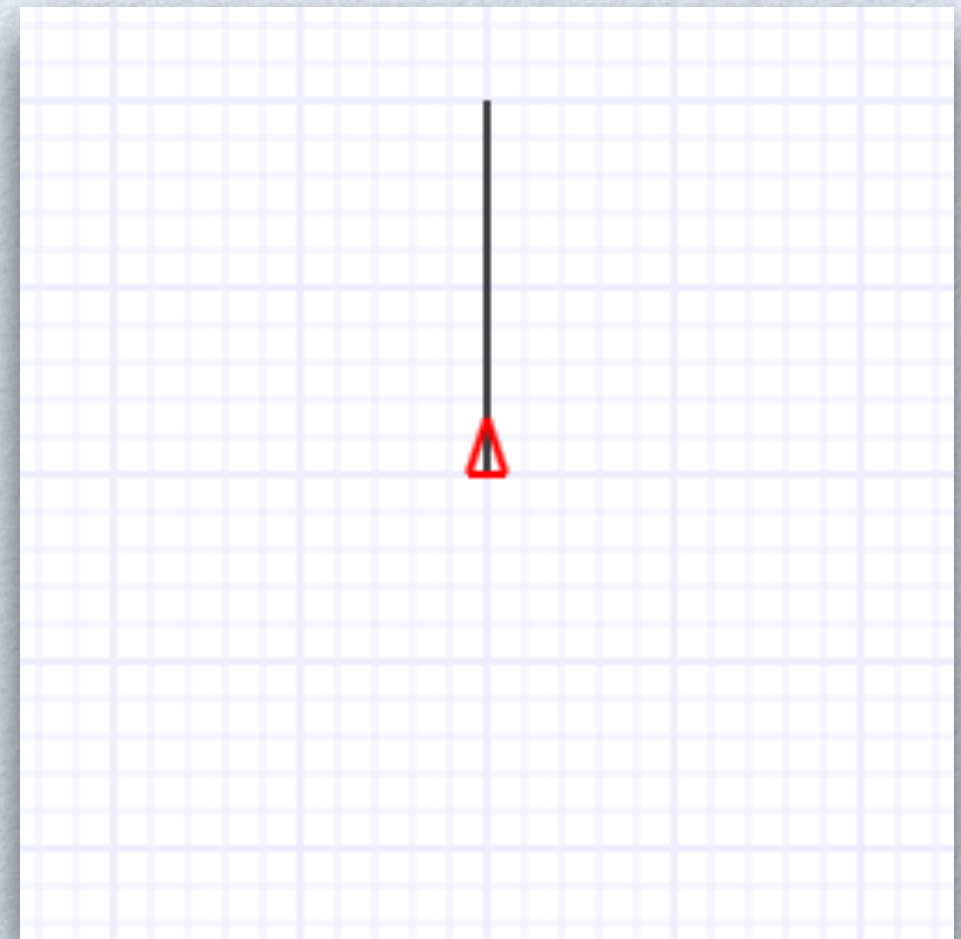
Ensemble de
points défini
récursivement

Dessiner un arbre

- Si on utilise le **niveau minimum de détail**, un arbre peut être vu simplement comme une **tige verticale**

```
var arb0 = function (h) { // h = hauteur  
  fd(h);  
  bk(h);  
};
```

niveau de détail
le plus faible = 0



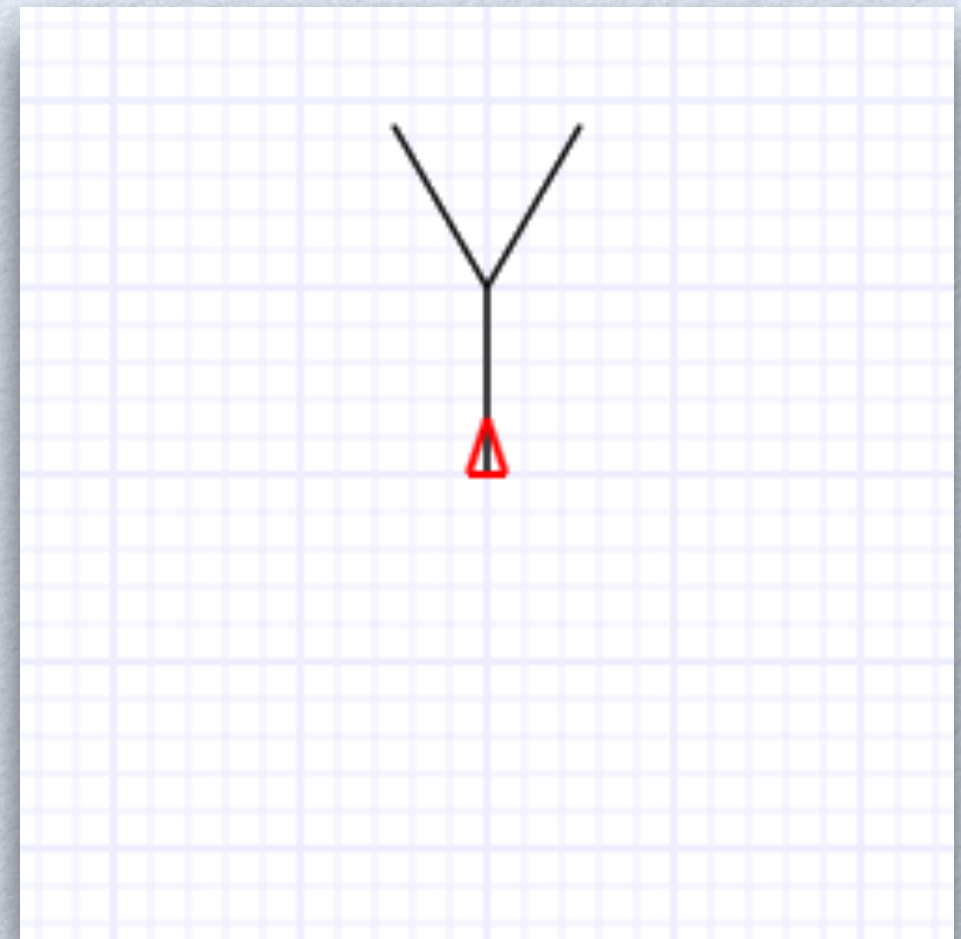
```
arb0(100);
```


Dessiner un arbre

- Un peu plus de détail : la moitié inférieure est une tige, la moitié supérieure deux arbres de niveau 0

```
var arb0 = function (h) { // h = hauteur
  fd(h) ;
  bk(h) ;
};

var arb1 = function (h) {
  fd(h/2) ;
  rt(30) ; arb0(h/2) ; lt(30) ;
  lt(30) ; arb0(h/2) ; rt(30) ;
  bk(h/2) ;
};
```



`arb1(100) ;`

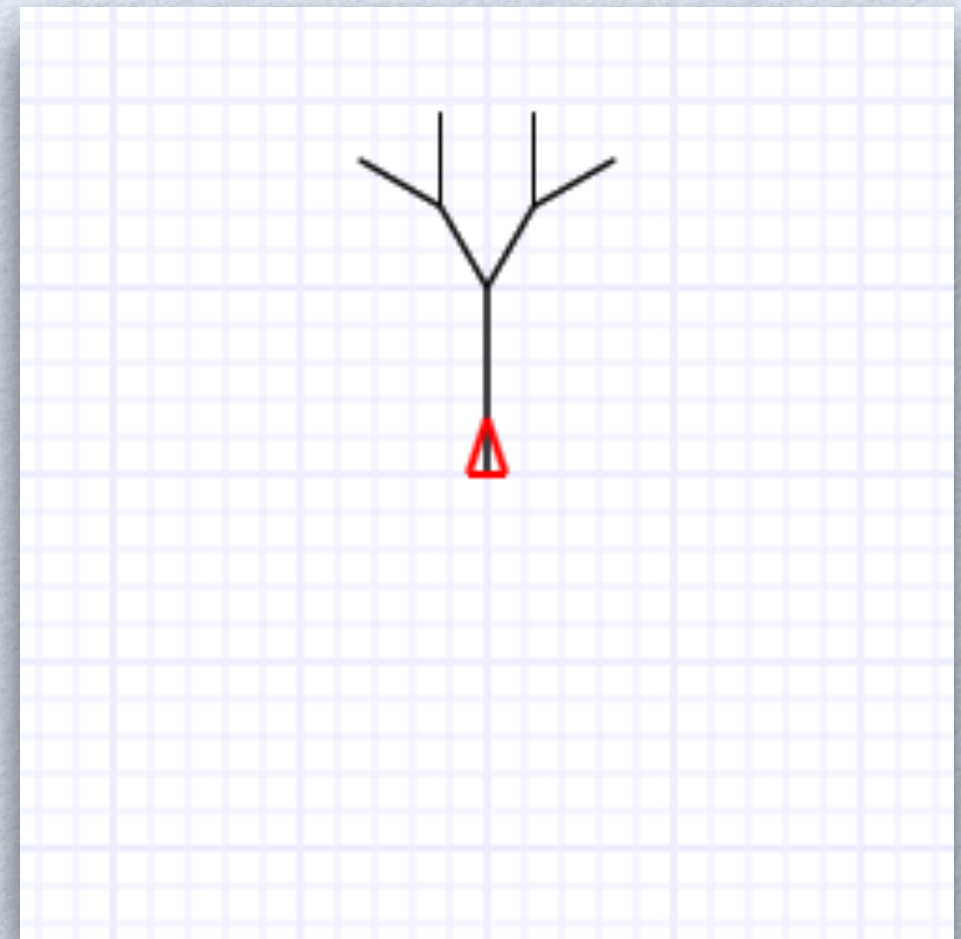
Dessiner un arbre

- Encore plus de détail : la moitié inférieure est une tige, la moitié supérieure deux arbres de niveau 1

```
var arb0 = function (h) { // h = hauteur
  fd(h) ;
  bk(h) ;
};

var arb1 = function (h) {
  fd(h/2) ;
  rt(30) ; arb0(h/2) ; lt(30) ;
  lt(30) ; arb0(h/2) ; rt(30) ;
  bk(h/2) ;
};

var arb2 = function (h) {
  fd(h/2) ;
  rt(30) ; arb1(h/2) ; lt(30) ;
  lt(30) ; arb1(h/2) ; rt(30) ;
  bk(h/2) ;
};
```

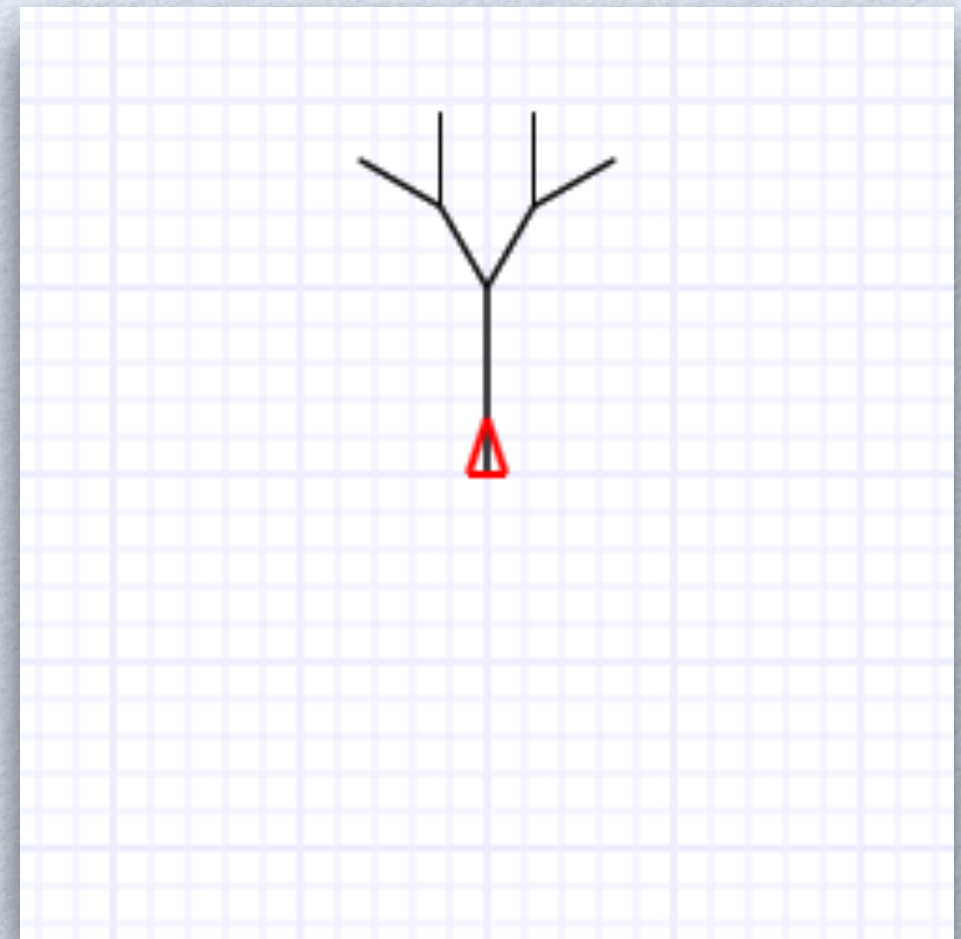


arb2(100) ;

Dessiner un arbre

- On peut généraliser en une fonction **arb** qui prend le niveau de détail (**d**) en paramètre :

```
var arb = function (d, h) {  
  if (d == 0) {  
    fd(h);  
    bk(h);  
  } else {  
    fd(h/2);  
    rt(30); arb(d-1, h/2); lt(30);  
    lt(30); arb(d-1, h/2); rt(30);  
    bk(h/2);  
  }  
};
```

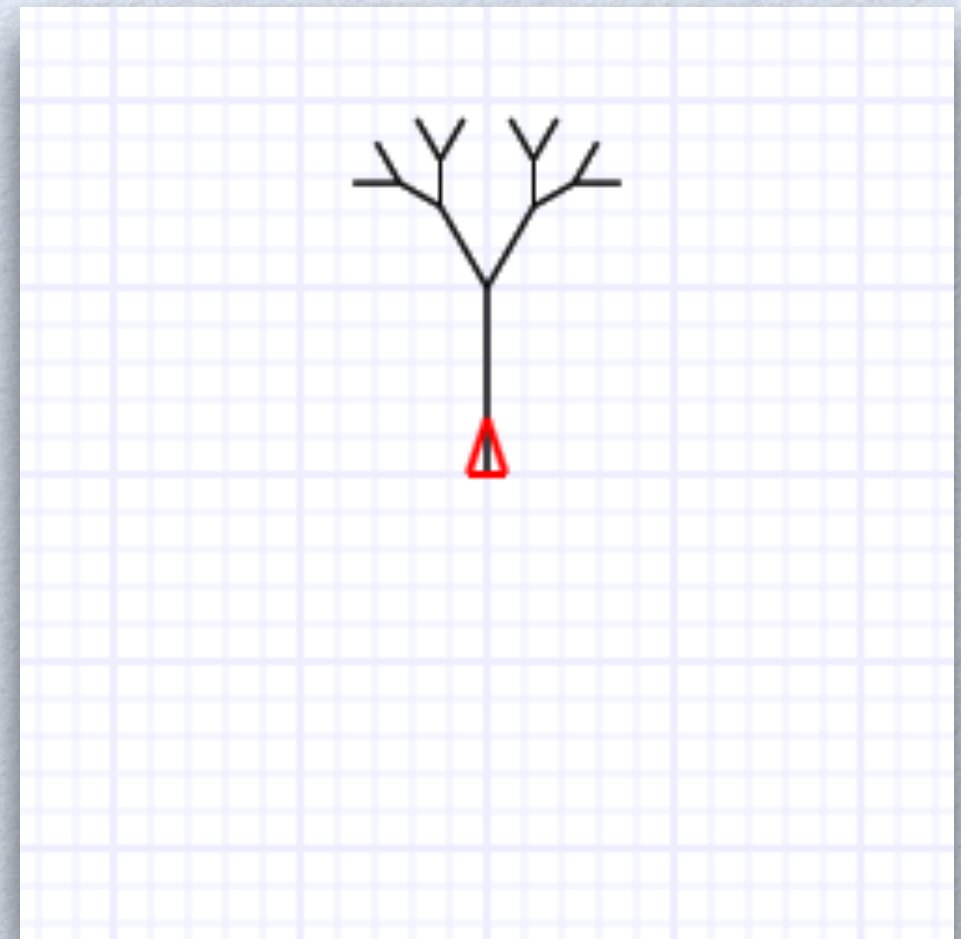


`arb(2, 100);`

Dessiner un arbre

- On peut généraliser en une fonction **arb** qui prend le niveau de détail (**d**) en paramètre :

```
var arb = function (d, h) {  
  if (d == 0) {  
    fd(h);  
    bk(h);  
  } else {  
    fd(h/2);  
    rt(30); arb(d-1, h/2); lt(30);  
    lt(30); arb(d-1, h/2); rt(30);  
    bk(h/2);  
  }  
};
```

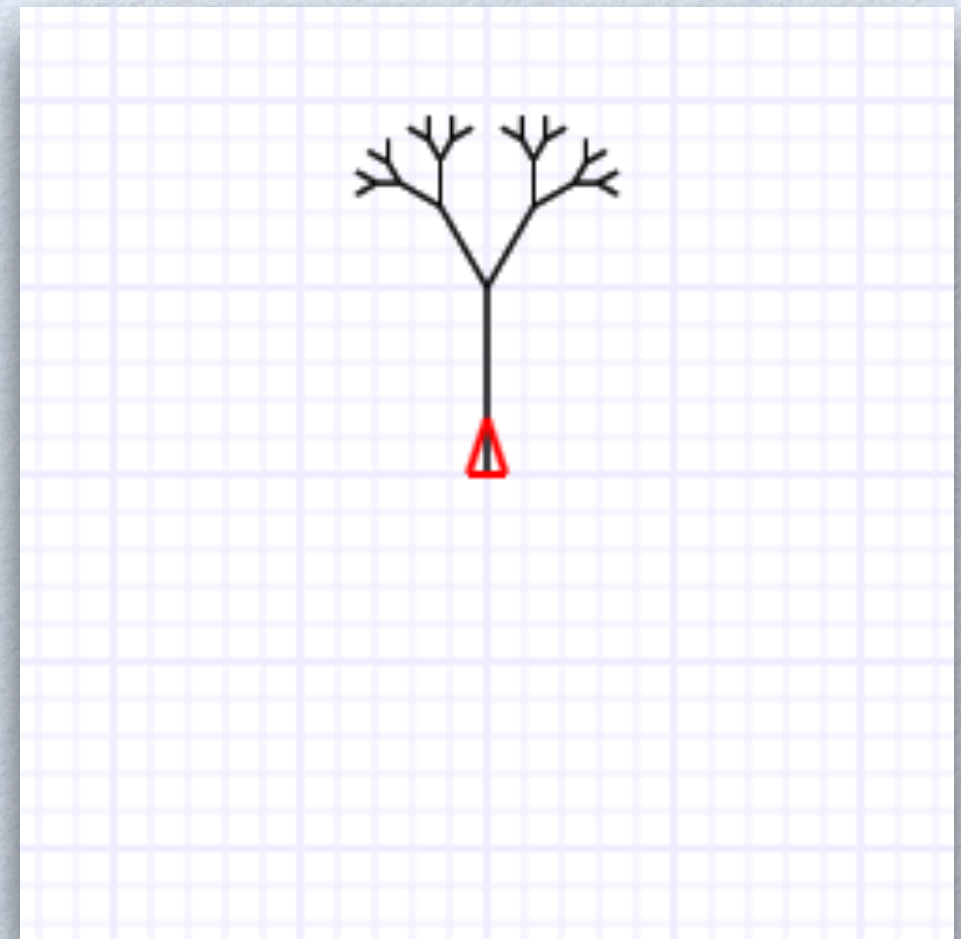


arb(3, 100);

Dessiner un arbre

- On peut généraliser en une fonction **arb** qui prend le niveau de détail (**d**) en paramètre :

```
var arb = function (d, h) {  
  if (d == 0) {  
    fd(h);  
    bk(h);  
  } else {  
    fd(h/2);  
    rt(30); arb(d-1, h/2); lt(30);  
    lt(30); arb(d-1, h/2); rt(30);  
    bk(h/2);  
  }  
};
```

