# Récursivité

IFT1015 — Philippe Langlais

# Plan

- Factorielle
- Afficher une chaîne sans boucle
- Fibonacci
- Palindromes — code dans *recursion.js*
- Quick sort — code dans *quicksort.js*
- Cavalier d’Euler — code dans *euler.js*

# Algorithme récursif

Un algorithme récursif est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème1. L'approche récursive est un des concepts de base en informatique.

Les premiers langages de programmation qui ont autorisé l'emploi de la récursivité sont LISP et Algol 60. Depuis, tous les langages de programmation généraux réalisent une implémentation de la récursivité.

# Factorielle

$$n! = \begin{cases} n * (n-1)! & \text{si } n > 1 \\ 1 & \text{sinon} \end{cases}$$

- sous-problème
- cas d'arrêt

```
/*
 * @param n entier
 * @return factorielle de n  (version itérative)
 */
var fact = function(n) {
    var  r = 1;
    for (var i= 2; i<n; i++)
      r *= i;
    return r;
};
```

Très bien, mais ne colle pas à la définition

# Factorielle

$$n! = \begin{cases} n * (n-1)! & \text{si } n > 1 \\ 1 & \text{sinon} \end{cases}$$

- sous-problème
- cas d'arrêt

```
/*
 * @param n entier
 * @return factorielle de n  (version recursive)
 */
var fact = function(n) {
    if (n === 0) return 1;
    return n * fact(n-1);
};
```

test d'arrêt

récursion

**Une fonction récursive (qui s'appelle elle même) doit contenir un test d'arrêt et un appel à un sous-problème**

# Pile de récursivité

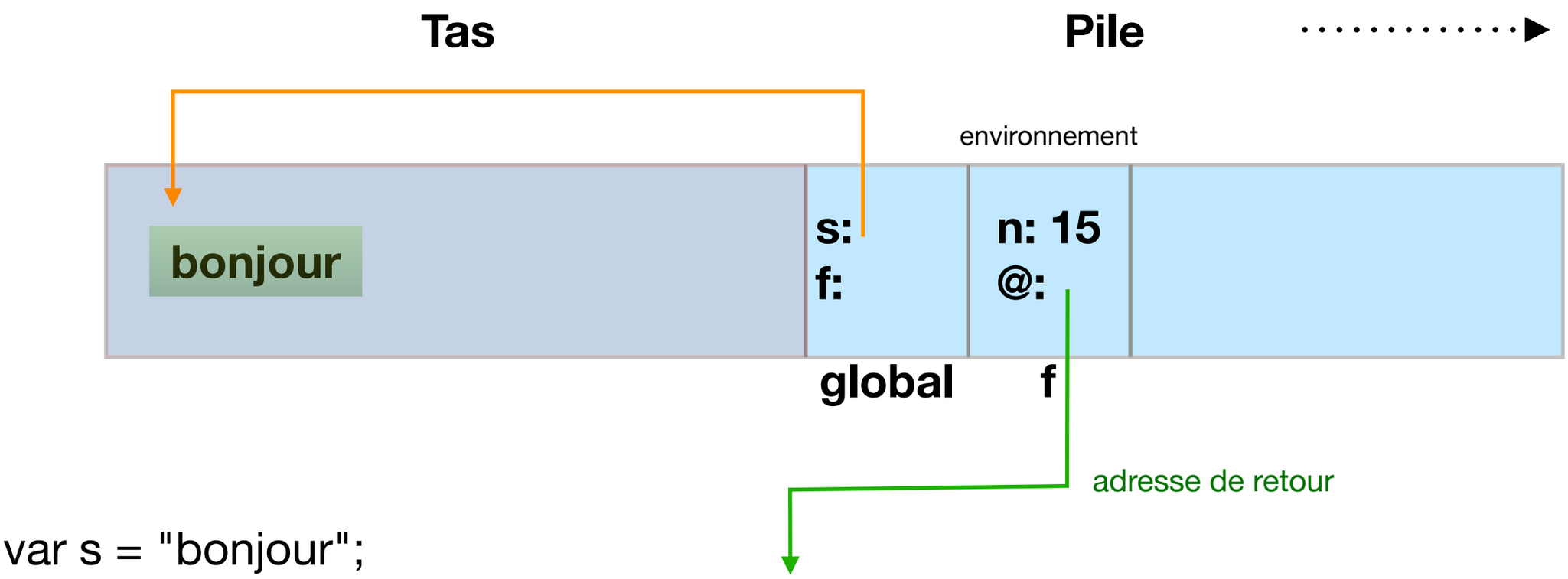
- La mémoire est partagée en deux zones:
  - le tas
  - la **pile**

- À chaque appel de fonction, l'interpréteur crée dans la pile un environnement qui contient:
  - les valeurs des arguments formels,
  - l'adresse de retour à l'appelant
  - les variables locales s'il y en a

- Le code de la fonction est alors déroulé « dans cet environnement » (les variables y sont recherchées, et lorsqu'elles ne sont pas trouvées, l'environnement global est consulté).

- Une fois la fonction terminée, l'adresse de retour est consultée, l'environnement est éliminé de la pile (dépilé) et le code continue à l'adresse de retour, dans l'environnement se trouvant au sommet de pile.