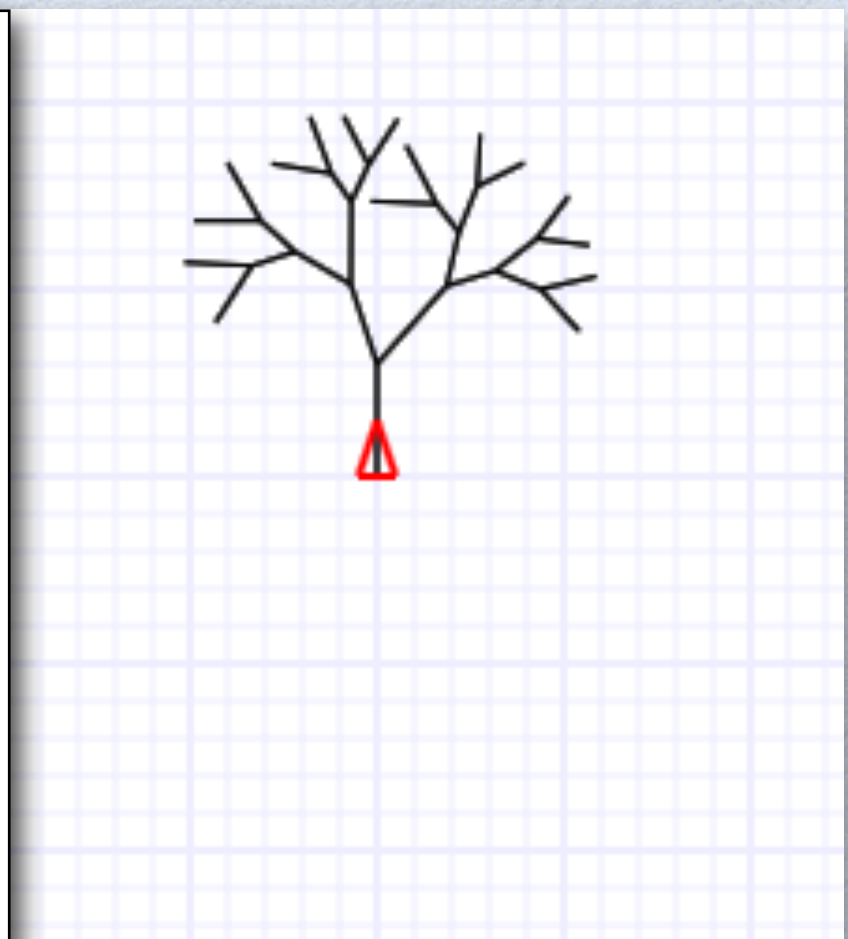


arb(4, 100);

Dessiner un arbre

- Pour que ce soit plus naturel (moins symétrique) on peut rajouter des **facteurs aléatoires** :

```
var arb = function (d, h) {  
  if (d == 0) {  
    fd(h);  
    bk(h);  
  } else {  
    var ratio = 0.3 + 0.2*Math.random();  
    var angle = 50 * (Math.random() - 0.5);  
    fd(h*ratio);  
    rt(angle);  
    rt(30); arb(d-1, h*(1-ratio)); lt(30);  
    lt(30); arb(d-1, h*(1-ratio)); rt(30);  
    lt(angle);  
    bk(h*ratio);  
  }  
};
```

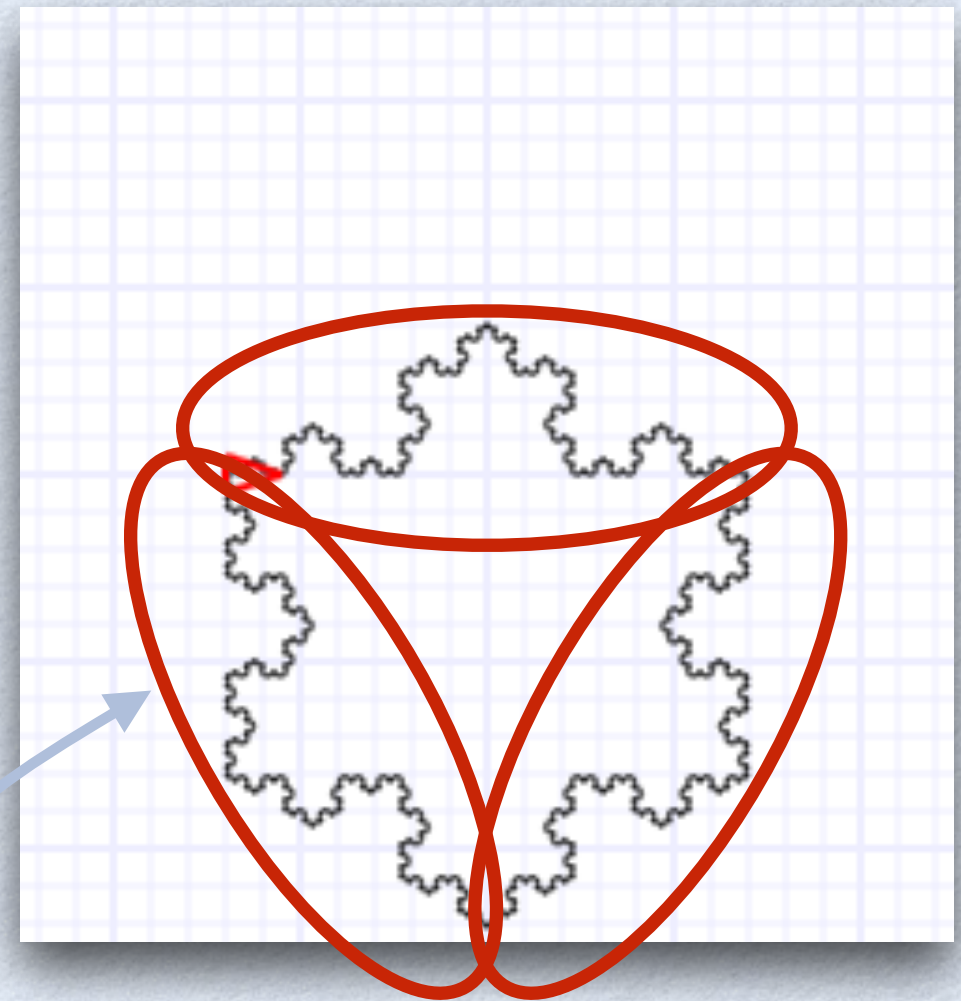


```
arb(4, 100);
```


Dessiner un flocon

- Un flocon de neige peut également se dessiner récursivement :

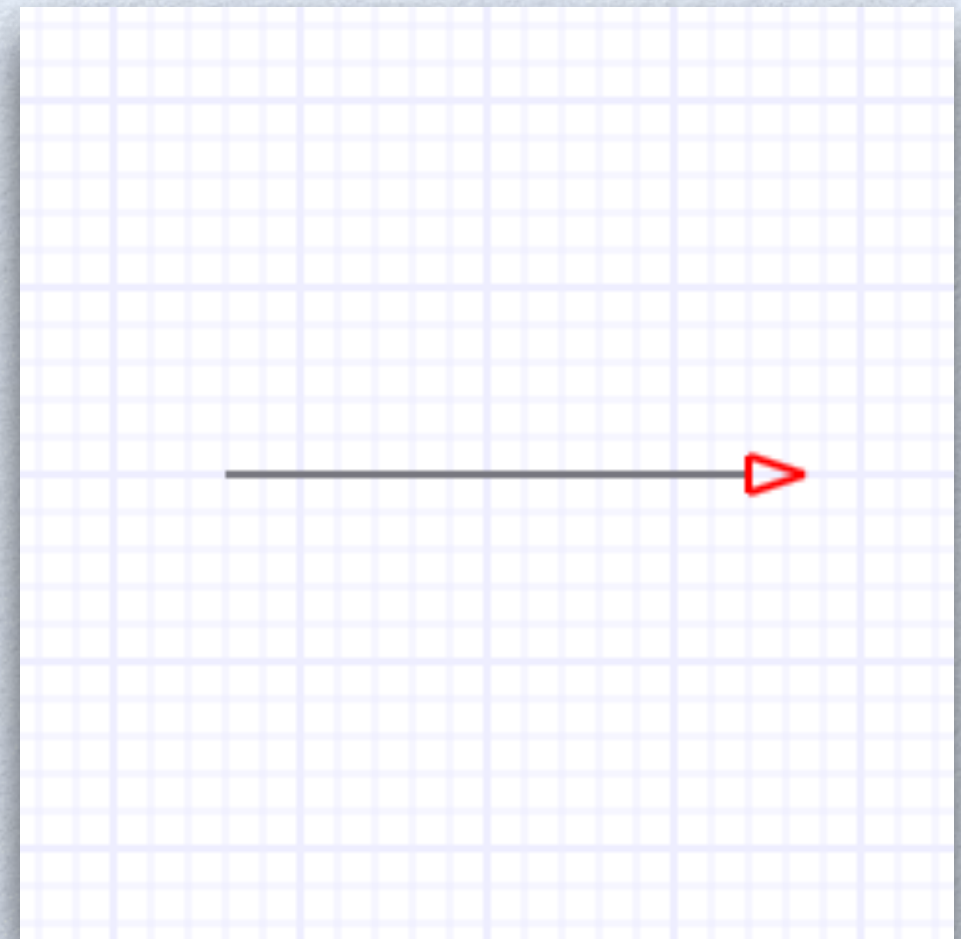
trois fois le même dessin
formant les côtés d'un
triangle équilatéral



Dessiner un flocon

- Un côté de flocon avec le minimum de détail c'est un segment de droite :

```
pu(); lt(90); fd(70); rt(180); pd();  
  
var c0 = function (dist) {  
  fd(dist);  
};
```

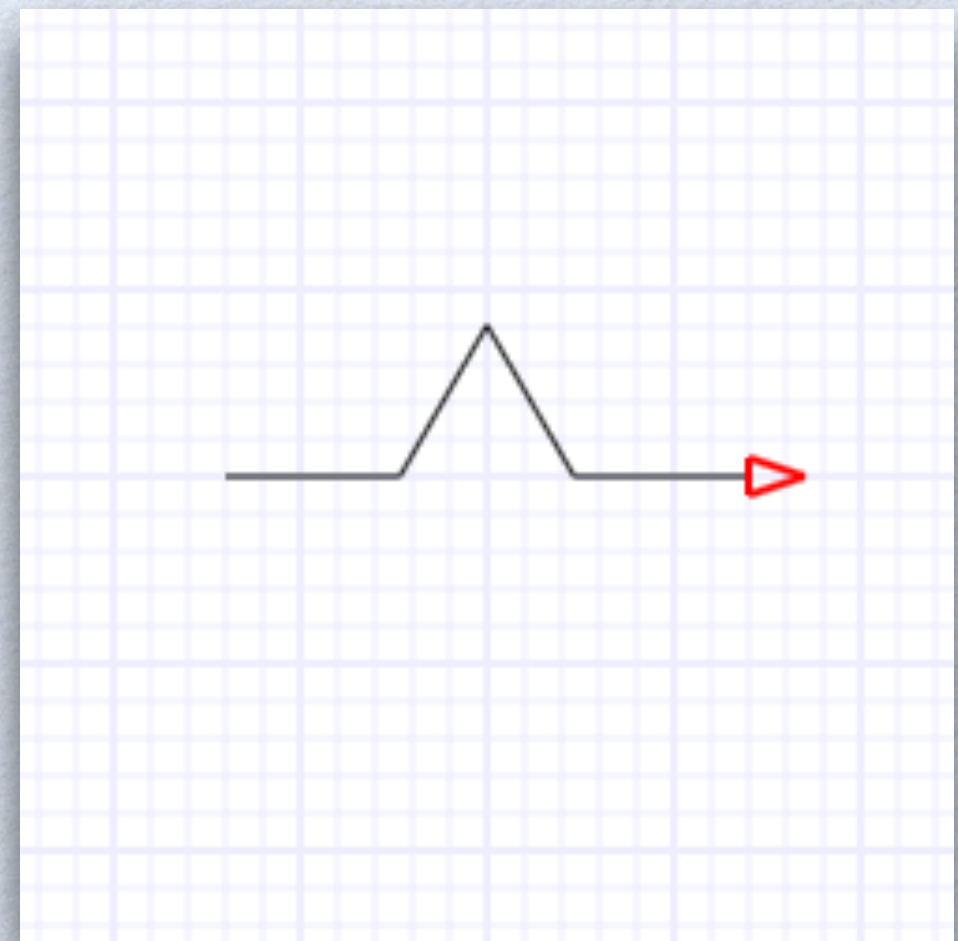


c0(140);

Dessiner un flocon

- Au prochain niveau de détail un côté c'est 4 côtés de niveau 0 de même longueur en “chapeau”

```
pu(); lt(90); fd(70); rt(180); pd();  
  
var c0 = function (dist) {  
  fd(dist);  
};  
  
var c1 = function (dist) {  
  c0(dist/3); lt(60);  
  c0(dist/3); rt(120);  
  c0(dist/3); lt(60);  
  c0(dist/3);  
};
```



c1(140);

Dessiner un flocon

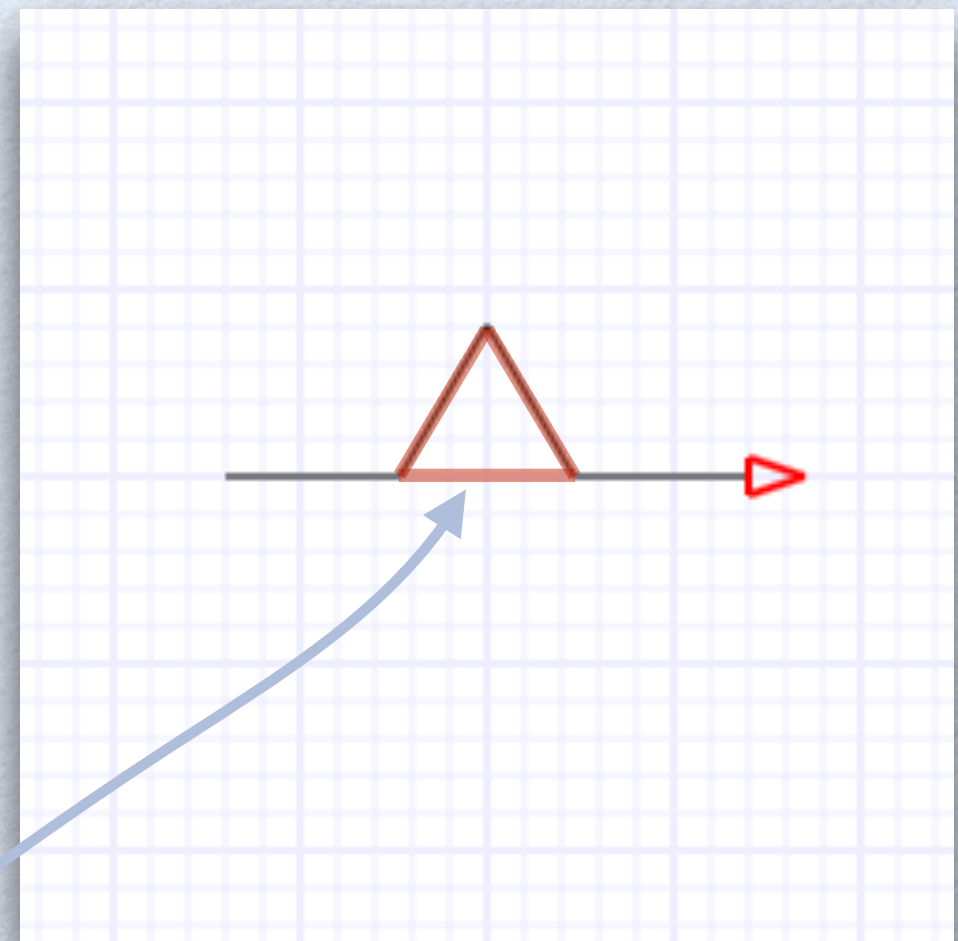
- Au prochain niveau de détail un côté c'est 4 côtés de niveau 0 de même longueur en “chapeau”

```
pu(); lt(90); fd(70); rt(180); pd();
```

```
var c0 = function (dist) {  
  fd(dist);  
};
```

```
var c1 = function (dist) {  
  c0(dist/3); lt(60);  
  c0(dist/3); rt(120);  
  c0(dist/3); lt(60);  
  c0(dist/3);  
};
```

Cette partie forme un triangle équilatéral



c1(140);

Dessiner un flocon

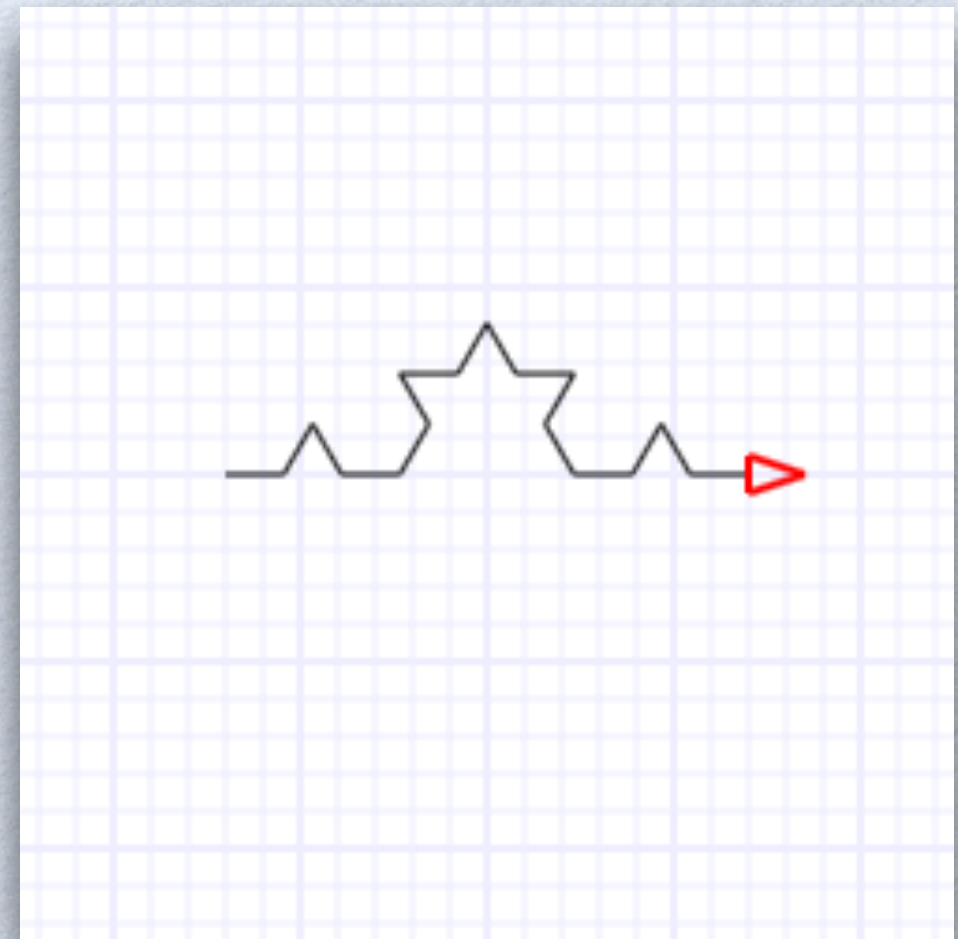
- Au prochain niveau de détail un côté c'est 4 côtés de niveau 1 de même longueur en “chapeau”

```
pu(); lt(90); fd(70); rt(180); pd();

var c0 = function (dist) {
  fd(dist);
};

var c1 = function (dist) {
  c0(dist/3); lt(60);
  c0(dist/3); rt(120);
  c0(dist/3); lt(60);
  c0(dist/3);
};

var c2 = function (dist) {
  c1(dist/3); lt(60);
  c1(dist/3); rt(120);
  c1(dist/3); lt(60);
  c1(dist/3);
};
```



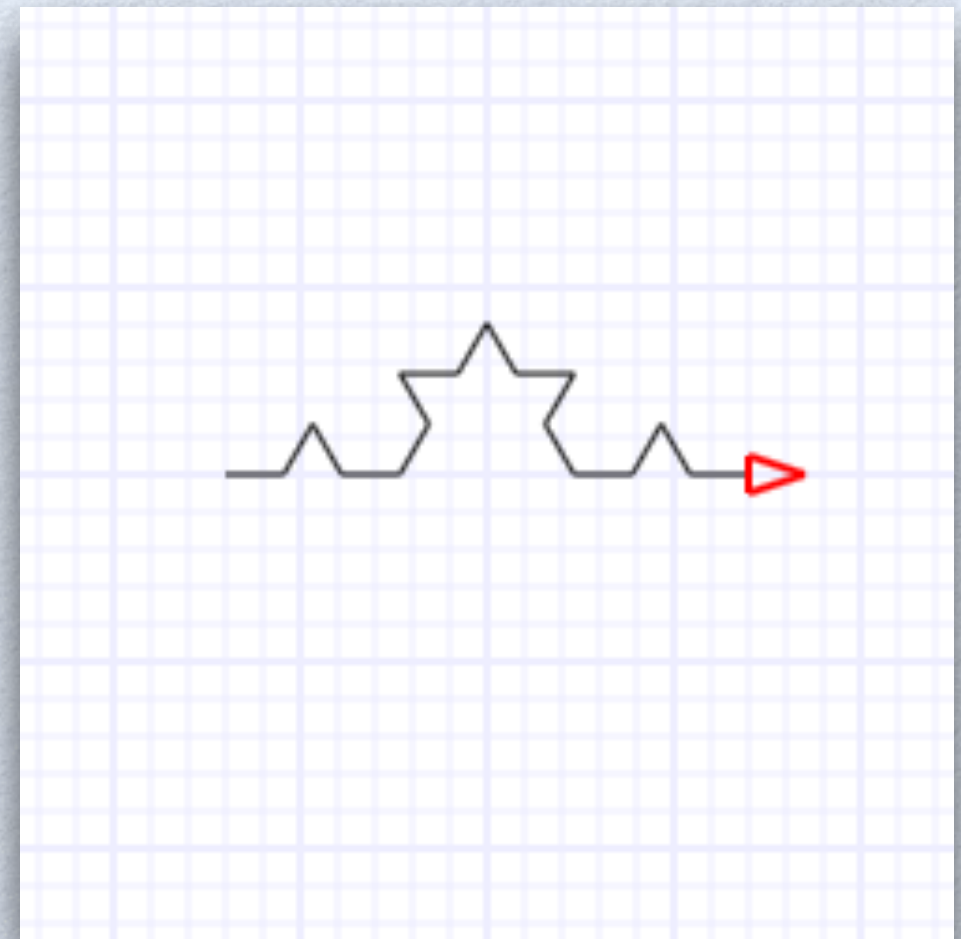
c2(140);