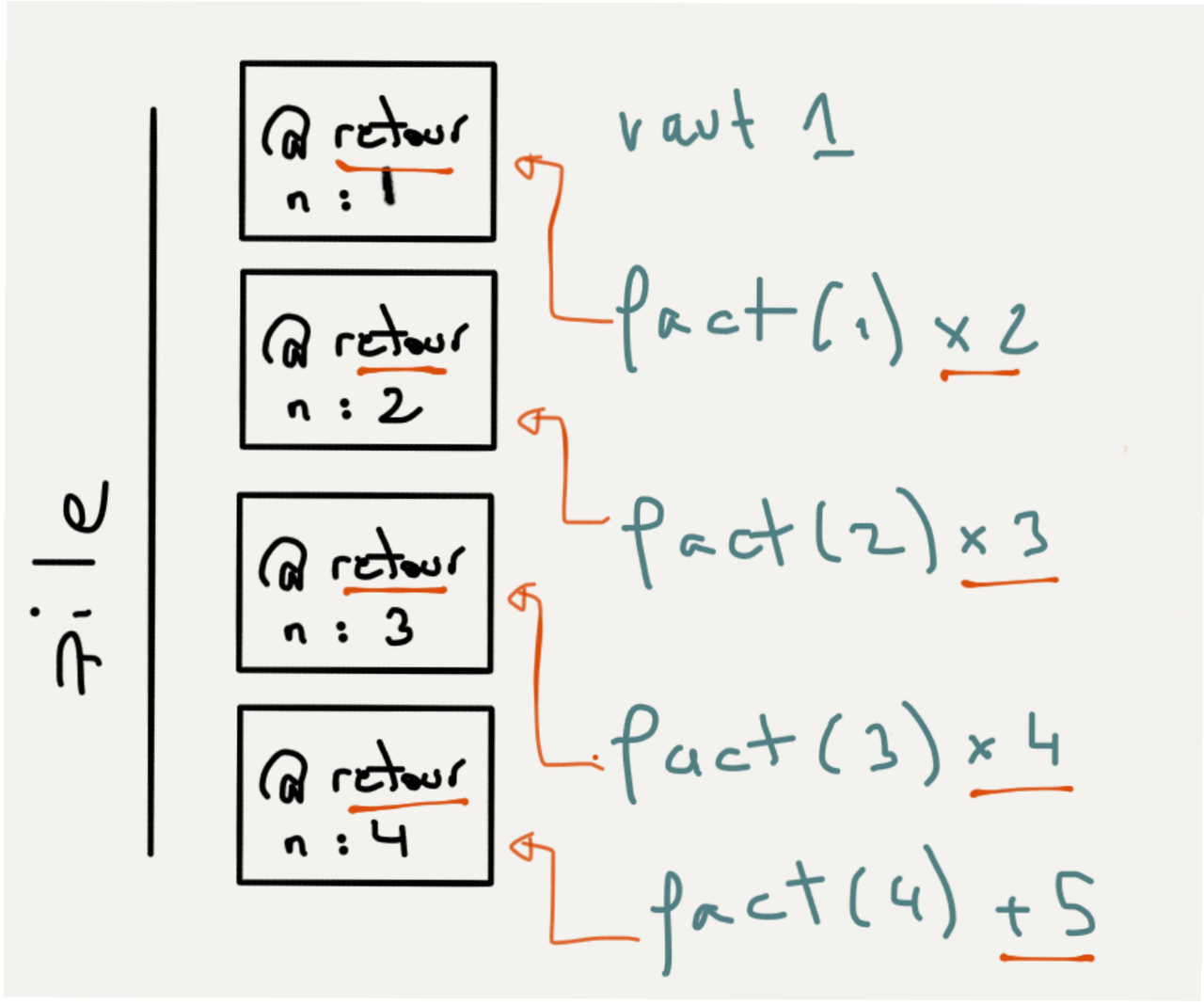
# Modèle mémoire: Tas et Pile




```
var s = "bonjour";
print( (function f(n) { return n+1; })(15) + 18 );
```

- Le tas contient les objets
- La pile contient les variables locales (et arguments formels)
  - on peut concevoir les variables globales comme des variables stockées dans un environnement « global » dans la pile.

# Factoriel




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

fact(4) + 5;

# Débordement de pile (stack overflow)

**Si une fonction récursive n'amène pas à un test d'arrêt, il y alors saturation de la pile par des environnements d'appel. En codeboot (via Safari), au bout de quelques dizaines de secondes, le navigateur relance codeboot en éliminant les codes ouverts de la cache ...**

# Traverser une structure sans boucle

- Nous nous concentrons ici sur la parcours d'une chaine (string) dans le but de l'afficher. (C'est bien sûr inutile !).

- Il faut faire intervenir un sous-problème:

- Parcourir une chaine peut se concevoir comme l'affichage de son premier caractère puis le parcours de la chaine privée de son premier caractère.

- Il s'agit bien d'un énoncé récursif, en voici quelques implémentations.

# Take 1

```
/*
 * @param s string
 * affiche s caractere par caractere sans boucle
 */
var affiche1 = function(s) {
    if (s.length > 0) {
        print(s.charAt(0));
        affiche1(s.substr(1))
    }
};
```