Dessiner un flocon

- Généralisation de cette fonction pour n'importe quel niveau de détail :

```
pu(); lt(90); fd(70); rt(180); pd();

var c = function (d, dist) {
  if (d == 0) {
    fd(dist);
  } else {
    c(d-1, dist/3); lt(60);
    c(d-1, dist/3); rt(120);
    c(d-1, dist/3); lt(60);
    c(d-1, dist/3);
  }
};
```



`c(2, 140);`

Dessiner un flocon

- Généralisation de cette fonction pour n'importe quel niveau de détail :

```
pu(); lt(90); fd(70); rt(180); pd();

var c = function (d, dist) {
  if (d == 0) {
    fd(dist);
  } else {
    c(d-1, dist/3); lt(60);
    c(d-1, dist/3); rt(120);
    c(d-1, dist/3); lt(60);
    c(d-1, dist/3);
  }
};
```



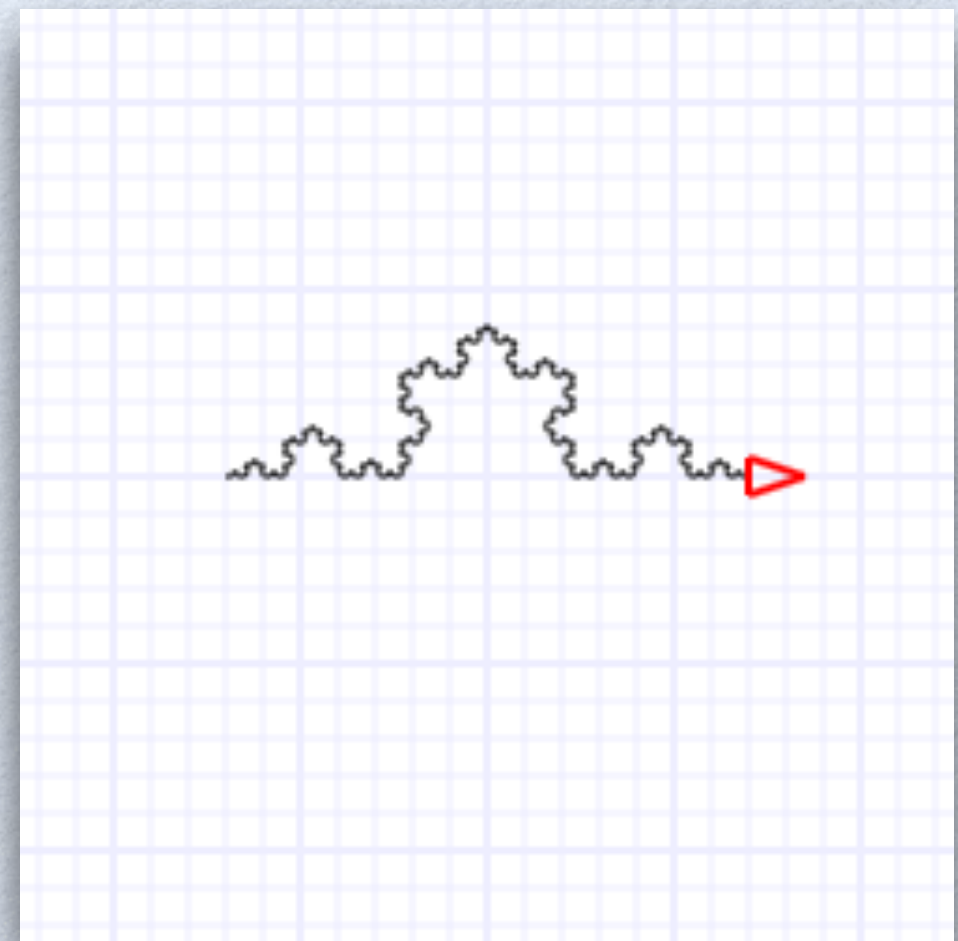
`c(3, 140);`

Dessiner un flocon

- Généralisation de cette fonction pour n'importe quel niveau de détail :

```
pu(); lt(90); fd(70); rt(180); pd();

var c = function (d, dist) {
  if (d == 0) {
    fd(dist);
  } else {
    c(d-1, dist/3); lt(60);
    c(d-1, dist/3); rt(120);
    c(d-1, dist/3); lt(60);
    c(d-1, dist/3);
  }
};
```



`c(4, 140);`

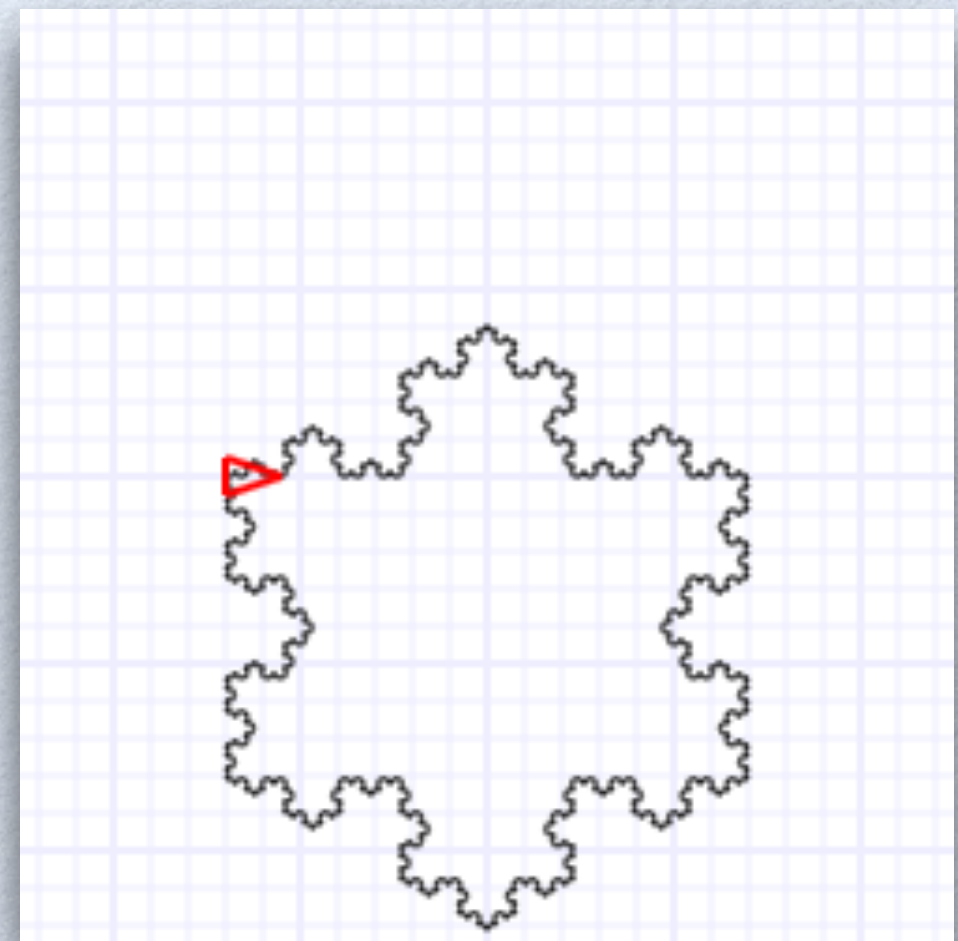
Dessiner un flocon

- Un flocon est formé de 3 côtés fractals

```
pu(); lt(90); fd(70); rt(180); pd();

var c = function (d, dist) {
  if (d == 0) {
    fd(dist);
  } else {
    c(d-1, dist/3); lt(60);
    c(d-1, dist/3); rt(120);
    c(d-1, dist/3); lt(60);
    c(d-1, dist/3);
  }
};

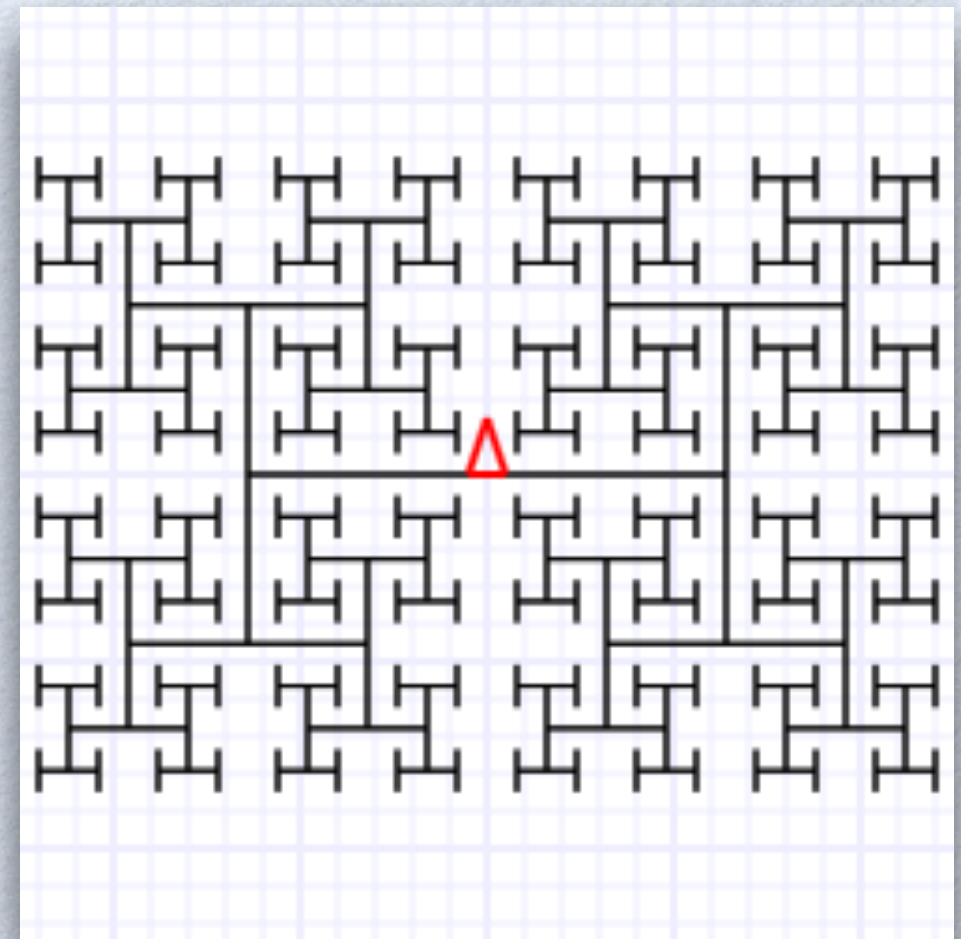
var flocon = function (d, dist) {
  for (var i=0; i<3; i++) {
    c(d, dist);
    rt(120);
  }
};
```



`flocon(4, 140);`

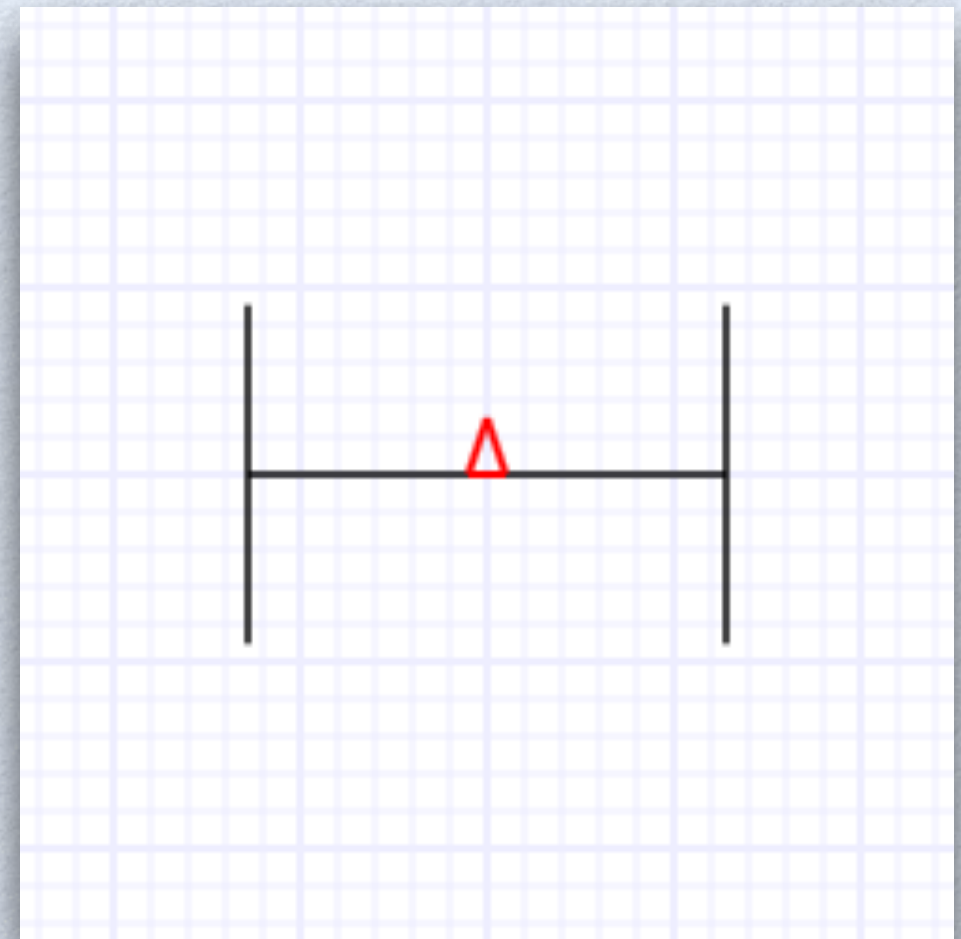
Dessiner un arbre-H

- Un arbre-H est un genre d'arbre constitué de segments formant des H :



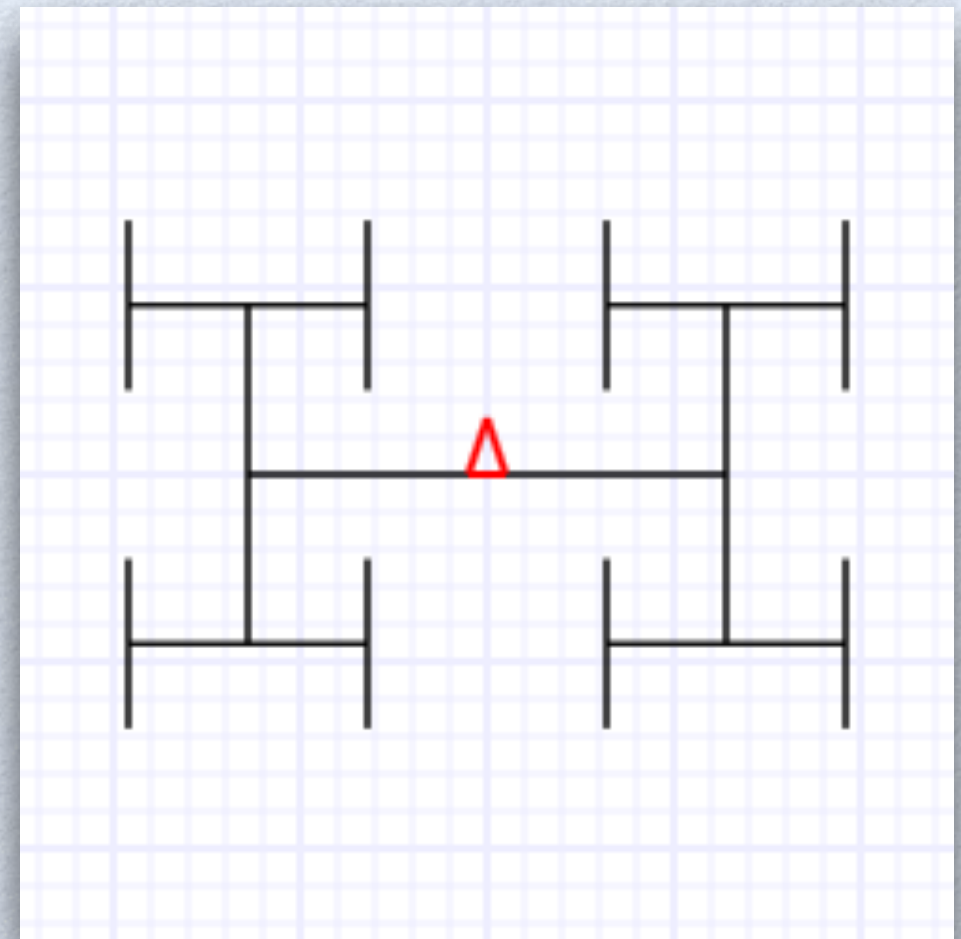
Dessiner un arbre-H

- On commence avec un simple H :



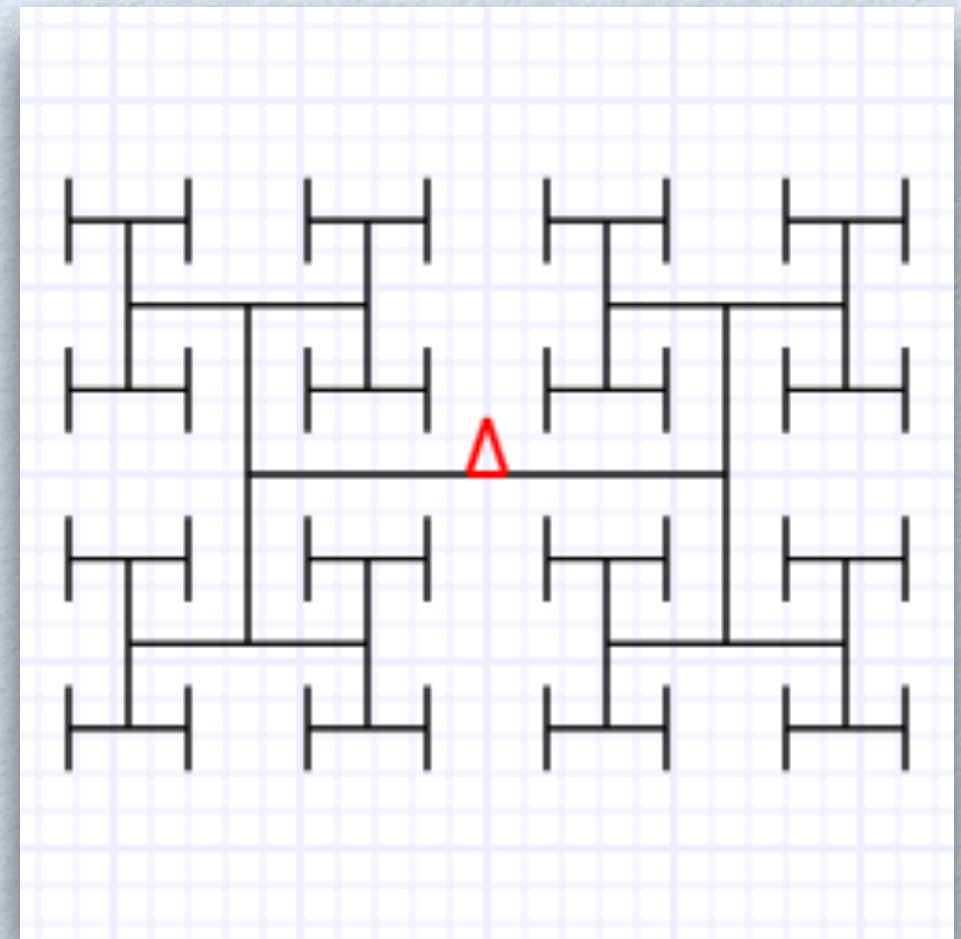
Dessiner un arbre-H

- Puis on rajoute des H de demi grandeur aux bouts des barres verticales :



Dessiner un arbre-H

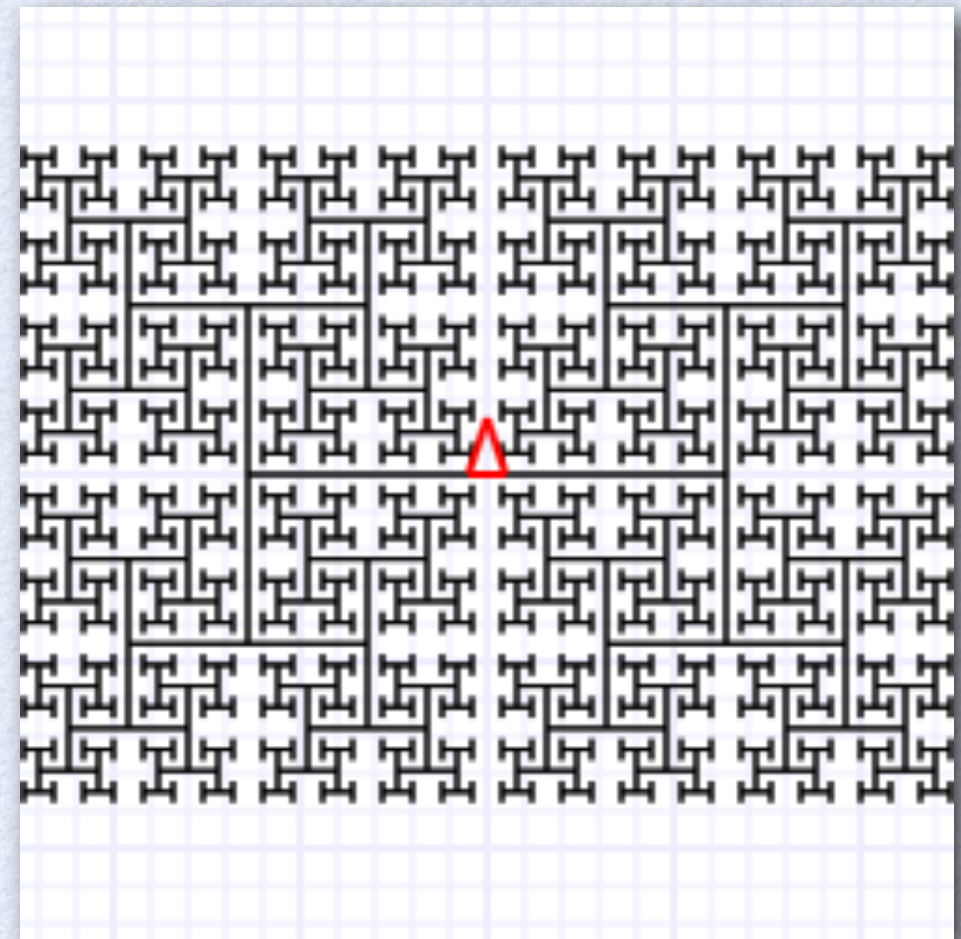
- Et on continue avec des H $1/4$ de la taille du H de départ :



Dessiner un arbre-H

- Définition réursive :

```
var arbh = function (dist) {  
  if (dist > 2) {  
    rt(90);  
    fd(dist);  
    arbh(dist/Math.sqrt(2));  
    bk(dist*2);  
    arbh(dist/Math.sqrt(2));  
    fd(dist);  
    lt(90);  
  }  
};
```

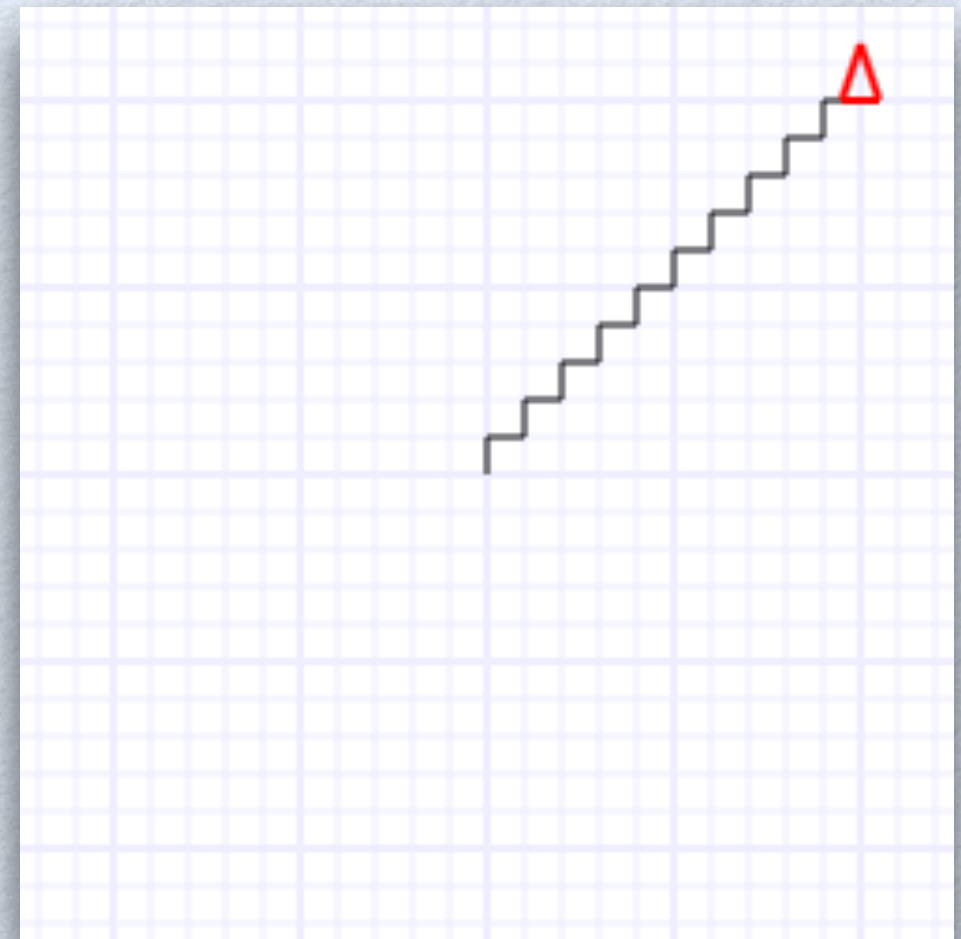


arbh(64);

Dessiner un escalier

- Un escalier peut être vu comme une succession de n marches :

```
var marche = function () {  
    fd(10); rt(90); fd(10); lt(90);  
};  
  
var escalier = function (n) {  
    for (var i=0; i<n; i++) {  
        marche();  
    }  
};
```

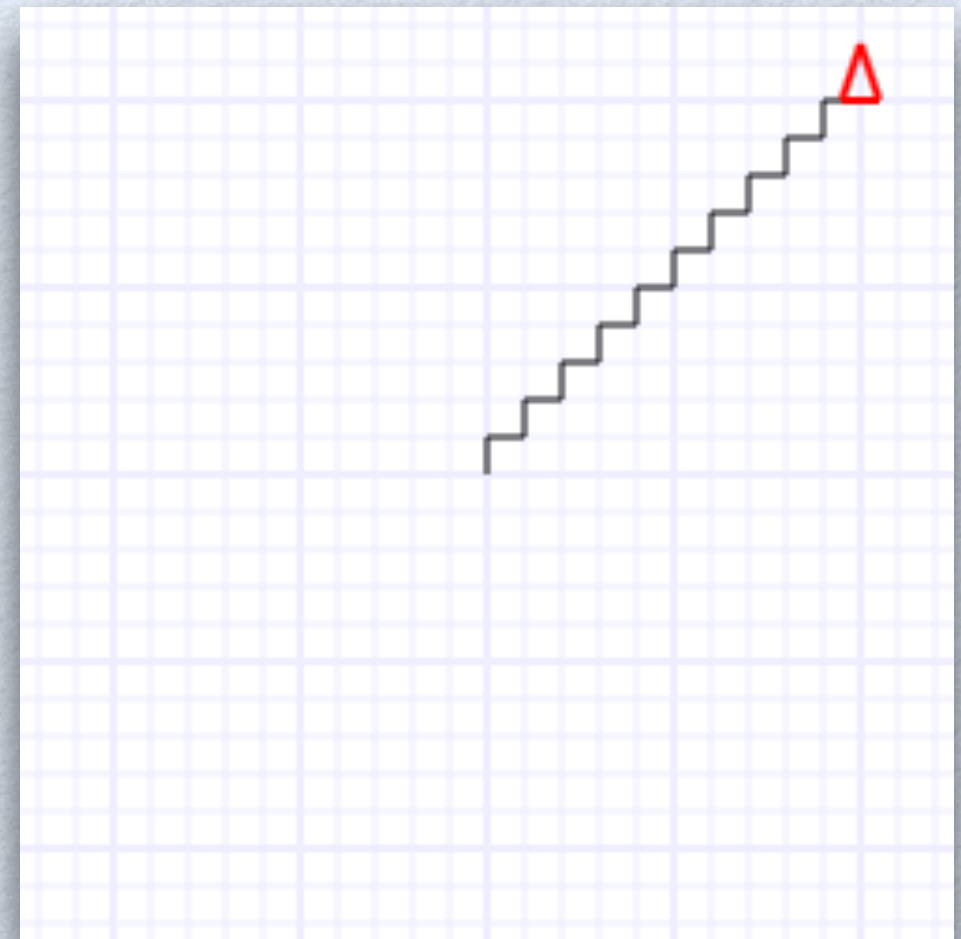


escalier(10);

Dessiner un escalier

- Un escalier peut être vu comme une succession de n marches :

```
var marche = function () {  
    fd(10); rt(90); fd(10); lt(90);  
};  
  
var escalier = function (n) {  
    while (n > 0) {  
        marche();  
        n--;  
    }  
};
```



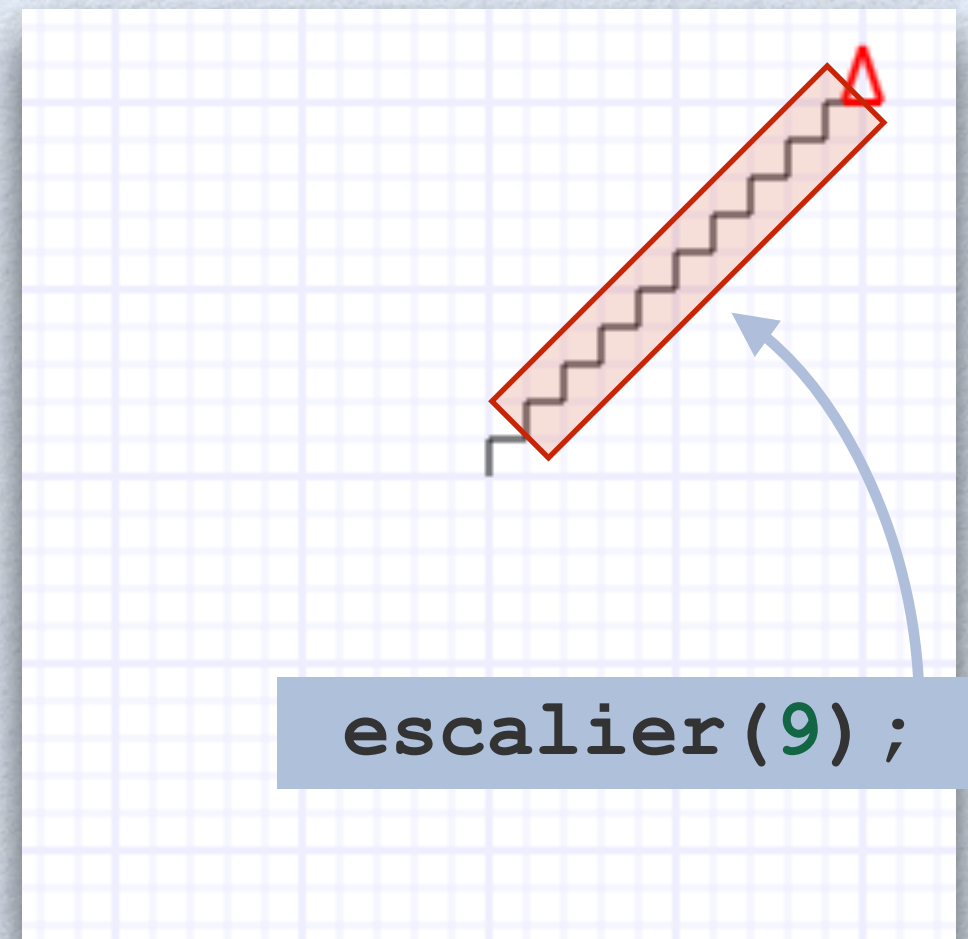
escalier(10);

Dessiner un escalier

- Un escalier peut aussi être vu comme une marche suivie d'un plus petit escalier (avec $n-1$ marches) :

```
var marche = function () {  
  fd(10); rt(90); fd(10); lt(90);  
};  
  
var escalier = function (n) {  
  if (n > 0) {  
    marche();  
    escalier(n-1);  
  }  
};
```

réursion terminale
(se trouvant à la
“fin” de la fonction)



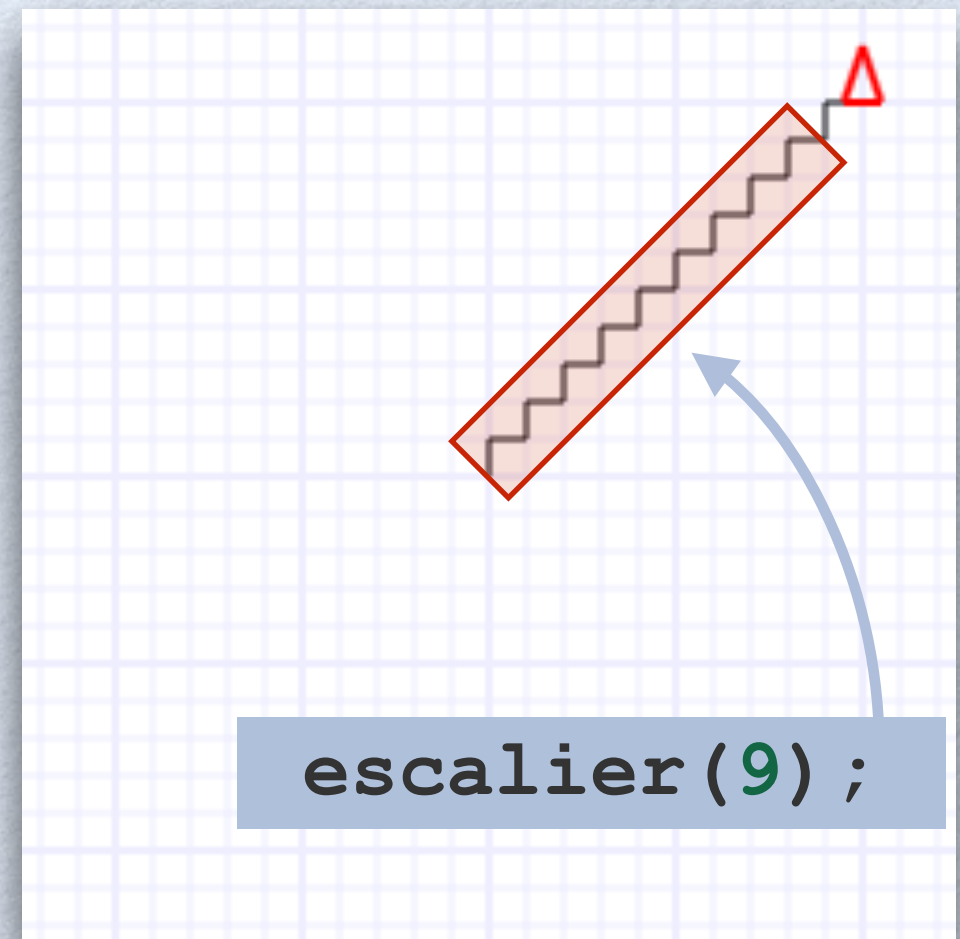
escalier(10);

Dessiner un escalier

- Un escalier peut aussi être vu comme un plus petit escalier (avec $n-1$ marches) suivi d'une marche :

```
var marche = function () {  
  fd(10); rt(90); fd(10); lt(90);  
};  
  
var escalier = function (n) {  
  if (n > 0) {  
    escalier(n-1);  
    marche();  
  }  
};
```

réursion non-terminale



escalier(10);

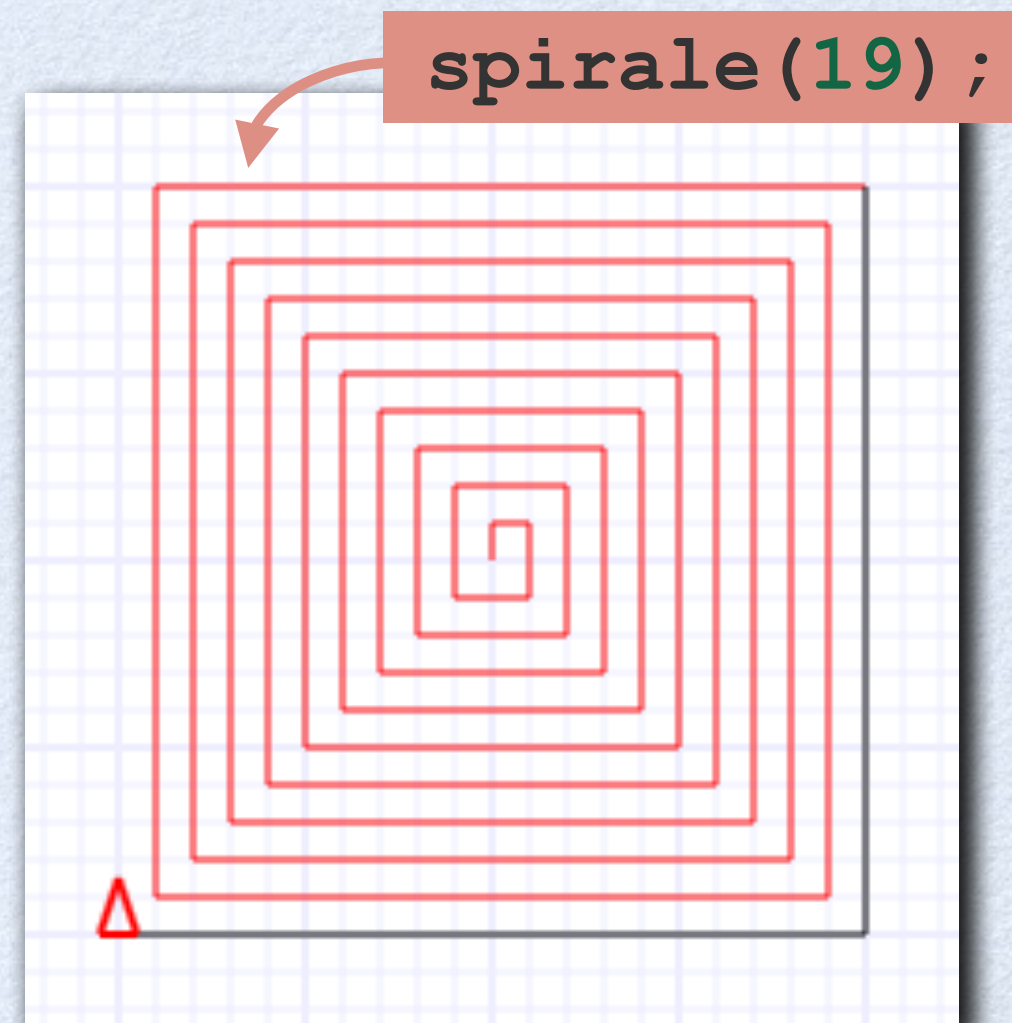
Dessiner une spirale

- Versions avec boucle et avec récursivité :

```
for (var i=1; i<=20; i++) {  
  fd(i*10); rt(90);  
  fd(i*10); rt(90);  
}
```

```
var spirale = function (n) {  
  if (n > 0) {  
    spirale(n-1);  
    fd(n*10); rt(90);  
    fd(n*10); rt(90);  
  }  
};  
  
spirale(20);
```

récursion non-terminale



Factoriel d'un entier

- En mathématique, le factoriel d'un entier n (dénnoté $n!$) est le produit des entiers de 1 à n :

$$n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n$$

```
var fact = function (n) {  
    var f = 1;  
    for (var i=1; i<=n; i++) {  
        f = f*i;  
    }  
    return f;  
};  
  
var testFact = function () {  
    assert( fact(0) == 1 );  
    assert( fact(1) == 1 );  
    assert( fact(3) == 6 );  
    assert( fact(6) == 720 );  
};
```


Factoriel d'un entier

- Une définition récursive s'obtient en remarquant que le produit des $n-1$ premiers termes est $(n-1)!$:

$$n! = \underbrace{1 \times 2 \times 3 \times \dots \times (n-1)}_{(n-1)!} \times n$$

```
var fact = function (n) {  
  if (n == 0) {  
    return 1;  
  } else {  
    return fact(n-1) * n;  
  }  
};
```

```
var testFact = function () {  
  assert( fact(0) == 1 );  
  assert( fact(1) == 1 );  
  assert( fact(3) == 6 );  
  assert( fact(6) == 720 );  
};
```

cas de base de
la récursion

$$n = 0$$

Factoriel d'un entier

- Dans une fonction réursive on doit traiter :
 - Le **cas de base**, c'est-à-dire le cas qui peut se traiter sans appel récursif (c'est normalement le cas le plus simple à traiter)
 - Le **cas récursif**, c'est-à-dire celui pour lequel il faut faire un traitement similaire mais “plus petit”
- Dans le cas de la fonction factorielle :
 - **cas de base** : $n = 0$ (on sait que le résultat est 1)
 - **cas récursif** : $n > 0$ (il faut calculer $(n-1)!$ et multiplier par n)

Permutations d'un tableau

- Un tableau est un groupe d'éléments ordonnés
- Une permutation d'un tableau c'est un tableau contenant les mêmes éléments réordonnés
- Par ex., il y a 6 permutations du tableau **[1, 2, 3]** :

[1, 2, 3]

[1, 3, 2]

[2, 1, 3]