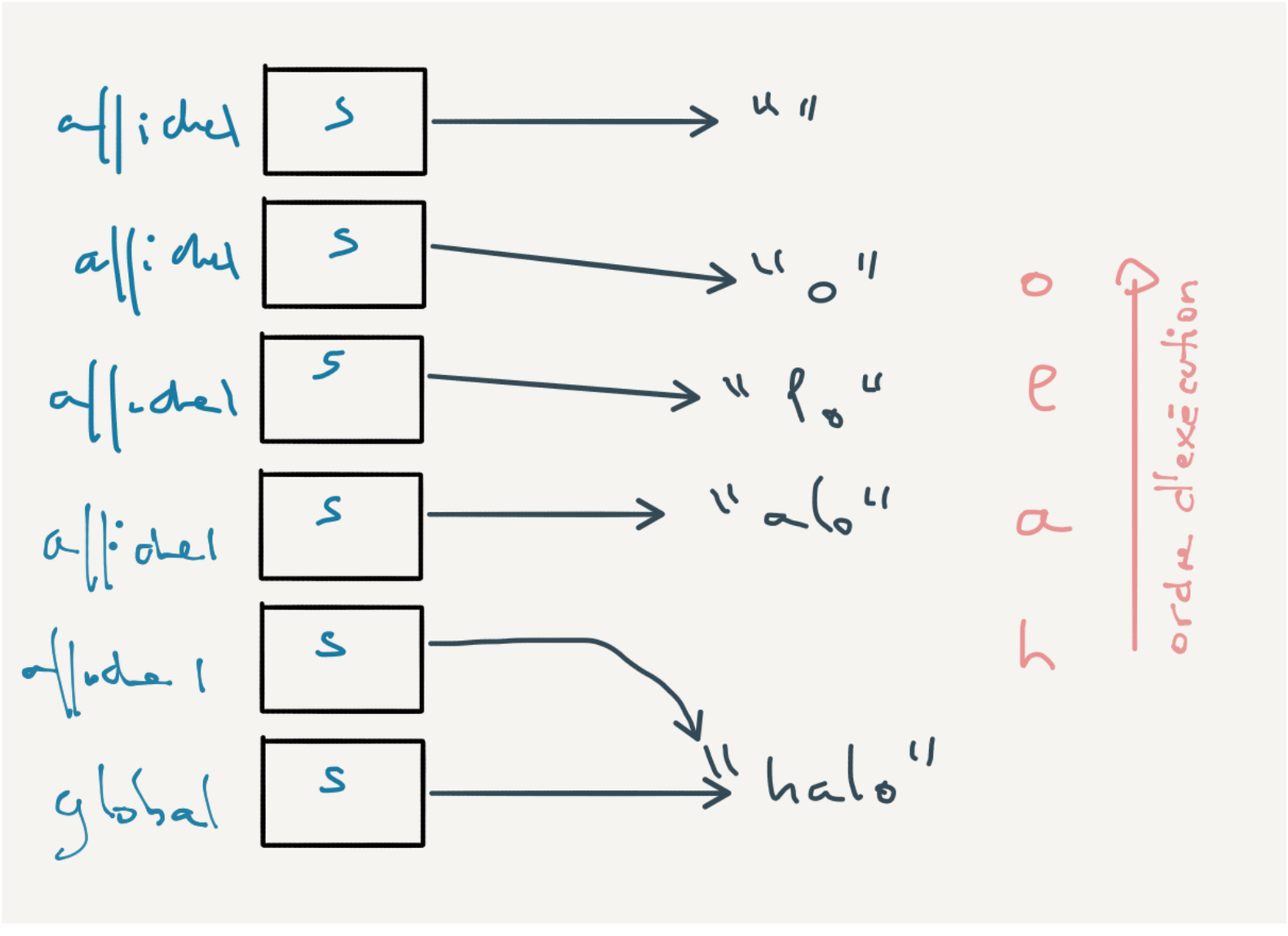
forme de test d'arrêt

substr crée une nouvelle chaine

Pour une chaîne de *n* symbole, il y a *n* sous-chaines crées !

# Take 1

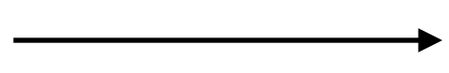
## Exécution




var s = ‘halo’ ;
**affiche1**(s);

# Take 2

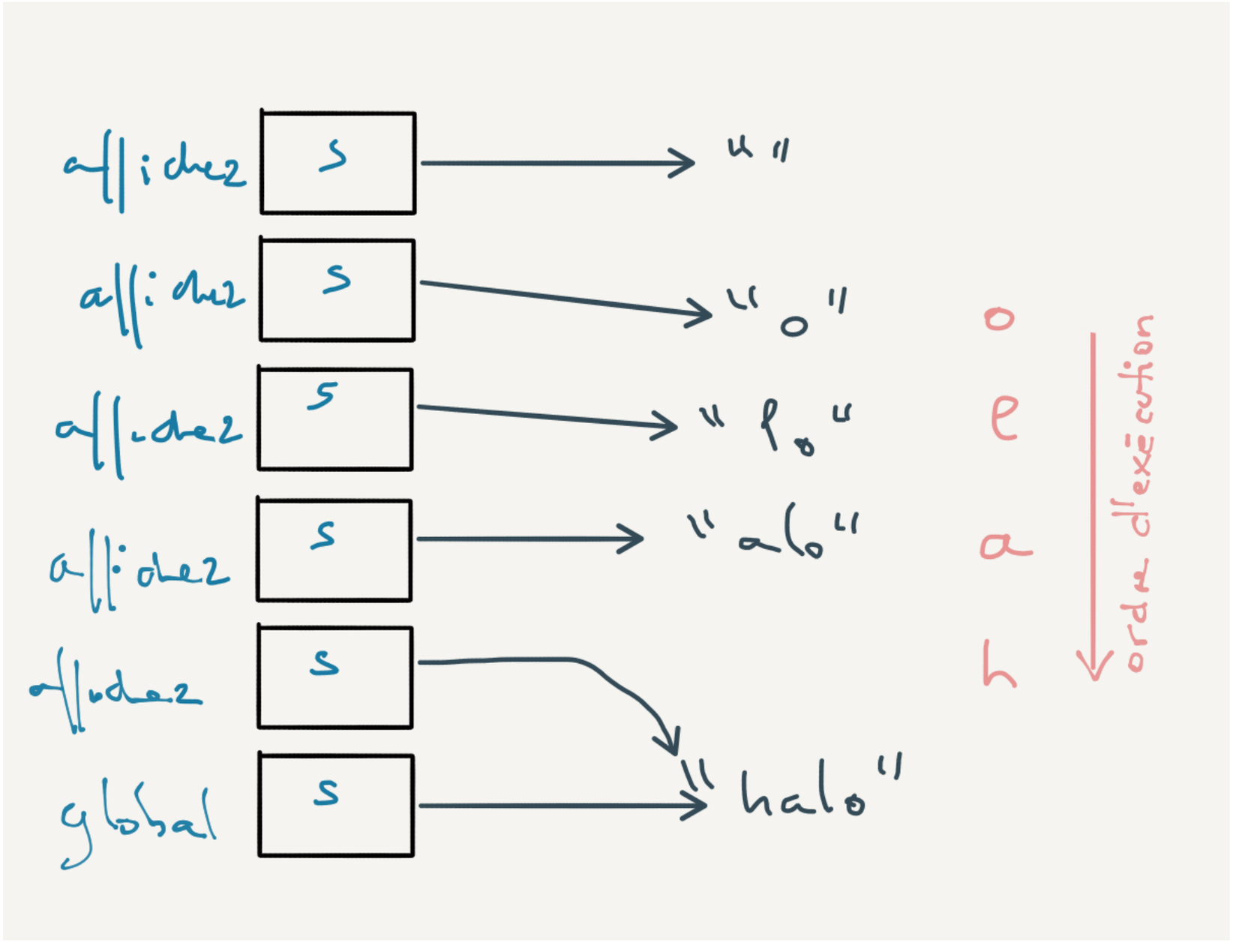
```
/*
 * @param s string
 * quiz: que fait cette fonction ?
*/
var affiche2 = function(s) {
    if (s.length > 0) {
       affiche2(s.substr(1))
       print(s.charAt(0));
   }
};
```