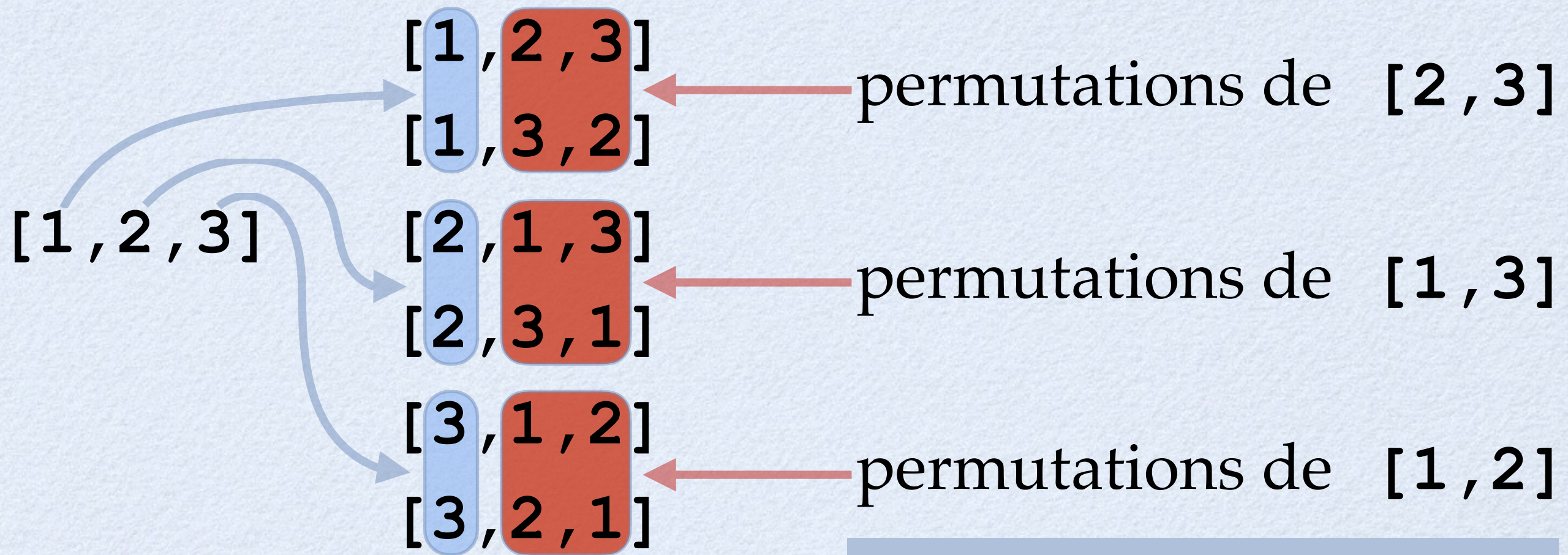
[2, 3, 1]

[3, 1, 2]

[3, 2, 1]

Permutations d'un tableau

- Les permutations ont une forme qui est reliée au tableau d'origine :



pour permuter un tableau de longueur n il faut permuter des tableaux de longueur $n-1$

Permutations d'un tableau

- **Spécification** : la fonction **perms** (*tab*) retourne un tableau des permutations de *tab*
- Par exemple, **perms** ([1, 2, 3]) retourne le tableau :

```
[ [1, 2, 3],  
  [1, 3, 2],  
  [2, 1, 3],  
  [2, 3, 1],  
  [3, 1, 2],  
  [3, 2, 1]]
```


Algorithme de permutations

- Pour chaque élément x de tab :
 - Soit $tabSansX$ une copie de tab sans l'élément x
 - Pour chaque permutation p de $tabSansX$:
 - $[x].concat(p)$ est une permutation de tab
- Quel est le **cas de base**? C'est-à-dire le cas où une récursion n'est pas nécessaire
- Lorsque tab a un seul élément (ou même zéro!) c'est la seule permutation possible

Codage de la fonction perms

```
var perms = function (tab) {  
  
    var resultat = [];  
  
    if (tab.length <= 1) {  
  
        resultat.push(tab.slice());  
  
    } else {  
  
        for (var i=0; i<tab.length; i++) {  
            var tabSansX = tab.slice(); // prendre copie de tab  
            tabSansX.splice(i,1);        // retirer élément i  
            perms(tabSansX).forEach(function (p) {  
                resultat.push([tab[i]].concat(p));  
            });  
        }  
  
    }  
  
    return resultat;  
  
};
```


Trier un tableau rapidement

- On a vu l'algorithme de **tri par sélection** et le **tri bulle** pour trier un tableau
- Rappel : le **tri par sélection** trouve le **plus petit élément**, le place au début du tableau, trouve le plus petit élément du reste du tableau, le place en 2ième position, etc


```
// Rappel :
```

```
var trier = function (t) { // tri par sélection
  for (var i=0; i<t.length-1; i++) {
    var m = positionMin(t, i);
    var temp = t[i];
    t[i] = t[m];
    t[m] = temp;
  }
};
```

```
var positionMin = function (t, debut) {
  // suppose que t.length > debut

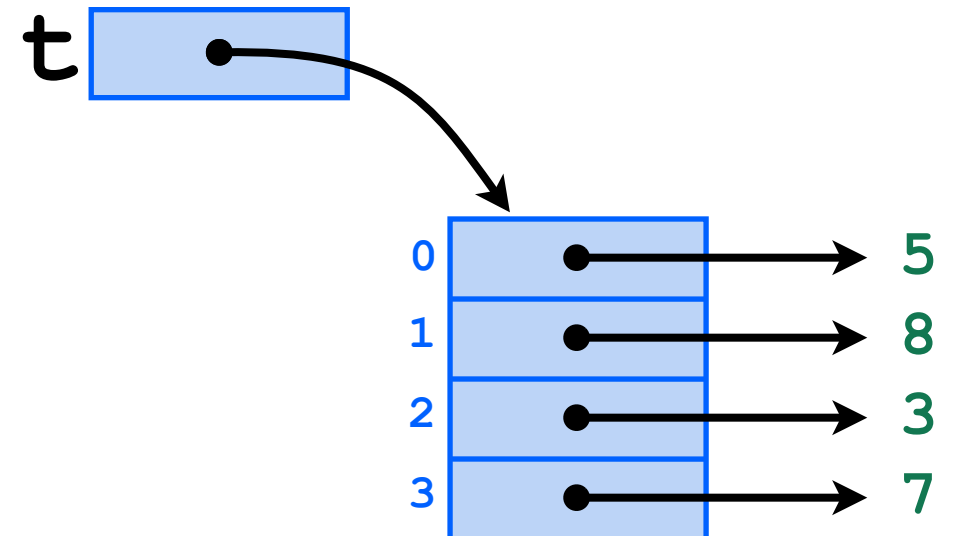
  var posMin = debut;

  for (var i=debut+1; i<t.length; i++) {
    if (t[i] < t[posMin]) {
      posMin = i;
    }
  }

  return posMin;
};
```

```
var tab = [5, 8, 3, 7];
```

```
trier(tab); print(tab); // imprime : 3,5,7,8
```



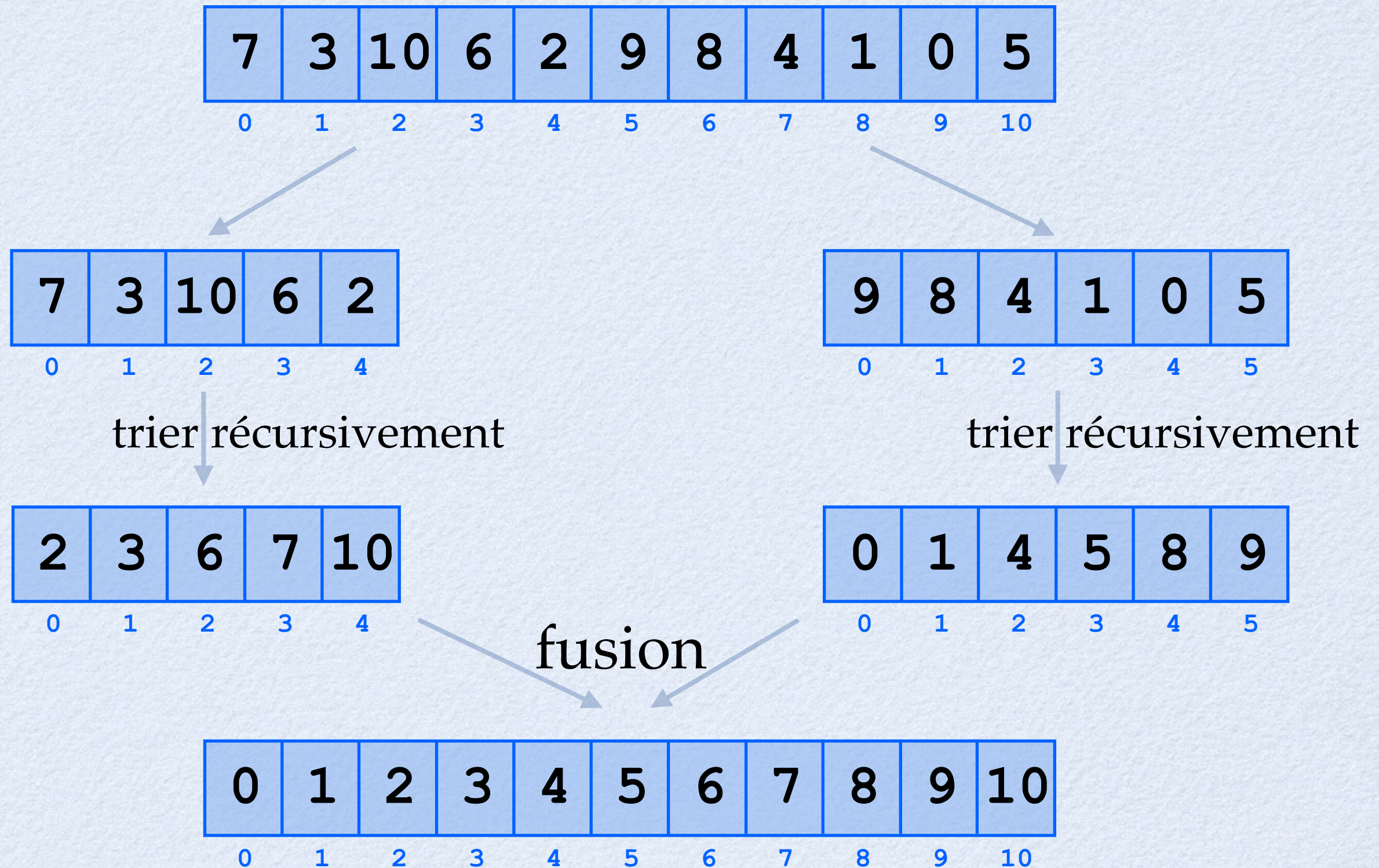
Trier un tableau rapidement

- Il y a des algorithmes récur­sifs plus rapides
- Par exemple le **tri fusion** (“mergesort”)
- Basé sur la stratégie “**diviser pour régner**” :
 - Diviser le problème principal en des sous-problèmes plus simples
 - Les résoudre **récur­sivement**
 - Combiner les solutions pour avoir la solution du problème principal

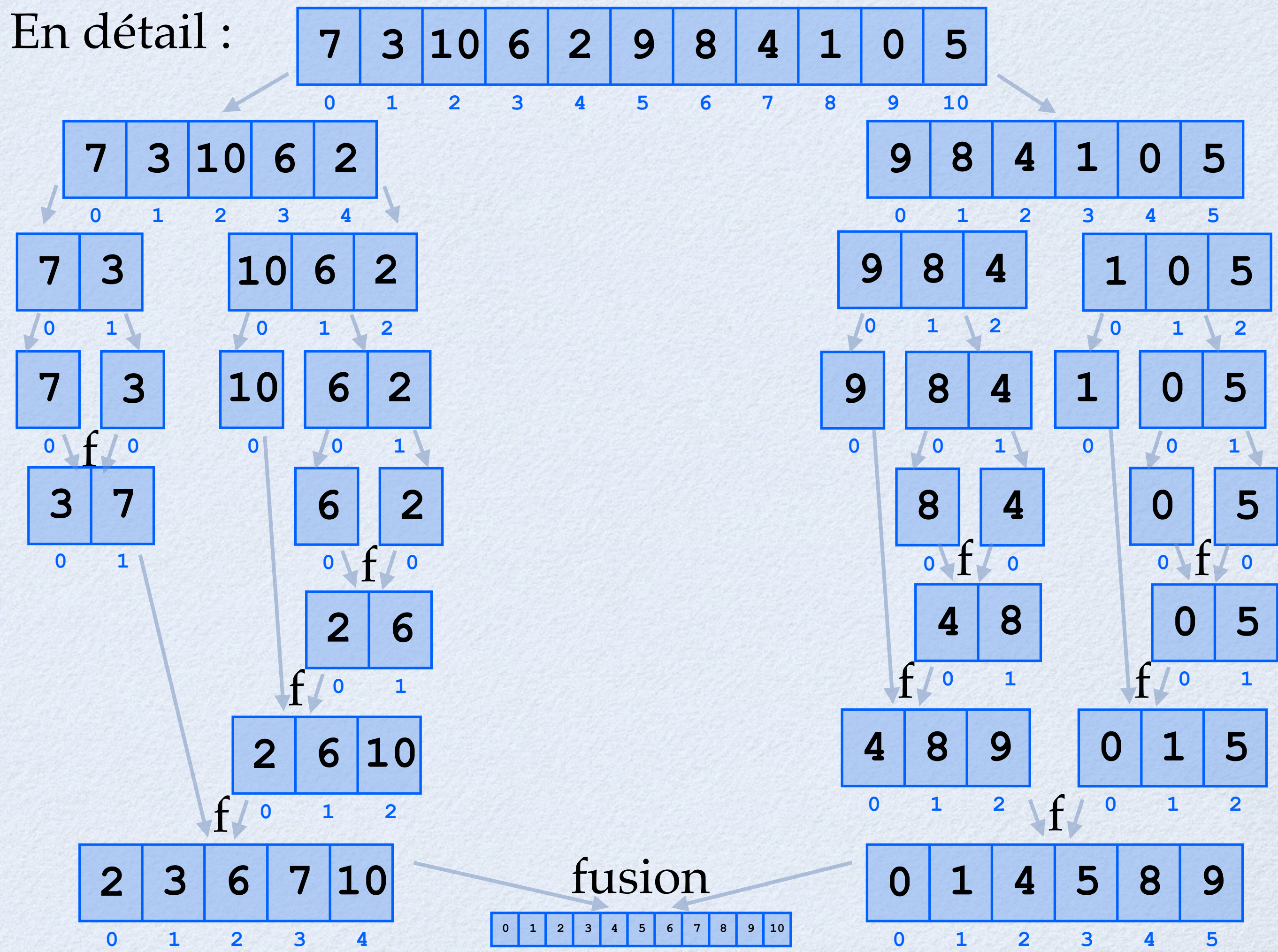
Tri fusion

- Algorithme de **tri fusion** :
 - Diviser le tableau en deux sous-tableaux
 - Les trier indépendamment **récurivement**
 - Utiliser l'algorithme de fusion pour combiner les résultats en un tableau trié
- Quel est le **cas de base**?
 - Lorsqu'un tableau contient moins de 2 éléments, il est déjà trié (rien à faire de plus!)

Tri fusion



En détail :



Tri fusion

- Rappel de la fusion de 2 listes de nombres :

```
var fusion = function (liste1, liste2) {  
  
    var resultat = [];  
    var i = 0;    // index du prochain élément de liste1  
    var j = 0;    // index du prochain élément de liste2  
  
    while (i < liste1.length && j < liste2.length) {  
        if (liste1[i] < liste2[j])  
            resultat.push(liste1[i++]);  
        else  
            resultat.push(liste2[j++]);  
    }  
  
    while (i < liste1.length) resultat.push(liste1[i++]);  
    while (j < liste2.length) resultat.push(liste2[j++]);  
  
    return resultat;  
};  
  
print( fusion([1,3,5,6], [0,4,8,9]) );
```


Tri fusion

- Tri fusion :

```
var trier = function (liste) {  
  if (liste.length < 2) {  
    return liste;  
  } else {  
    var mid = liste.length >> 1;  
    var tri1 = trier(liste.slice(0, mid));  
    var tri2 = trier(liste.slice(mid, liste.length));  
    return fusion(tri1, tri2);  
  }  
};
```

7	3	10	6	2	9	8	4	1	0	5
0	1	2	3	4	5	6	7	8	9	10

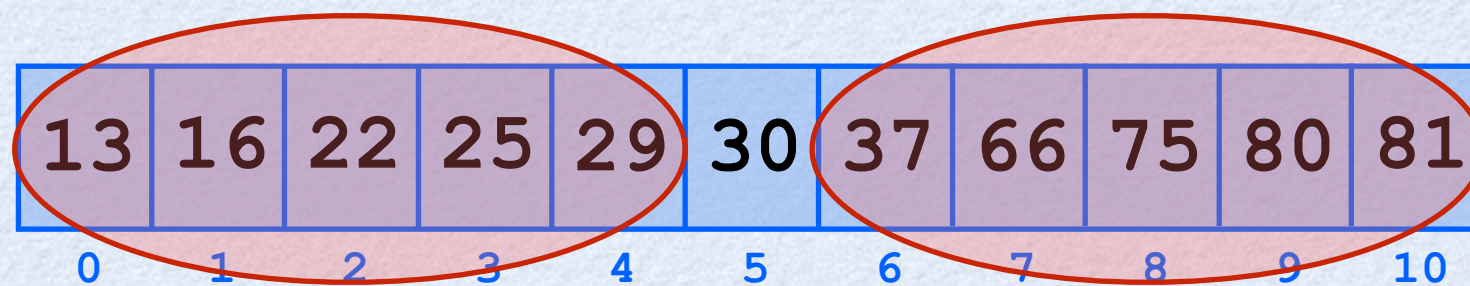
Rechercher rapidement

- On a vu l'algorithme de **recherche linéaire** pour trouver la position d'une valeur dans un tableau
- Rappel :

```
var position = function (t, val) {  
    for (var i=0; i<t.length; i++) {  
        if (t[i] == val) {  
            return i; // on a trouvé!  
        }  
    }  
  
    return -1; // code indiquant échec  
};  
  
var tab = [11, 22, 33, 44];  
  
print(position(tab, 22)); // imprime : 1  
print(position(tab, 100)); // imprime : -1
```


Recherche dichotomique

- La recherche dichotomique permet de trouver rapidement la position d'une valeur x dans un tableau trié



si $x < 30$ il doit
être dans cette
section

comparer
 x et 30

si $x > 30$ il doit
être dans cette
section

Recherche dichotomique

- Algorithme de **recherche dichotomique** :
 - Comparer x avec l'élément du milieu
 - Chercher récursivement dans la **moitié inférieure** si x est plus petit, sinon dans la **moitié supérieure**
- **Cas de base?**
 - Lorsque le tableau est de longueur 0 c'est impossible de trouver x (position = -1)
 - Lorsque x = élément du milieu, on a trouvé

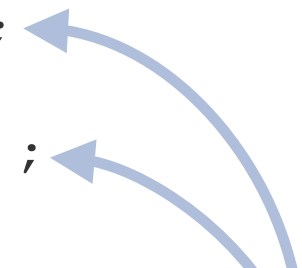
Recherche dichotomique

```
var positionFenetre = function (t, val, bas, haut) {  
  
    // si val est dans t sa position est dans  
    // l'intervalle d'index bas..haut inclusivement  
  
    if (bas <= haut) {  
        var i = (bas+haut) >> 1;  
        if (val < t[i]) {  
            return positionFenetre(t, val, bas, i-1);  
        } else if (val > t[i]) {  
            return positionFenetre(t, val, i+1, haut);  
        } else {  
            return i;  
        }  
    }  
  
    return -1; // pas trouvé  
};
```

```
var positionTrie = function (t, val) {  
    return positionFenetre(t, val, 0, t.length-1);  
};
```

```
var tab = [13,16,22,25,29,30,37,66,75,80,81];
```

```
print( positionTrie(tab, 25) );  
print( positionTrie(tab, 26) );
```



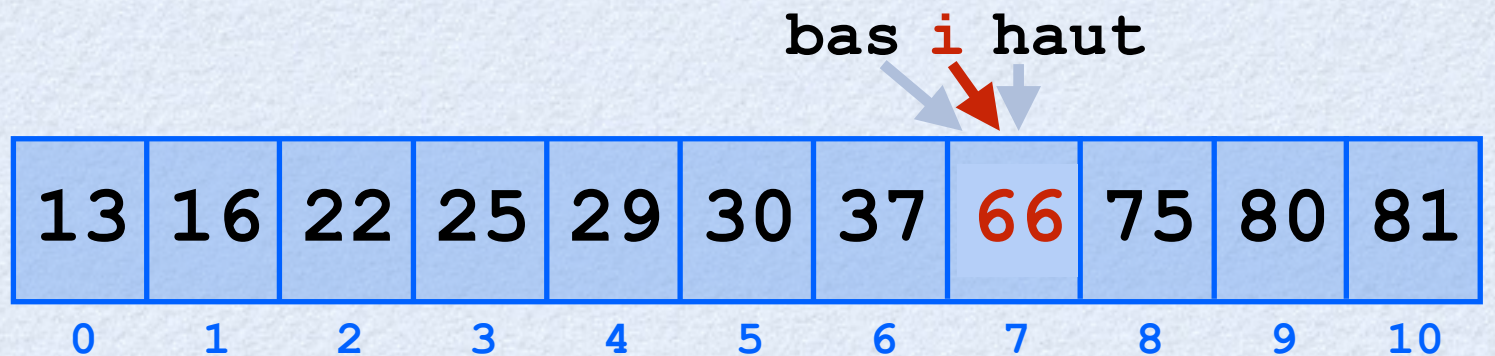
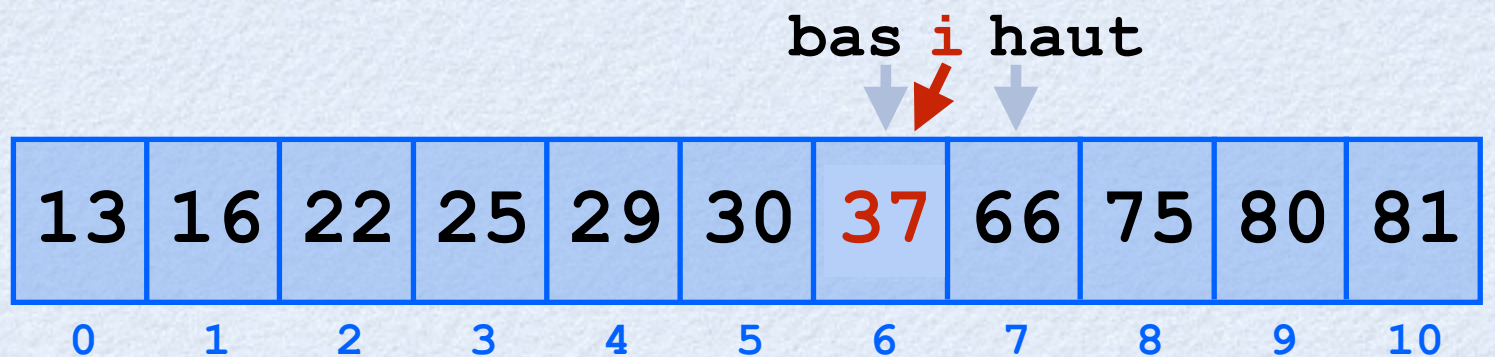
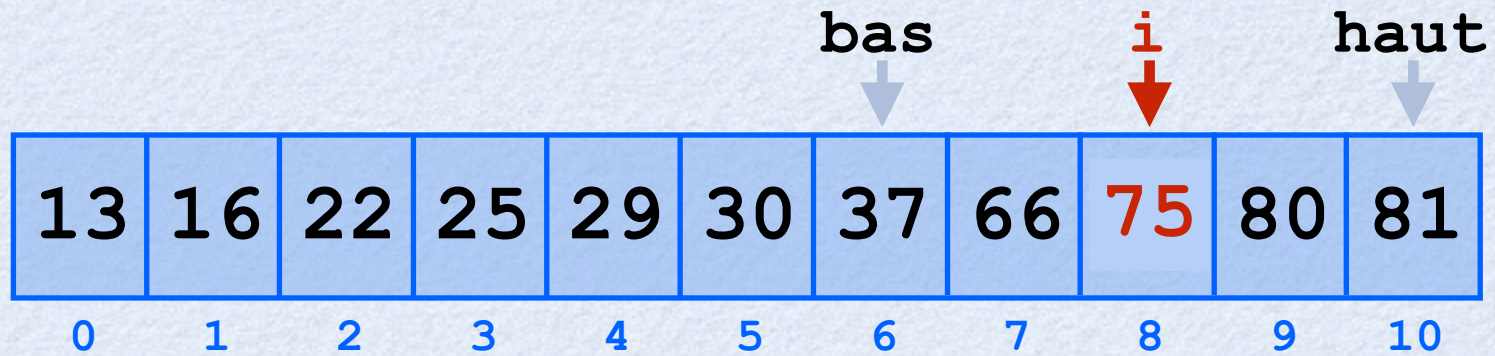
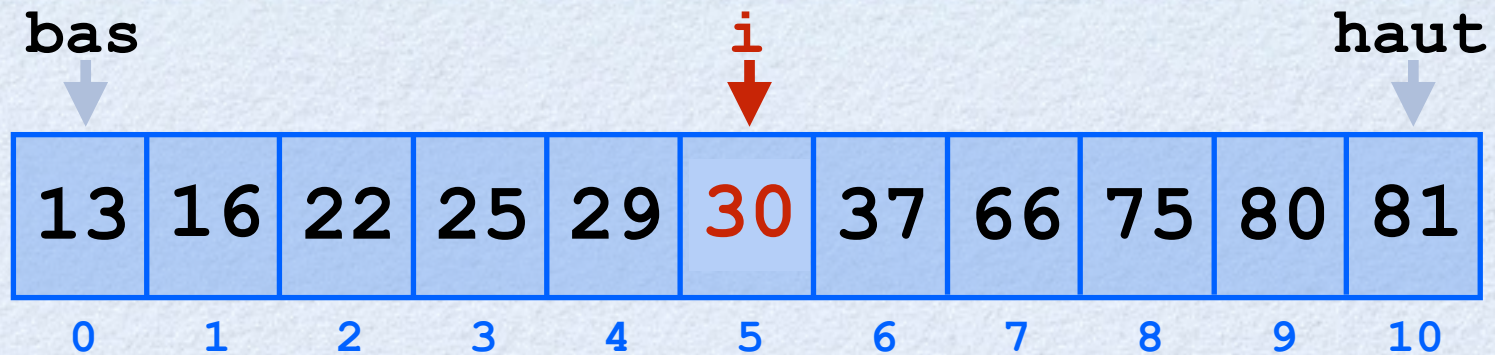
Récursions
terminales

Recherche dichotomique

```
var positionFenetre = function (t, val, bas, haut) {  
  
    // si val est dans t sa position est dans  
    // l'intervalle d'index bas..haut inclusivement  
  
    while (bas <= haut) {  
        var i = (bas+haut) >> 1;  
        if (val < t[i]) {  
            haut = i-1;  
        } else if (val > t[i]) {  
            bas = i+1;  
        } else {  
            return i;  
        }  
    }  
  
    return -1; // pas trouvé  
};  
  
var positionTrie = function (t, val) {  
    return positionFenetre(t, val, 0, t.length-1);  
};  
  
var tab = [13,16,22,25,29,30,37,66,75,80,81];  
  
print( positionTrie(tab, 25) );  
print( positionTrie(tab, 26) );
```

Remplacement
des récursions
terminales par
des boucles

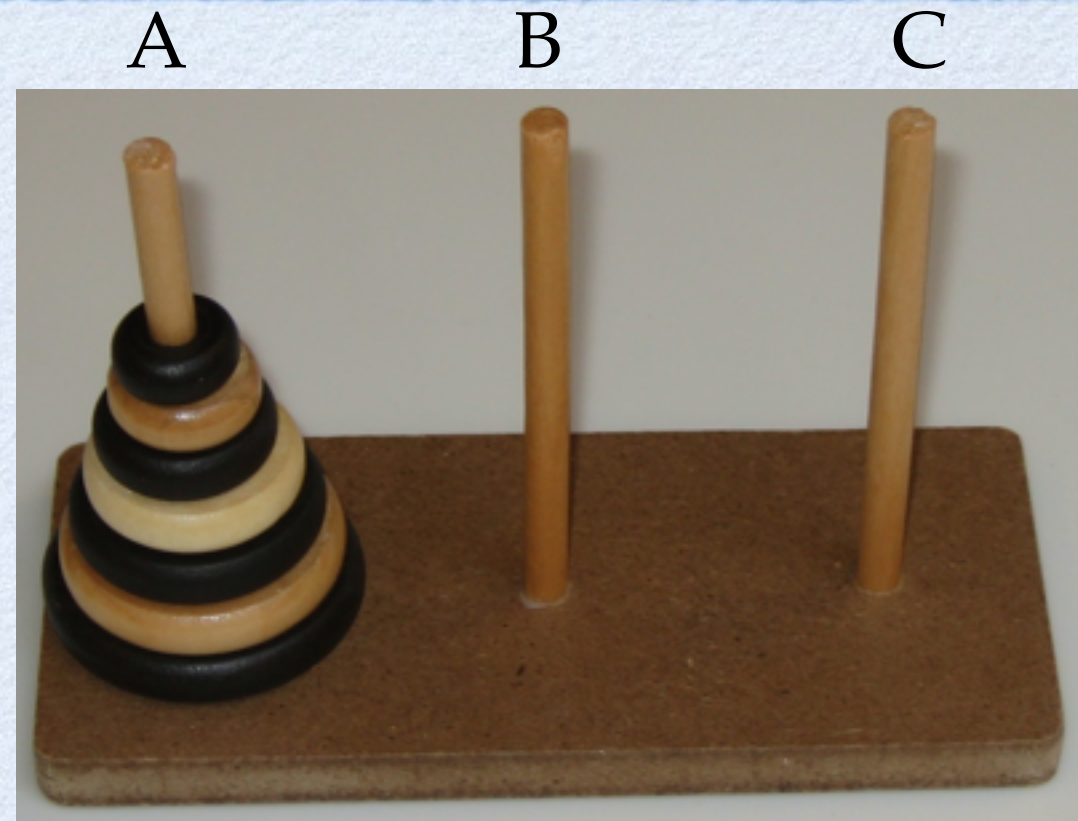
Chercher 66



Trouvé!
position = 7

Puzzle des tours de Hanoi

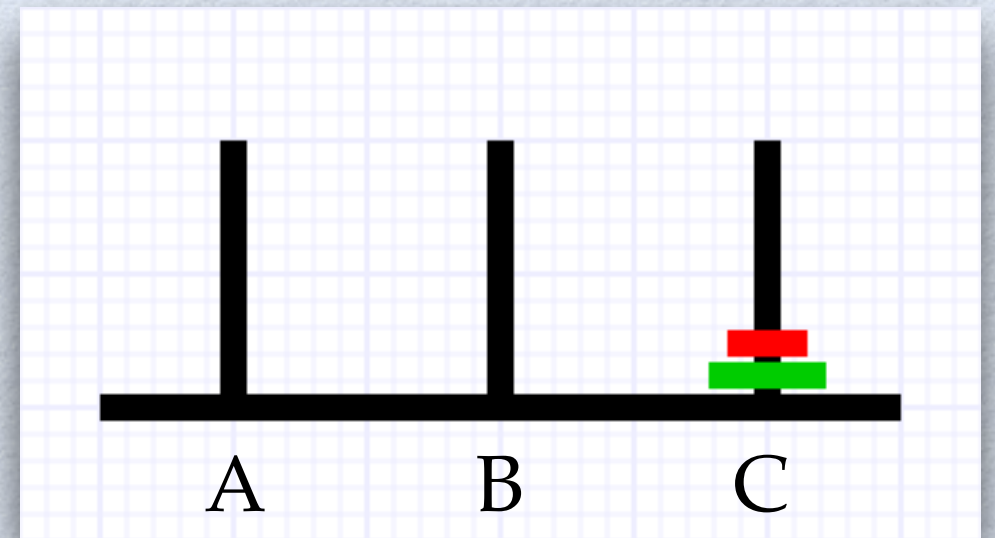
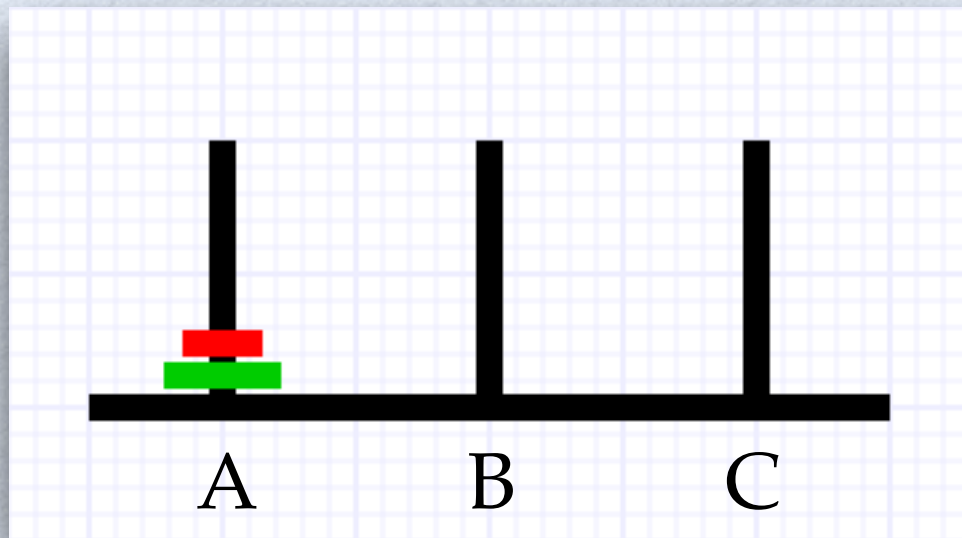
- Trois tiges A, B et C :



- La tige A contient une tour de n disques troués ordonnés du plus grand en bas au plus petit en haut
- Objectif : déplacer la tour de la tige A à C en respectant :
 - Un seul disque déplacé à la fois d'une tige à une autre tige
 - Il ne faut pas empiler un disque sur un plus petit

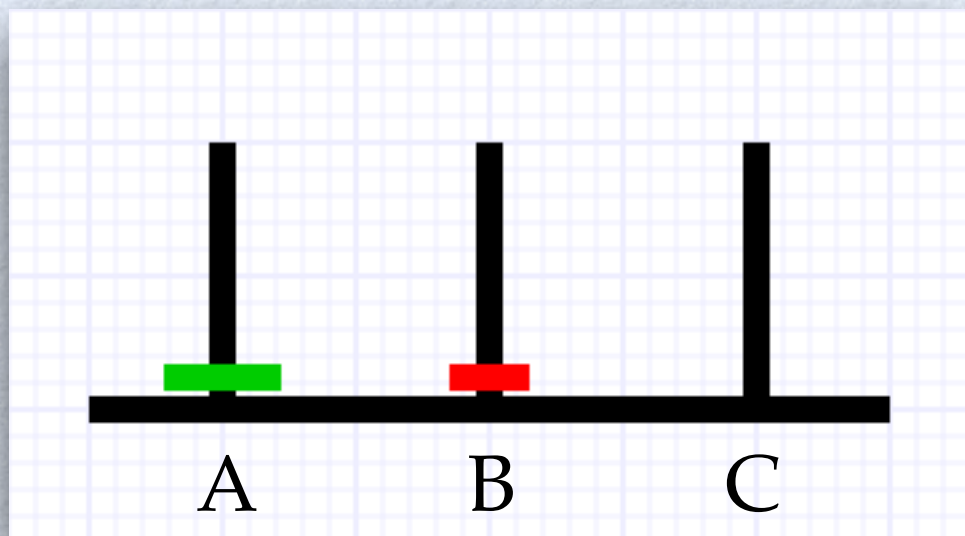
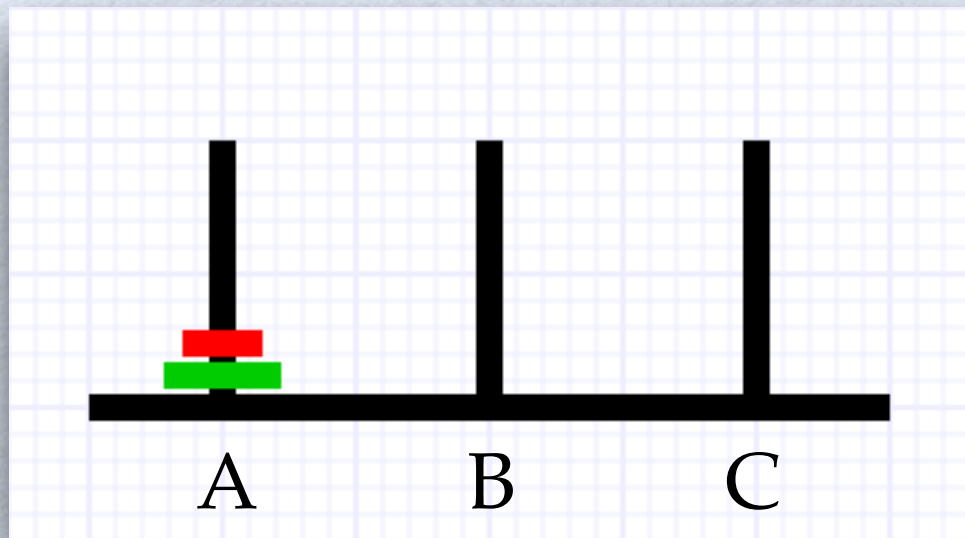
Puzzle des tours de Hanoi

- Exemple simple, pour $n = 2$:

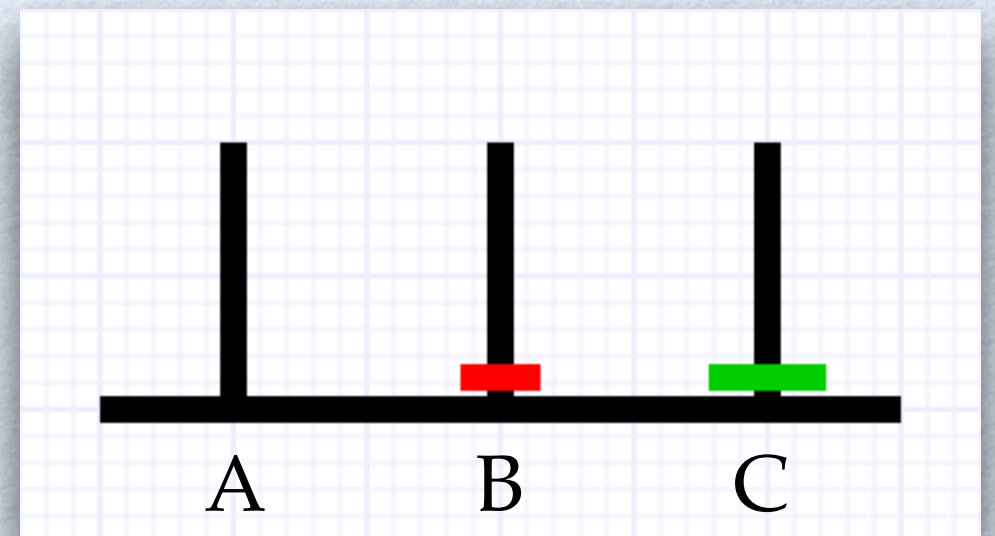
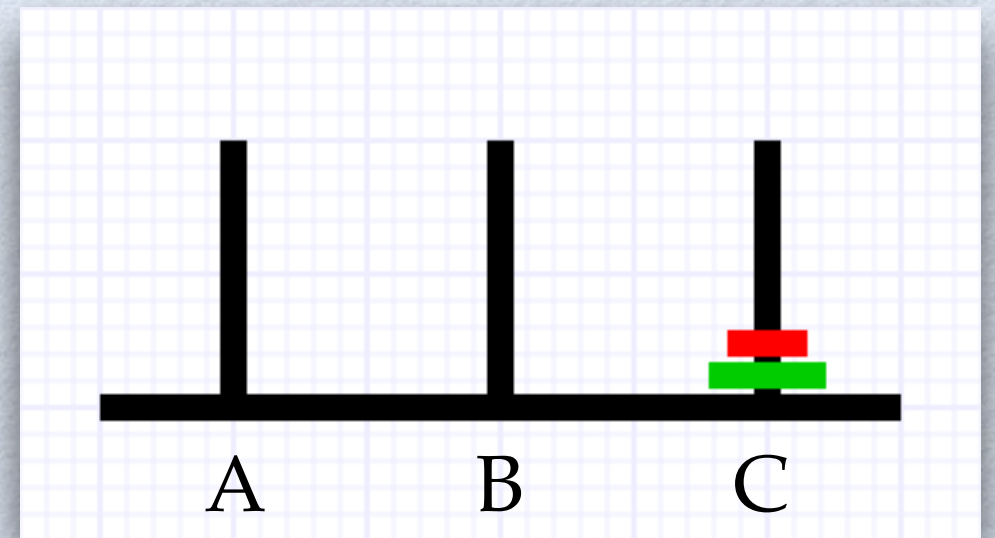


Puzzle des tours de Hanoi

- Exemple simple, pour $n = 2$:

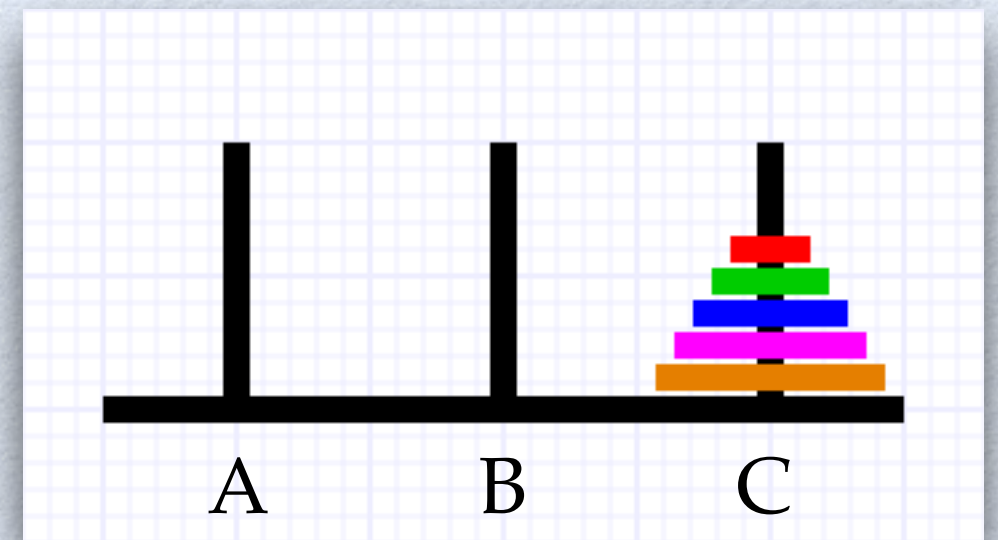
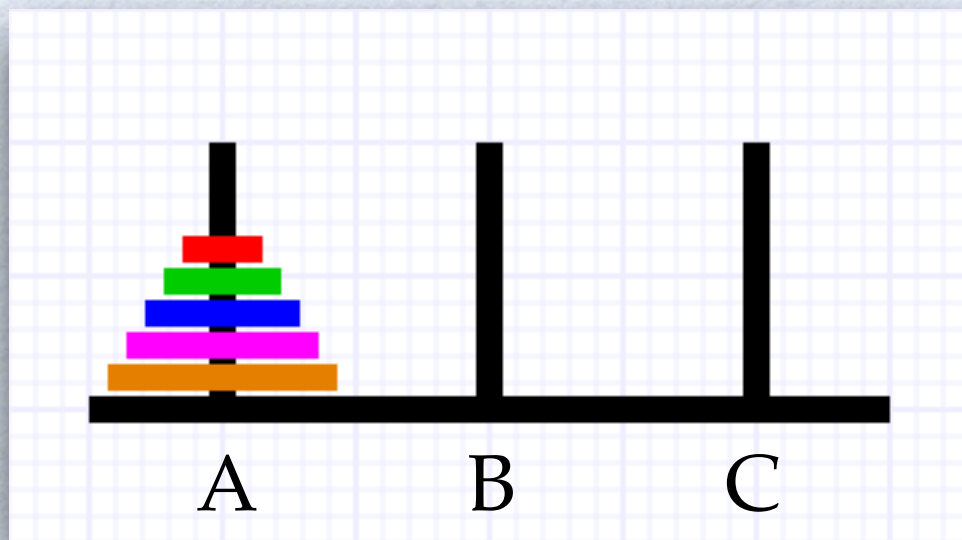


A à C

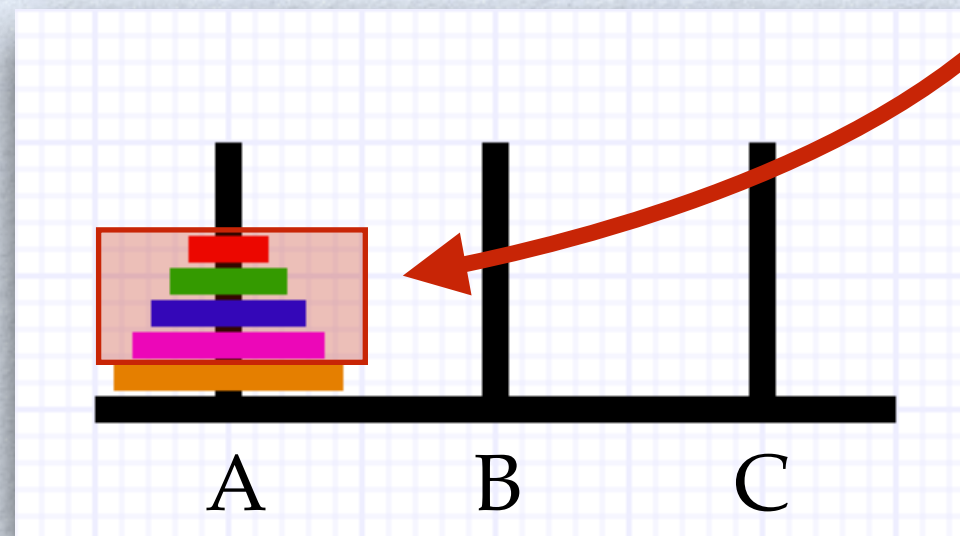


Puzzle des tours de Hanoi

- Comment faire pour un plus grand n ? Par exemple $n = 5$:

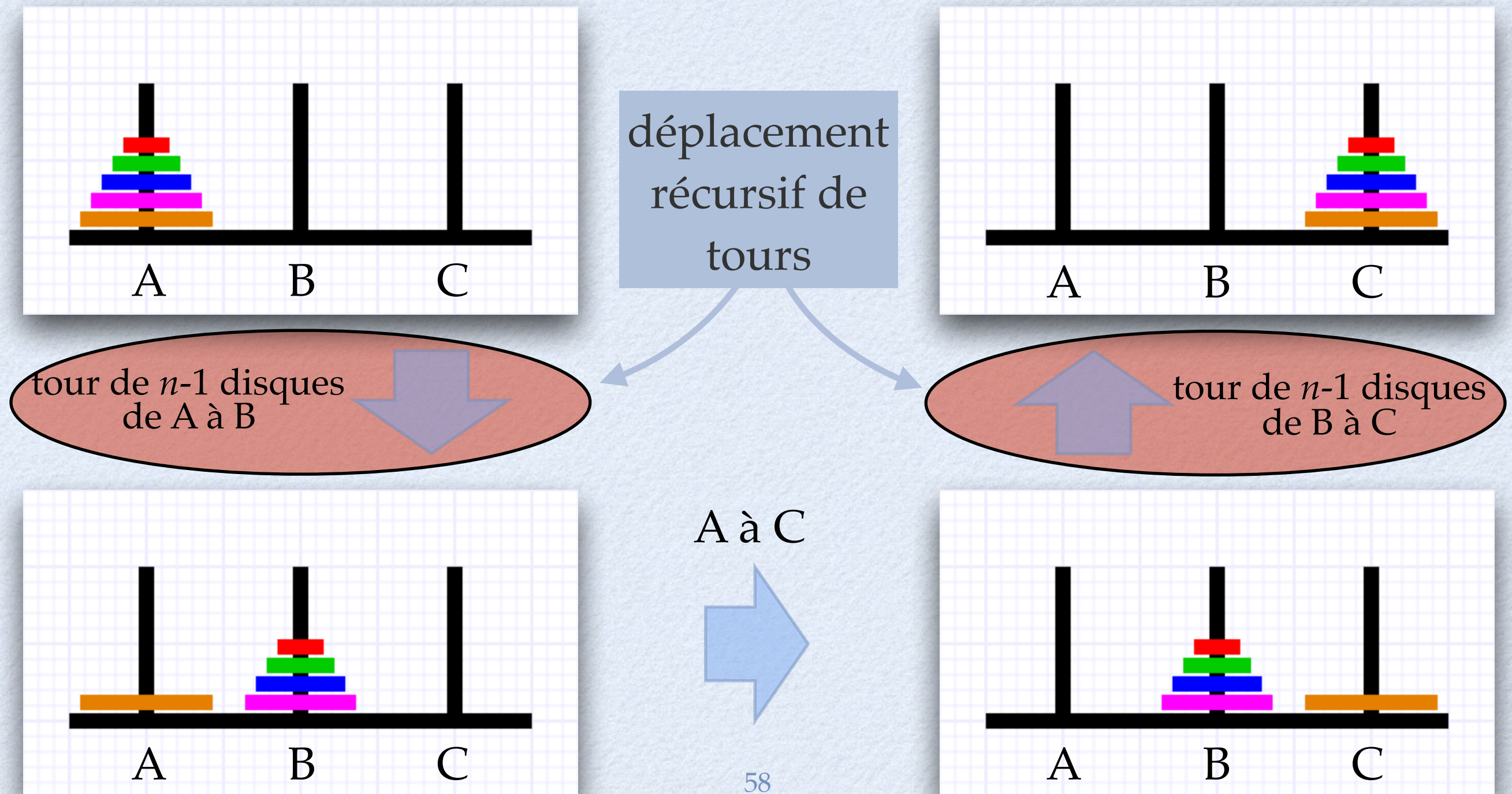


- Idée : une tour de n disques, c'est une **tour de $n-1$ disques** au dessus d'un disque de taille n



Puzzle des tours de Hanoi

- Supposer qu'on sait déplacer une tour de $n-1$ disques :



Puzzle des tours de Hanoi

- Cas de base?
- On pourrait utiliser le cas $n=1$
 - Une tour de taille $n=1$ c'est **un seul disque**, donc on a simplement à le déplacer de la tige source à la destination
- C'est encore plus simple d'utiliser le cas $n=0$
 - Une tour de taille $n=0$ c'est **rien**, donc on a rien à faire pour la déplacer!

Puzzle des tours de Hanoi

- Algorithme récursif pour déplacer une tour de n disques d'une tige **source** à une tige **destination** :

```
var deplacerDisque = function (source, destination) {  
    print(source + " à " + destination);  
};  
  
var deplacerTour = function (n, source, destination) {  
    if (n > 0) {  
        var autre;  
        if (source != "A" && destination != "A") {  
            autre = "A";  
        } else if (source != "B" && destination != "B") {  
            autre = "B";  
        } else {  
            autre = "C";  
        }  
        deplacerTour(n-1, source, autre);  
        deplacerDisque(source, destination);  
        deplacerTour(n-1, autre, destination);  
    }  
};  
  
deplacerTour(5, "A", "C");
```

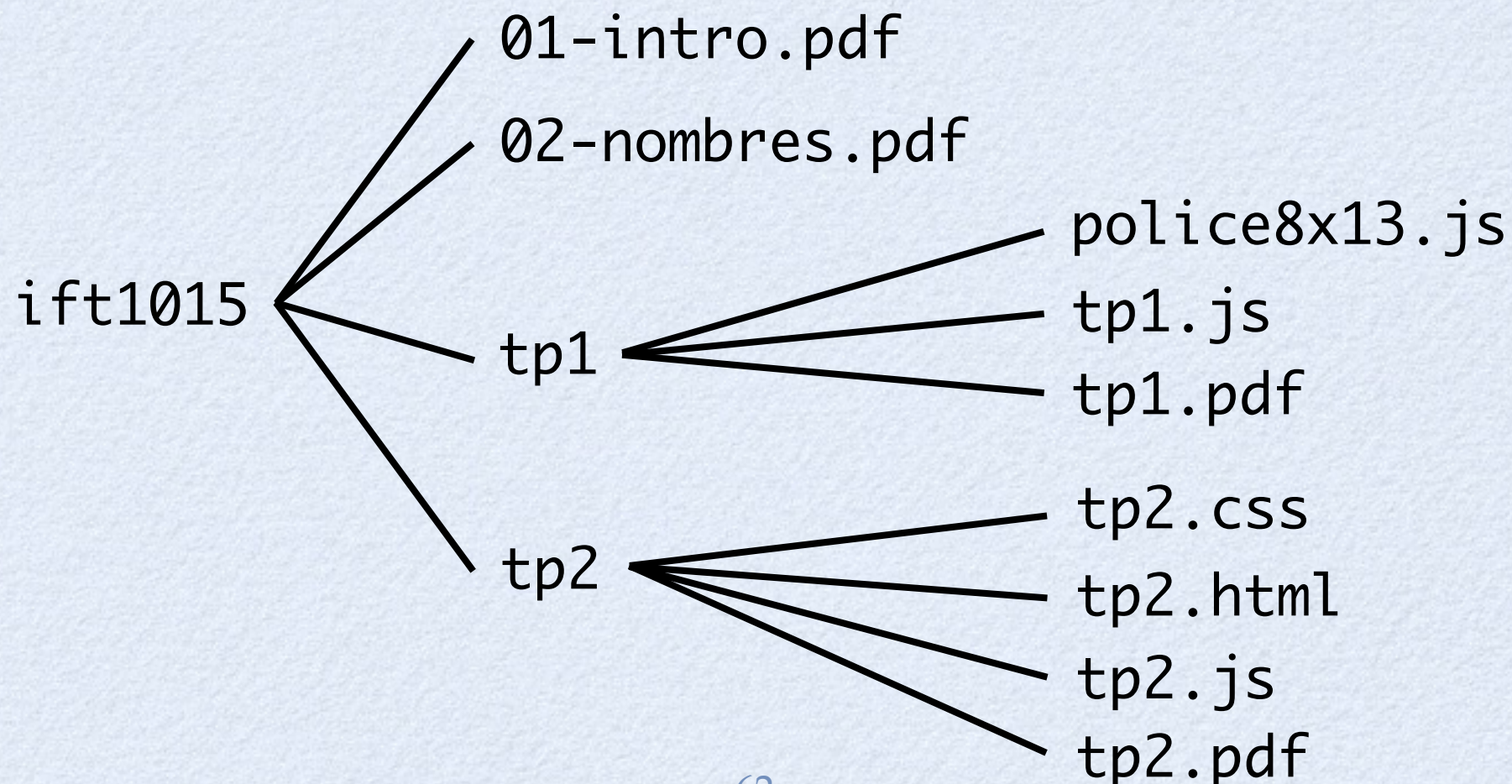

Puzzle des tours de Hanoi

- Algorithme qui ne recalcule pas l'autre tige à chaque fois :

```
var deplacerDisque = function (source, destination) {  
    print(source + " à " + destination);  
};  
  
var deplacerTour = function (n, source, destination, autre) {  
    if (n > 0) {  
        deplacerTour(n-1, source, autre, destination);  
        deplacerDisque(source, destination);  
        deplacerTour(n-1, autre, destination, source);  
    }  
};  
  
deplacerTour(5, "A", "C", "B");
```

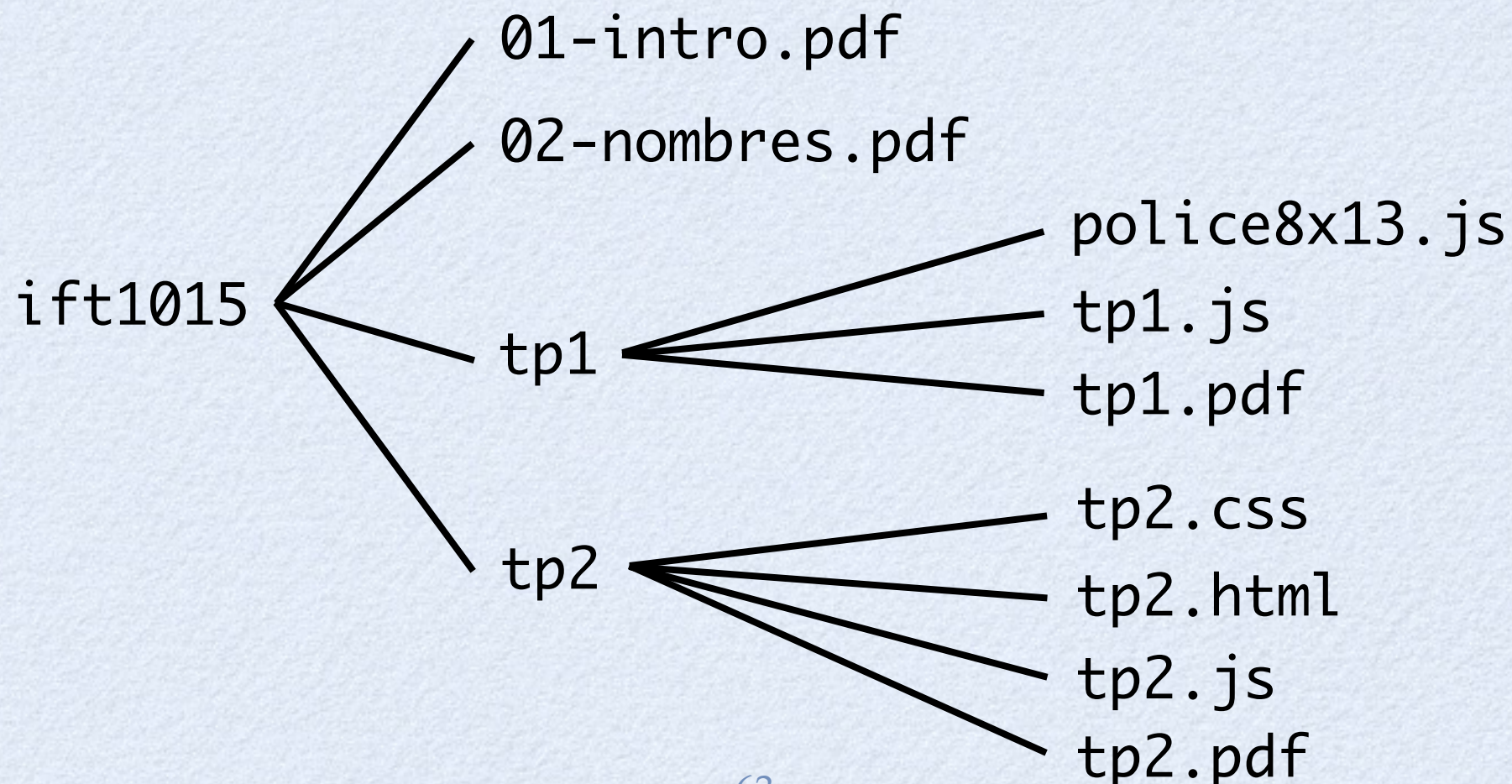

Système de fichier

- Un système de fichier hiérarchique a une structure d'arbre
- Chaque **répertoire** peut contenir des **fichiers** et des **répertoires**
- C'est donc une structure qui est **récursive**



Système de fichier

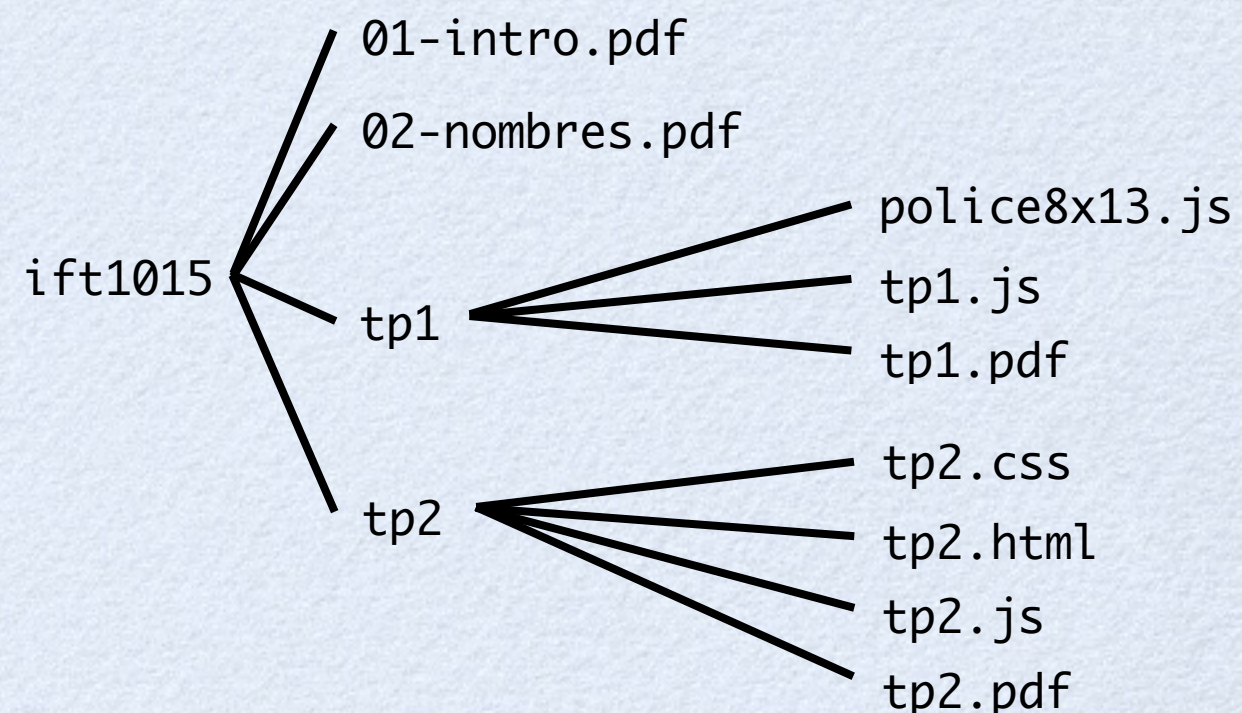
- Avec la récursivité on peut parcourir le système de fichier pour visiter chaque fichier qu'il contient
- **Applications** : lister tous les fichiers, chercher un fichier spécifique, copier tous les fichiers, etc



readdirSync

- Les fonctions suivantes de **nodejs** sont utiles pour consulter le contenu du système de fichiers
- **fs.readdirSync** (*path*) retourne un tableau de tous les items contenus directement dans le répertoire *path* :

```
> fs.readdirSync("ift1015");  
["01-intro.pdf",  
 "02-nombres.pdf",  
 "tp1",  
 "tp2"]  
  
> fs.readdirSync("ift1015/tp1");  
["police8x13.js",  
 "tp1.js",  
 "tp1.pdf"]
```