forme de test d’arrêt

# Take 2

## Exécution



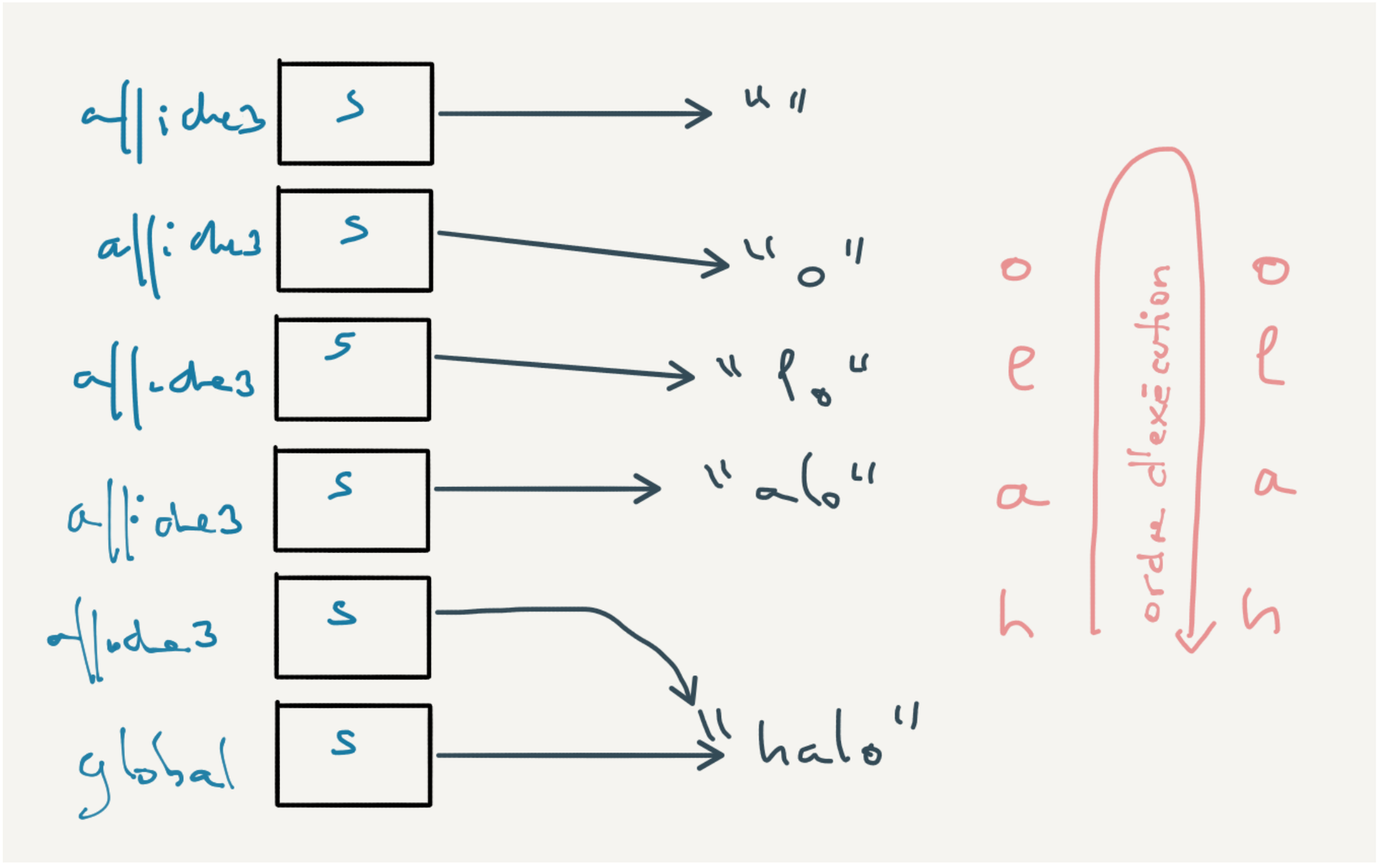

var s = ‘halo’ ;
**affiche2**(s);

# Take 3

```
/*
 * @param s string
 * quiz: que fait cette fonction ?
 */
var affiche3 = function(s) {
    if (s.length > 0) {
        print(s.charAt(0));
        affiche3(s.substr(1))
        print(s.charAt(0));
    }
};
```

# Take 3

## Exécution




```
var s = 'halo' ;
affiche3(s);
```

# Take 4

```
/*
 * @param s string
 * quiz: que fait cette fonction ?
 */
var affiche4 = function(s) {
    if (s.length > 0) {
        affiche4(s.substr(1));
        print(s.charAt(0));
        affiche4(s.substr(1));
    }
};
```

# Take 5

```
var affiche5 = function(s) {
    var helper = function(s,i) {
      if (i < s.length) {
          print (s.charAt(i));
          helper(s,i+1);
      }
    };

    helper(s,0);
};
```

On peut déclarer une fonction à l'intérieur d'une autre

On aurait tout aussi bien pu ajouter un argument à affiche5, mais quel intérêt d'imposer à un utilisateur d'en comprendre la sémantique ?

Cette variante est beaucoup plus efficace que *affiche1*

# Quiz

```
var affiche6 = function(s,ii) {

    var i = (undefined === ii)? 0 : ii; // i vaut ii si specifie, 0 sinon

    if (i<s.length) {
      switch (s.charAt(i)) {
      case 'a':  case 'e':
      case ‘i':   case 'o':
      case ‘u':  case 'y':
          print(s.charAt(i));
          affiche6(s,i+1);
          break;
        default:
          affiche6(s,i+1);
          print(s.charAt(i));
          break;
        } // switch
     } // i<s.length
} // affiche6

affiche6("bonjour");
```

# Fibonacci

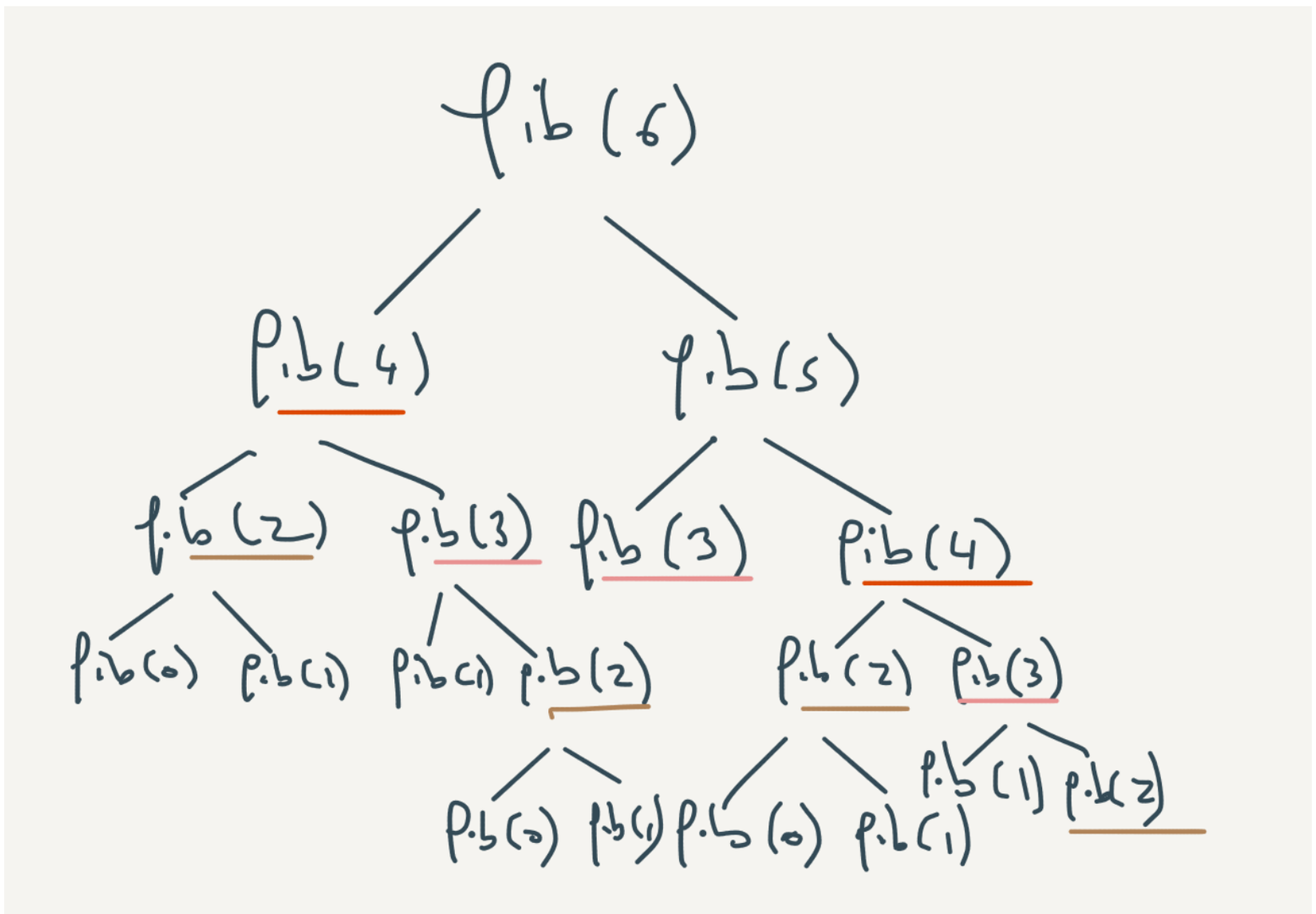
$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{sinon} \end{cases}$$

```
var fibonacci = function (i) {
    switch (i) {
    case 0:
    case 1:
     return i;
    default:
     return fibonacci(i-2) + fibonacci(i-1);
    }
}
```