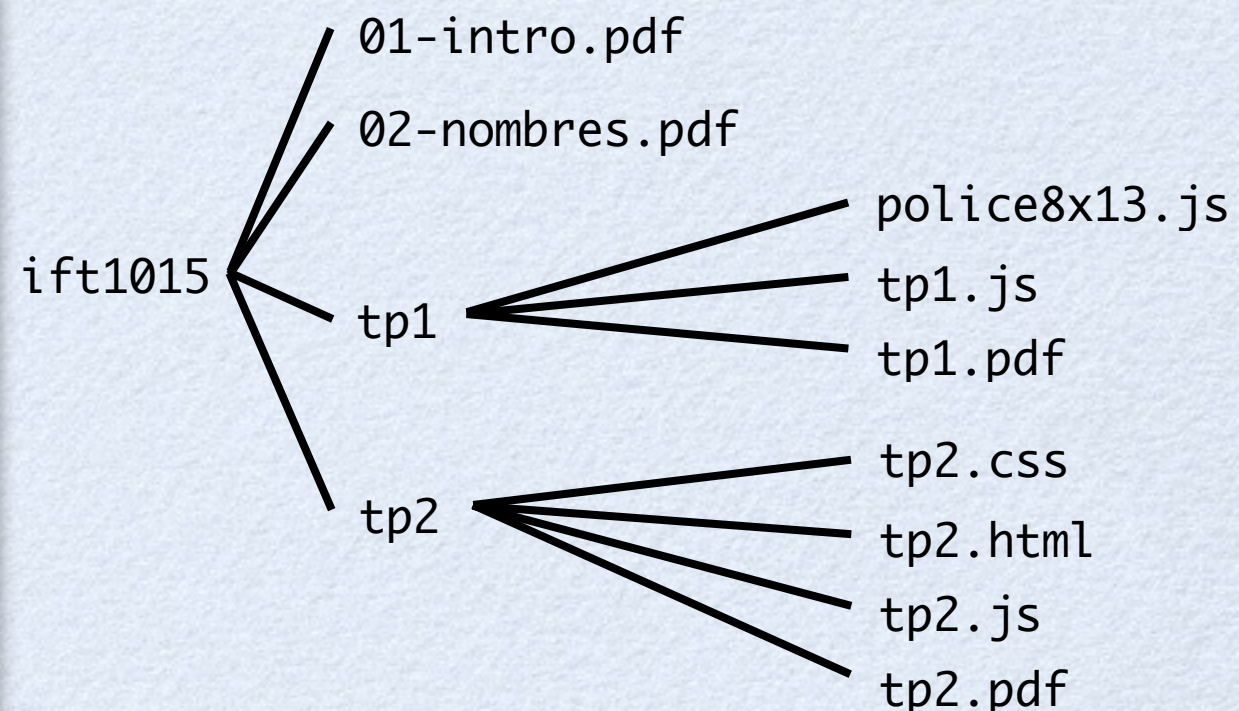
statSync et isDirectory

- **`fs.statSync(path)`** retourne une structure donnant des informations sur le fichier *path*
- **`fs.statSync(path).isDirectory()`** retourne un Booléen indiquant si *path* est un répertoire

```
> fs.statSync("ift1015/tp1");
{dev: 16777220,
  mode: 16877,
  nlink: 5,
  uid: 501,
  gid: 20,
  rdev: 0,
  blksize: 4096,
  ino: 72450745,
  size: 170,
  blocks: 0,
  atime: Sat Apr 09 2016 21:30:48 GMT-0400 (EDT),
  mtime: Sat Apr 09 2016 21:28:14 GMT-0400 (EDT),
  ctime: Sat Apr 09 2016 21:28:14 GMT-0400 (EDT),
  birthtime: Sat Apr 09 2016 21:27:48 GMT-0400 (EDT)}

> fs.statSync("ift1015/tp1").isDirectory();
true

> fs.statSync("ift1015/tp1/tp1.pdf").isDirectory();
false
```



Lister les fichiers récursivement

```
var fs = require("fs");

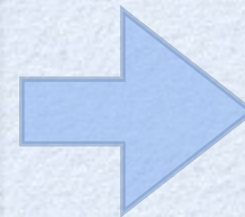
var listerFichiers = function (path) {
  var noms = fs.readdirSync(path);

  noms.forEach(function (nom) {

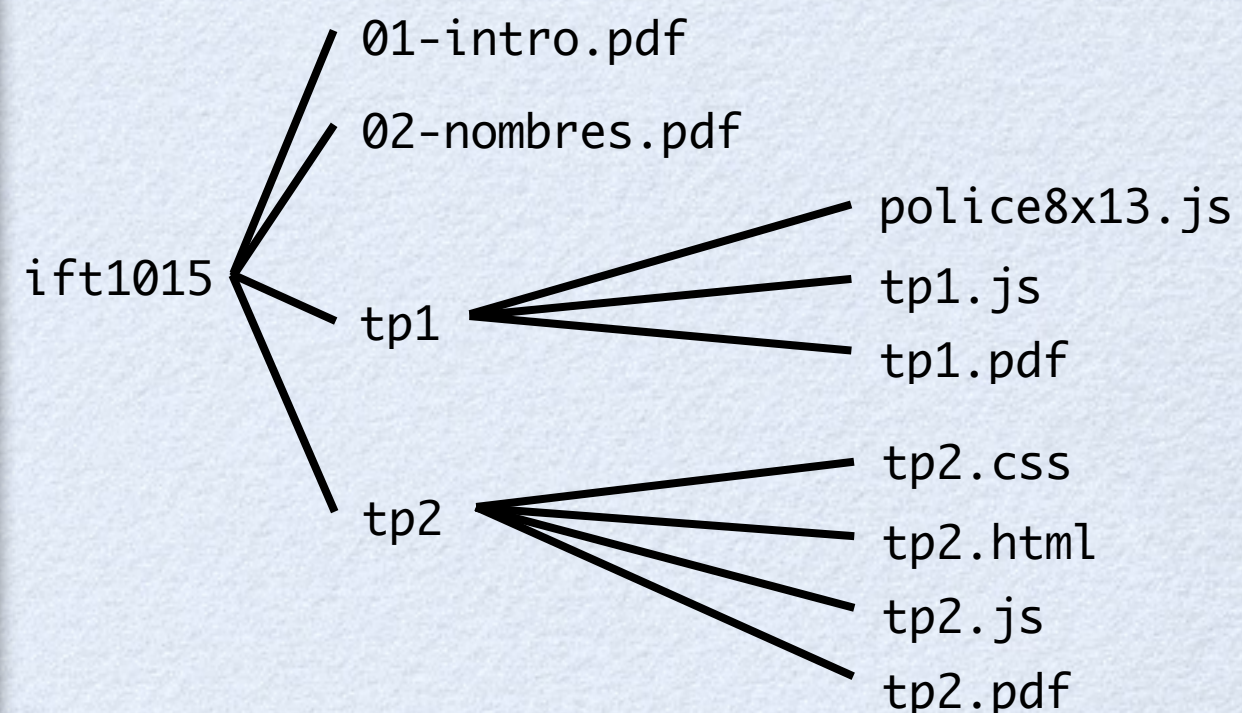
    var p = path + "/" + nom;

    if (fs.statSync(p).isDirectory()) {
      console.log(p + "/");
      listerFichiers(p);
    } else {
      console.log(p);
    }
  });
};

listerFichiers("ift1015");
```



```
ift1015/01-intro.pdf
ift1015/02-nombres.pdf
ift1015/tp1/
ift1015/tp1/police8x13.js
ift1015/tp1/tp1.js
ift1015/tp1/tp1.pdf
ift1015/tp2/
ift1015/tp2/tp2.css
ift1015/tp2/tp2.html
ift1015/tp2/tp2.js
ift1015/tp2/tp2.pdf
```



Chercher des fichiers d'un certain type

```
var fs = require("fs");

var listerFichiersDeType = function (path, ext) {

  var noms = fs.readdirSync(path);

  noms.forEach(function (nom) {

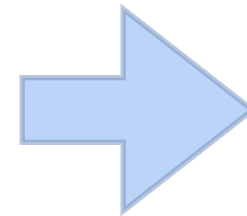
    var p = path + "/" + nom;

    if (fs.statSync(p).isDirectory()) {
      listerFichiersDeType(p, ext);
    } else if (p.slice(-ext.length) == ext) {
      console.log(p);
    }

  });

};

listerFichiersDeType("ift1015", ".pdf");
```



```
ift1015/01-intro.pdf
ift1015/02-nombres.pdf
ift1015/tp1/tp1.pdf
ift1015/tp2/tp2.pdf
```


Parcours générique des fichiers

```
var fs = require("fs");

var pourChaqueFichier = function (path, f) {

  var noms = fs.readdirSync(path);

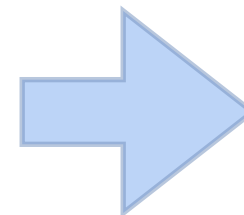
  noms.forEach(function (nom) {

    var p = path + "/" + nom;

    if (fs.statSync(p).isDirectory()) {
      pourChaqueFichier(p, f);
    } else {
      f(p);
    }

  });
};

pourChaqueFichier("ift1015", function (path) {
  if (path.slice(-4) == ".pdf") {
    console.log(path);
  }
});
```



```
ift1015/01-intro.pdf
ift1015/02-nombres.pdf
ift1015/tp1/tp1.pdf
ift1015/tp2/tp2.pdf
```


Jeu de tic-tac-toe

- La récursivité est particulièrement utile pour les **problèmes d'optimisation combinatoire**, par exemple déterminer le meilleur coup pour un joueur au tic-tac-toe
- On cherche à trouver le **meilleur coup** d'un joueur qui mènera au meilleur résultat **étant donné un adversaire qui lui aussi choisira son meilleur coup comme réplique**

Pour cette configuration, quel est le meilleur coup pour O?
4, 5, 7 ou 8?

O	X	X
X		
O		

Quel est le
meilleur coup
pour O?
4, 5, 7 ou 8?

○	×	×
×		
○		

4

○	×	×
×	○	
○		

5

○	×	×
×	○	×
○		

7

8

8



On peut simuler le reste de la
partie pour chaque coup et
réplique possible pour déterminer
le résultat final pour O

Si on simule tous
les coups et
répliques

○	×	×
×		
○		

4 5 7 8

○	×	×
×	○	
○		

○	×	×
×		○
○		

○	×	×
×		
○	○	

○	×	×
×		
○		○

5 7 8

4 7 8

4 5 8

4 5 7

○	×	×
×	○	×
○		

○	×	×
×	○	
○	×	

○	×	×
×	○	
○		×

○	×	×
×	×	○
○		

○	×	×
×		○
○	×	

○	×	×
×		○
○		×

○	×	×
×	×	
○	○	

○	×	×
×		×
○	○	

○	×	×
×		
○	○	×

○	×	×
×	×	
○		○

○	×	×
×		×
○		○

○	×	×
×		
○	×	○

7 8 5 7

5 8

5 7

7 8

4 8

4 7

5 8

4 8

4 5

5 7

4 7

4 5

8 8 7 5

8 7

8 4

7 4

8 8

8 5

5 4

7 7

4 7

4 4

8 8 7 5

8 7

8 4

7 4

8 8

8 5

5 4

7 7

4 7

4 4

○	×	×
×		
○		

O peut forcer une victoire s'il joue 8

4 5 7 8

○	×	×
×	○	
○		

○	×	×
×		○
○		

○	×	×
×		
○	○	

○	×	×
×		
○		○

5 7 8

4 7 8

4 5 8

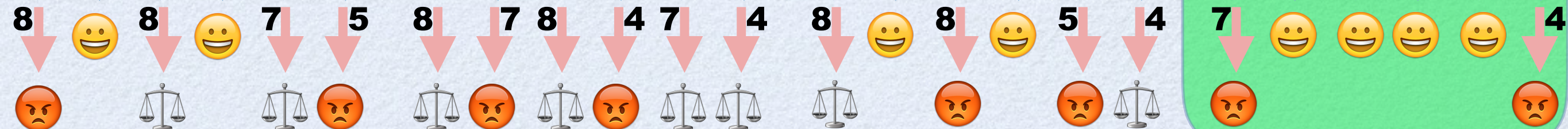
4 5 7

○	×	×
×	○	×
○		

○	×	×
×	×	○
○		

○	×	×
×	×	
○	○	

○	×	×
×	×	
○		○



Evaluation des configurations

- On peut associer à chaque configuration la valeur **+1**, **0** ou **-1** en fonction du résultat final obtenu par le joueur qui vient de jouer **étant donné un jeu parfait par le joueur et l'adversaire** (meilleur coup joué jusqu'à la fin)
 - +1** = 😊 le joueur qui a joué a ou va **gagner**
 - 0** = ⚖️ la partie est ou sera **nulle**
 - 1** = 😡 le joueur qui a joué a ou va **perdre**
- La valeur d'une configuration se calcule facilement lorsqu'il y a 3 **X** ou **O** enlignés (valeur **+1**) et lorsque les 9 cases sont occupées (valeur **0**)

Evaluation des configurations

- Sinon, le joueur adverse va choisir le coup qui maximisera la valeur de la configuration résultante (pour lui), et donc qui minimisera la valeur pour le joueur qui vient de jouer :
 - Si **un des coups de l'adversaire** a une configuration résultante de valeur **+1**, la configuration de départ a une valeur **-1** pour le joueur qui vient de jouer
 - Si **tous les coups de l'adversaire** ont une configuration résultante de valeur **-1**, la configuration de départ a une valeur **+1** pour le joueur qui vient de jouer
- Ça revient à la **négation du maximum** des valeurs des configurations résultantes pour tous les coups possibles

Calcul de la valeur des configurations

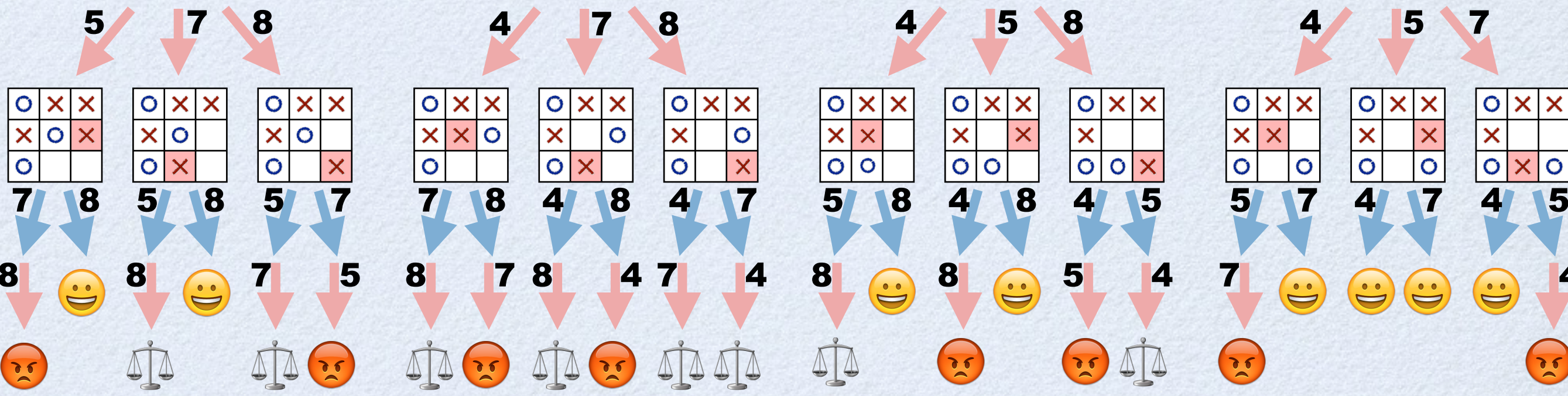
		4
	7	8



		5
	7	8

	7	8

		5
	7	

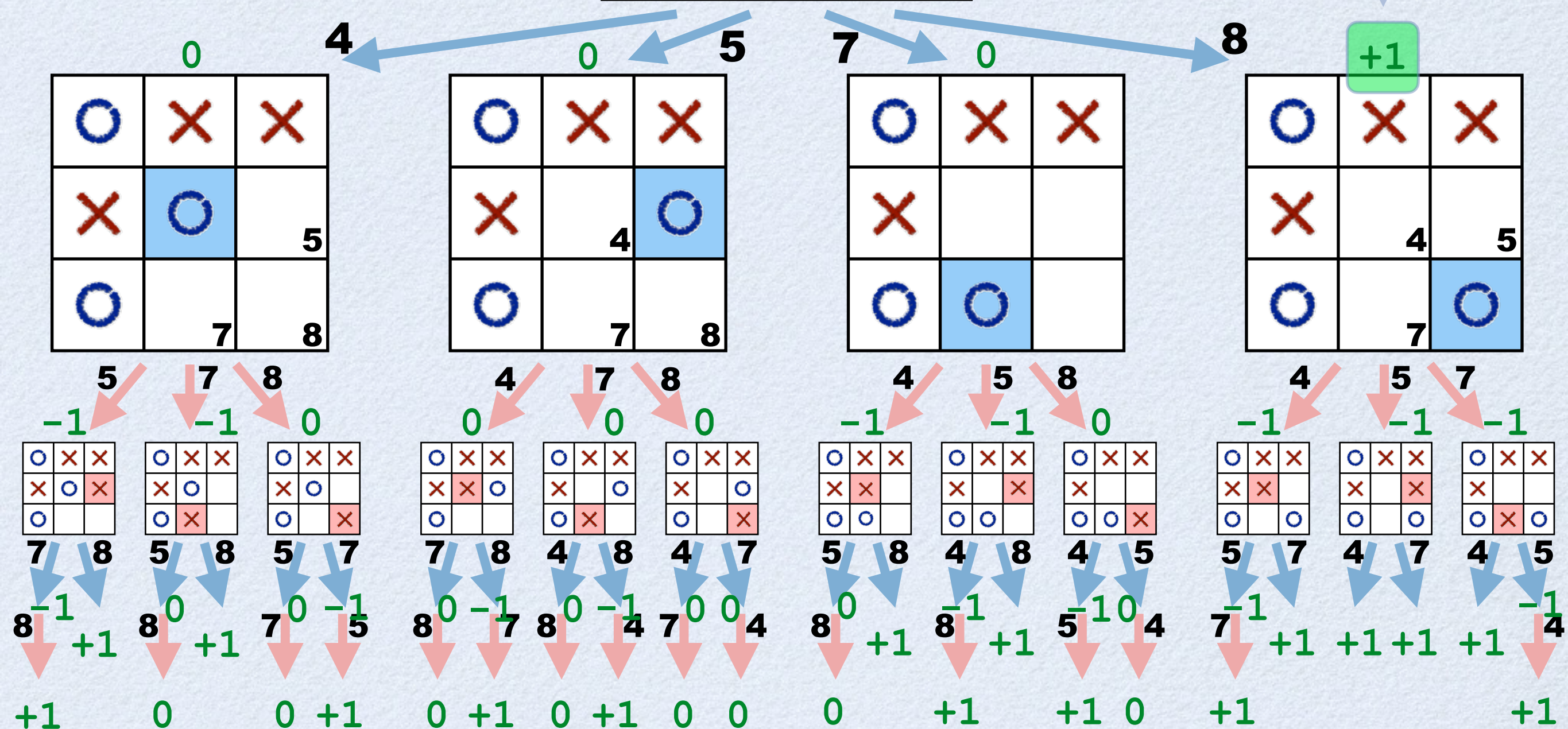


Calcul de la
valeur des
configurations

○	×	×
×		
○		

4 5
7 8

Meilleur
coup pour ○



Evaluation des configurations

- L'évaluation d'une configuration non-finale demande donc de faire l'évaluation des configurations résultant de chaque coup possible à partir de cette configuration
- C'est donc un calcul **récuratif**
- **Cas de base?** Configuration avec 3 **X** ou **O** enlignés ou 9 cases occupées
- On utilisera 2 fonctions :
 - **meilleurCoup()** retourne la position de la case qui est le meilleur coup pour le joueur à qui c'est le tour
 - **valeurCoup(*pos*)** retourne la valeur de la configuration obtenue en jouant à la case *pos*

clic

```
var clic = function (id) {  
  
    if (grille[id] == 0) {  
  
        grille[id] = tour;  
        document.getElementById(id).innerHTML = symboleJoueur(tour);  
  
        coups = coups.slice(0, occupees);  
        coups.push(id);  
  
        document.getElementById("undo").style.visibility = "visible";  
        document.getElementById("redo").style.visibility = "hidden";  
  
        var gagnant = victoire();  
        if (gagnant != 0) {  
            alert(nomJoueur(gagnant) + " est le gagnant!");  
            init();  
        } else {  
            tour = autreJoueur(tour);  
            if (++occupees == 9) {  
                alert("match nul!");  
                init();  
            } else if (tour == 2) {  
                clic(meilleurCoup());  
            }  
        }  
    }  
};
```

ajouté (noter que
clic est récurive)

meilleurCoup

```
var meilleurCoup = function () {  
  
    var valMax = -2;  
    var posMax;  
  
    for (var pos=0; pos<9; pos++) {  
        if (grille[pos] == 0) {  
            var val = valeurCoup(pos);  
            if (val > valMax) {  
                valMax = val;  
                posMax = pos;  
            }  
        }  
    }  
  
    return posMax;  
};
```


valeurCoup

```
var valeurCoup = function (pos) {  
  
    var val;  
  
    grille[pos] = tour;  
    occupees++;  
  
    if (victoire() != 0) {  
        val = +1; // victoire du joueur qui vient de jouer  
    } else if (occupees == 9) {  
        val = 0; // partie nulle  
    } else {  
        tour = autreJoueur(tour);  
        val = -valeurCoup(meilleurCoup());  
        tour = autreJoueur(tour);  
    }  
  
    grille[pos] = 0;  
    occupees--;  
  
    return val;  
};
```

réursion directe et
réursion mutuelle

meilleurCoup

valeurCoup