Fidèle à la définition mais très lent !

# Fibonacci

Cette solution récursive est inefficace: $o(2^n)$




**Recalcul des sous-problèmes**

# Fibonnaci

## Complexité exponentielle

$T(n) = T(n-1) + T(n-2) + c$

or $T(n-2) \leq T(n-1)$

(paire de fn avec la borne inf.)

donc $T(n) \leq 2^1 T(n-1) + c$

de même, $T(n-1) \leq 2^1 T(n-2) + c$

donc $T(n) \leq 2^2 T(n-2) + 2c + c$

or $T(n-2) \leq 2\,T(n-3) + c$

$\dots$

donc $T(n) \leq 2^{n-1} \underbrace{T(n-(n-1))}_{T(1) \equiv c} + 2^{n-2}c + 2^{n-3}c + \dots + 2^0 c$

$\leq c\left[2^{n-1} + 2^{n-2} + \dots + 2^1 + 2^0\right]$

$a^0 + a^1 + \dots + a^n = \frac{a^{n+1} - 1}{a - 1}$

$\Rightarrow 2^0 + 2^1 + \dots + 2^{n-1} = 2^n - 1$

donc $T(n) \leq c\left[2^n - 1\right]$

Soit *T(n)* le temps pour calculer Fibo(n)

et *c* le temps d'une comparaison (test d'arrêt)

# Fibonacci

## Plus rapide (au détriment de la mémoire)

```
var fibonacciMemo = function (i) {

    var memo = new Array(i+1);

    var helper = function (i) {
      if (undefined === memo[i]) {
          switch (i) {
          case 0:
          case 1:
            return memo[i] = i;
          default:
            return memo[i] = helper(i-2) + helper(i-1);
          }
      }
      else
          return memo[i];
  }; // helper

  return helper(i);
};
```

**Cette variante mémorise tous les sous problèmes (possible car leur nombre est *i*).**

# Fibonacci

## Variante itérative !

```
var fibonaccilter = function (n) {
    if (n < 2) return n;    // hyp: n positif
    var f2=0, f1=1, somme = 0;
    for (var i=2; i<= n; ++i) {
       somme = f1 + f2;
       f2 = f1;
       f1 = somme;
    }
    return somme;
}
```

**Pas forcément très lisible, mais beaucoup plus rapide !**

***f2*** **est à comprendre comme fib(n-2),** ***f1*** **comme fib(n-1)**

# Récursion terminale

## Notion

Une récursion est dite terminale si le résultat de l'appel est récursif est retourné sans nécessiter d'autre traitement.

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

Cette récursion n'est pas terminale car la valeur de retour de fact doit être multipliée par n

```
var add = function(n,res) {
    if (n === 0) return res;
    return add(n-1,n+res);
};
```

Cette récursion est terminale car la valeur de retour de add est retournée directement

Les bons compilateurs/interpréteurs transforment les récursions terminales en boucle.

# Fibonacci

## Version récursive et aussi rapide qu'une boucle

```
var fibonacciLast = function (i) {

    var helper = function(i,f2,f1) { // f1=fib(i-1) f2=fib(i-2)

        switch (i) {
          case 0: return f2;
          case 1: return f1;
          default:
              return helper(i-1,f1, f2+f1);
          } // switch
      };

      return helper(i,0,1);
};
```

**f(6)**

helper(6,0,1)
helper(5,1,1)
helper(4,1,2)
helper(3,2,3)
helper(2,3,5)
helper(1,5,8) **vaut 8**

Le résultat d'un appel récursif n'a pas à être traité (**récursion terminale**)

# Palindrome

**Une chaîne est un palindrome si sa lecture de la gauche vers la droite et dans l'autre sens donne la même chaîne.**

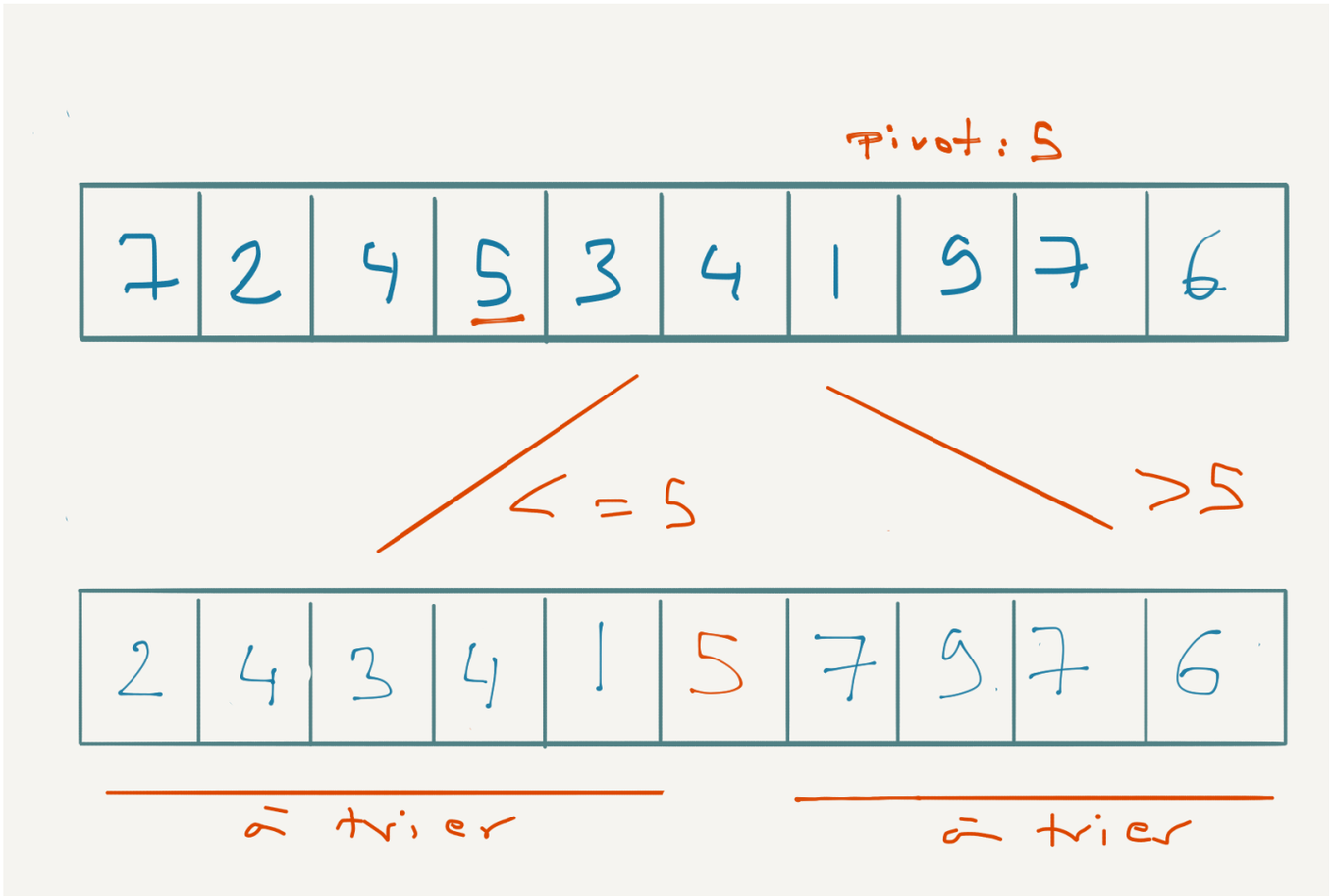
ex: **laval, abcddbca,** et **a** sont des palindromes

```
var isPalindrome = function (s) {
    if (s.length <= 1) return true;
    return (s.charAt(0) === s.charAt(s.length-1)) &&   isPalindrome(s.slice(1,s.length-1));
}

var isPalindrome2 = function (s) { // plus efficace

    var helper = function(s, deb, fin) {
      if (deb >= fin) return true;
      return (s.charAt(deb) === s.charAt(fin)) && helper(s,deb+1,fin-1);
    };

    return helper(s,0,s.length-1);
}
```

# Tri Rapide

*C.A.R. Hoare, 1961*

- L'un des tris les plus rapides (complexité moyenne: *o(n x log(n))*)
- Si le tableau est déjà trié, il est préférable d'utiliser par exemple le tri par insertion.




- **Trier:**
  - Prendre une valeur **pivot**
  - Mettre les valeurs inférieure ou égale à gauche du pivot
  - Mettre les valeurs supérieures à droite
  - **Trier** les deux zones ainsi délimitées

Si le pivot est bien choisi, on divise le problème en deux

# Tri rapide

## squelette

```
/*
 * @param t tableau à trier
 * @param si deb n'est pas specifie, 0 est utilise
 * @param si fin n'est pas specifie, t.length-1 est utilise
 * @return l'adresse du tableau trié sur l'intervalle [deb,fin]
 */
var quick = function(t, deb, fin) {
      var ipivot;

   if (deb < fin) {
      ipivot = partition(t,deb,fin,select(t,deb,fin));
      quick(t,deb,ipivot-1);
      quick(t,ipivot+1,fin);
   }

   return t;
};
```

dans *quicksort.js*

- **select** (à écrire) retourne l'indice du pivot
- **partition** (à écrire) répartit les valeurs <= au pivot d'un côté, et les valeurs > de l'autre et retourne l'indice du pivot une fois la répartition faite

Reste bien sûr à écrire la fonction de partition (**partition**) des valeurs par rapport au pivot, et la fonction de sélection (**select**) de la valeur pivot.

# Tri rapide

## Choix (triviaux) d'un pivot

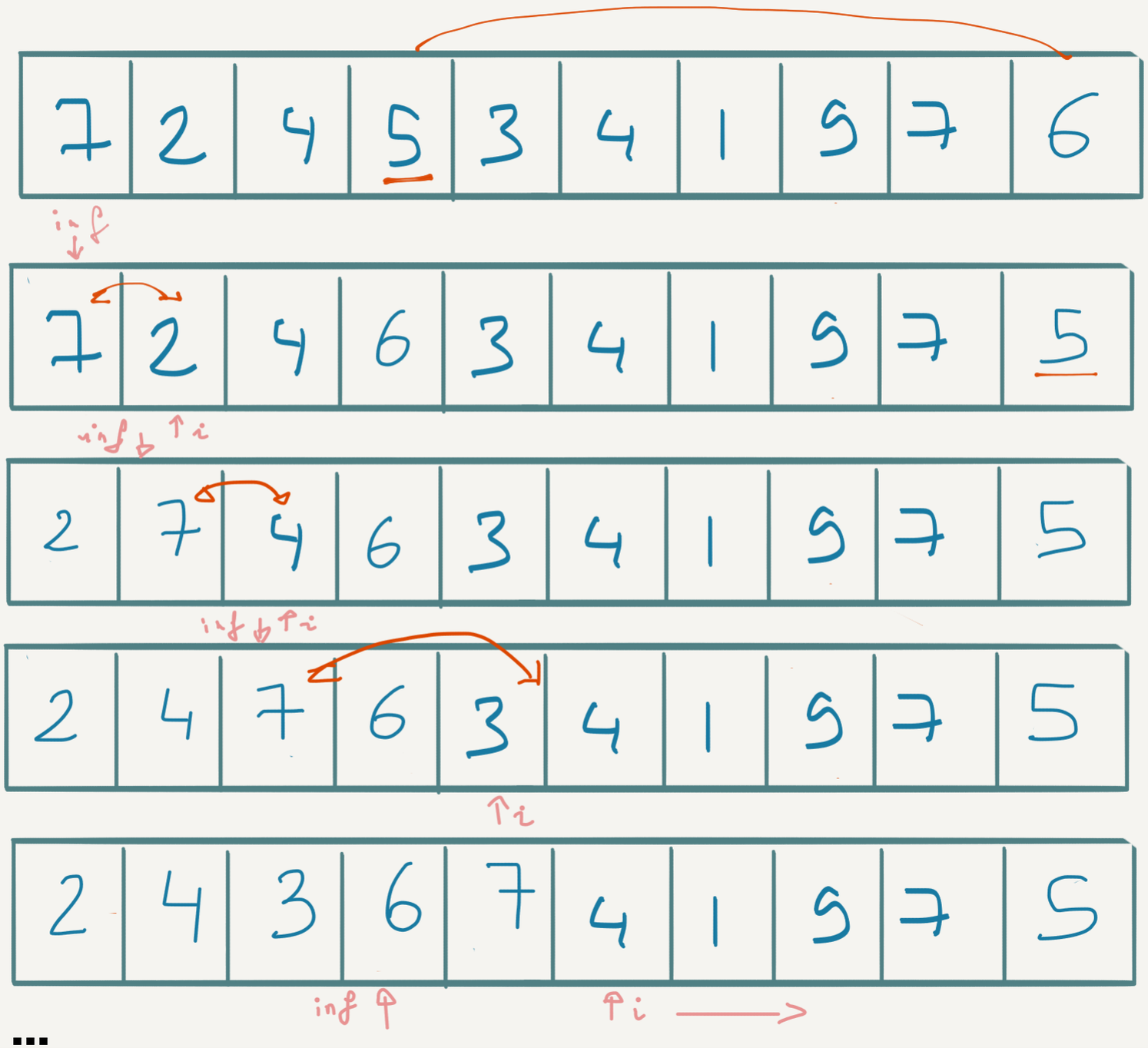
```
/*
 * selection d'un pivot dans t dans l'intervalle [deb,fin]
 * les fonctions retournent l'indice de la valeur pivot
 */
var selectPivotFirst = function(t,deb,fin) {
    return deb;                // life is easy, eh ?
};

var selectPivotAlea = function(t,deb,fin) {
    return alea(deb,fin+1);   //un entier dans [deb,fin]
};
```

catastrophique en pratique

alea dans *tools.js*

# Tri rapide

## Partition / take1




- *t[deb,inf[* contient des valeurs inférieures ou égales au pivot (init: *inf = 0*)
- Traverser *t* et mettre en *inf* les valeurs <= *pivot*.

# Tri rapide

## Partition / take1

```
/*
 * @param t tableau
 * @param deb de l'intervalle
 * @param fin de l'intervalle
 * @param ipivot indice du pivot
 */
var partitionne1 = function(t, deb,fin, ipivot) {
```

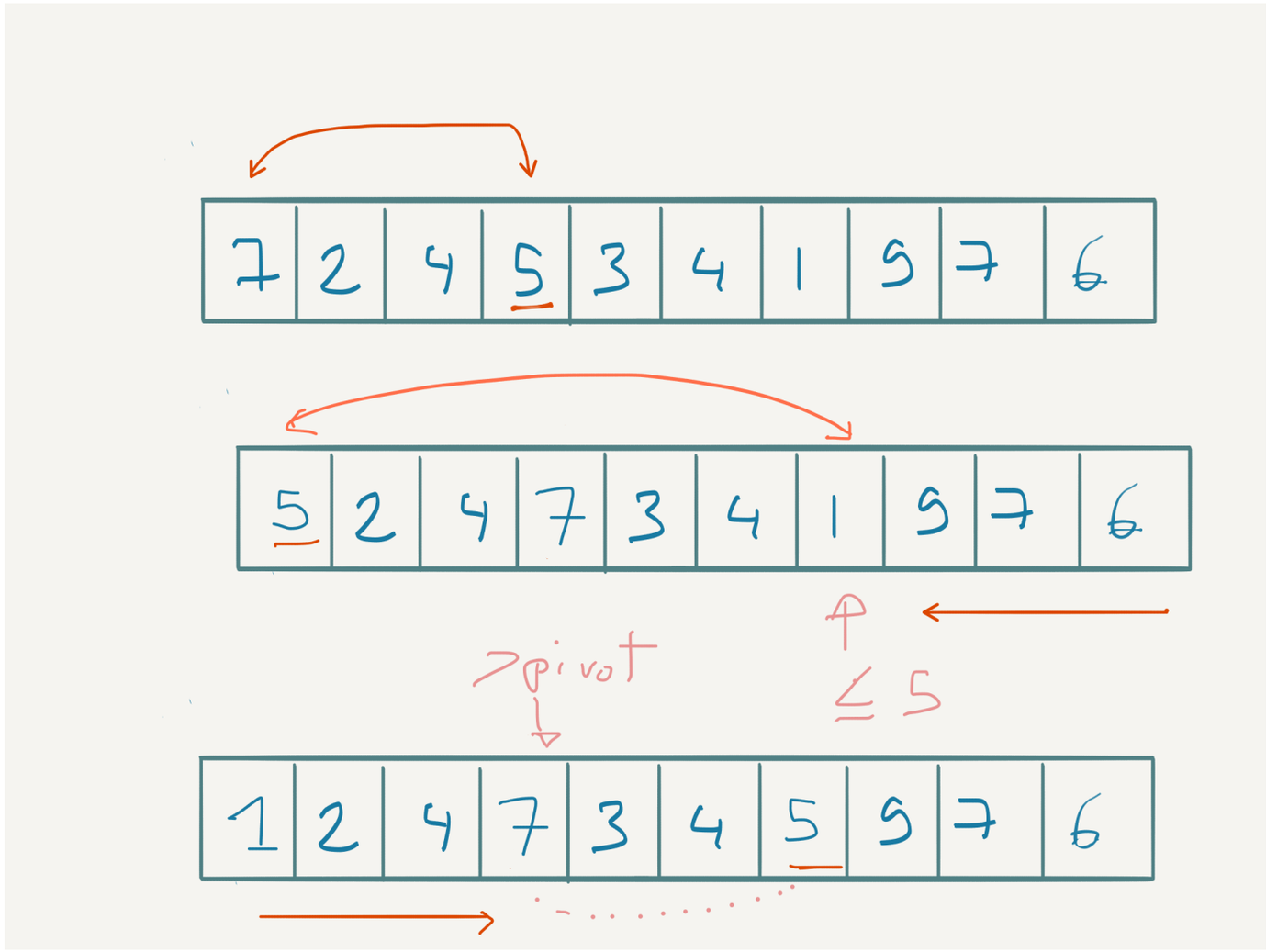
- *t[deb,inf[* contient des valeurs inférieures ou égales au pivot (init: *inf = 0*)
- Traverser *t* et mettre en *inf* les valeurs <= *pivot*.

**swap** est dans *algo-tab.js*

```
   swap(t,ipivot,fin); // mettre le pivot à la fin
   var inf = deb; // aucune valeur inferieure au pivot avant
   var pivot = t[fin];

   for (var i=deb; i<=fin; ++i)
      if (t[i] <= pivot) {
         swap(t,inf,i);
         inf++; // on vient de mettre en inf, une valeur inferieure (ou egale) au pivot
      }

   return inf-1;      // la place du pivot (-1 car le dernier elt est le pivot et inf est incremente)
};
```

# Tri rapide

## Partition / take2




- swapper le pivot avec la première valeur
- tant que pas tout visité
  - avancer (G->D) tant qu'on a une valeur <= pivot
  - swapper la valeur > pivot avec le pivot
  - avancer D->G tant que valeur > pivot

# Tests unitaires

- ***test_partition*** Comparaison des fonctions de partition
- ***test_quick*** Test du quickSort
- ***test_worst_case*** Tri d’un tableau trié

dans *quicksort.js*

# Comparaison des fonctions de partition

```
var test_partition = function(nbTest) {

   var t, c, nb, nb2;
   var winner1 = 0;            // #fois ou partitionne1 a un nombre inférieur de swaps
   var total1=0, total2=0;  // # de swap de chaque méthode de partition

   var mycall = function(t,func) { swap.calls = 0; func(t,0,t.length-1,0); return swap.calls; };

   for (var i=0; i<nbTest; ++i) {

      t = aleaTab(alea(1000,10000),0,100); // tableau de taille aleatoire (entre 1k et 10k)
      c = copie(t);                                  // de valeurs alatoires entre 0 et 100

      nb1 = mycall(t,partitionne1);       // n1 est le nb de swaps realises par partionne1
      nb2 = mycall(c,partitionne2);       // n2 est le nb de swaps realises par partionne2

      total1 += nb1;
      total2 += nb2;
      if (nb1 < nb2) ++winner1;
     }

   print(nbTest, "partition1 winner pour"  ,winner1, "tests, soit: ", percent(winner1,nbTest));
};
```

**aleaTab, copie** dans *algo-tab.js*

**alea** et **swap** dans *tools.js*

# Test quick-sort

```
var test_quick = function(partitionne, p) {

    var t = aleaTab(50000,10,3000);          // 50k elt entre 10 et 30000
    var c1 = copie(t);
    var c2 = copie(t);

    timing( function() { return triInsertion(t); }, "tri (ms) par insertion -- random"  );

    timing( function() { return quickSort(c1,0,c1.length-1, partitionne, selectPivotFirst); },
            "tri (ms) par quick("+p+",1st) -- random"  );

   timing( function() { return quickSort(c2,0,c2.length-1, partitionne, selectPivotAlea);  },
            "tri (ms) par quick("+p+",alea) -- random"  );

   assert(isSorted(t),"echec tri quick (1)");      // les tableaux doivent etre tries:
   assert(isSorted(c1),"echec tri quick (2)");
   assert(isSorted(c2),"echec tri quick (3)");

   timing( function() {  return quickSort(t,0,t.length-1,partitionne,selectPivotFirst); },
            "tri (ms) par quick("+p+",1st) -- trie");
   timing(function() { quickSort(t); }, "tri (ms) par quick("+p+",alea) -- trie");
};
```

**triInsertion** dans *sort.js*

**timing** dans *tools.js*

**isSorted** dans *algo-tab.js*

# (détails)

## timing(f,m)

dans *tools.js*

```
/*
 * @param f fonction a executer
 * @param message a afficher le cas échéant
 * @return le temps en ms mis pour executer f
*/
var timing = function(f, message) {
    var t_before = new Date().getTime();                  // temps
    f();                                                  // lancement de f
    var temps = new Date().getTime() - t_before; // temps

    if (undefined !== message)
      print(message,temps,"ms");
    return temps;
};
```

# Le pire cas

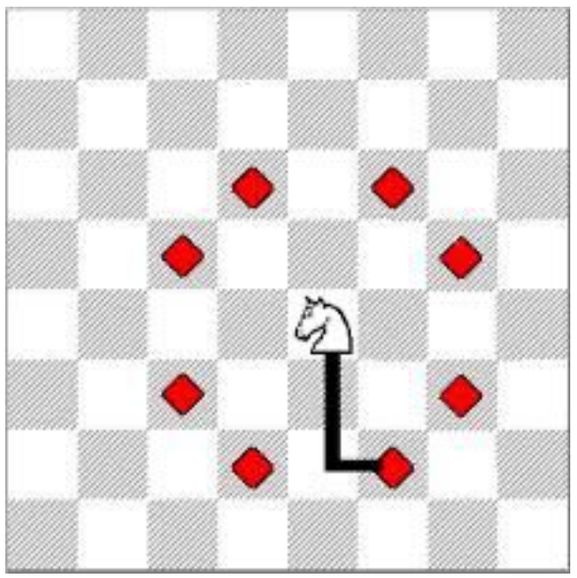
**Tableau déjà trié**

```
var test_worst_case = function(pivot,verbose) {

    var t = new Array(1000000);
    for (var i=0; i<t.length; ++i)
        t[i] = i;
    assert(isSorted(quickSort(t,0,t.length-1,partitionne2,pivot)));
    if (verbose)
        print(t.slice(0,10));
};
```

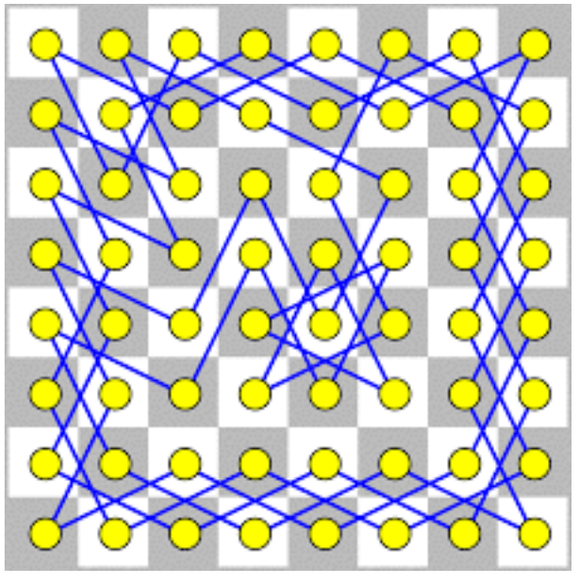
```
test_worst_case(selectPivotAlea,true);
test_worst_case(selectPivotFirst,true);
```

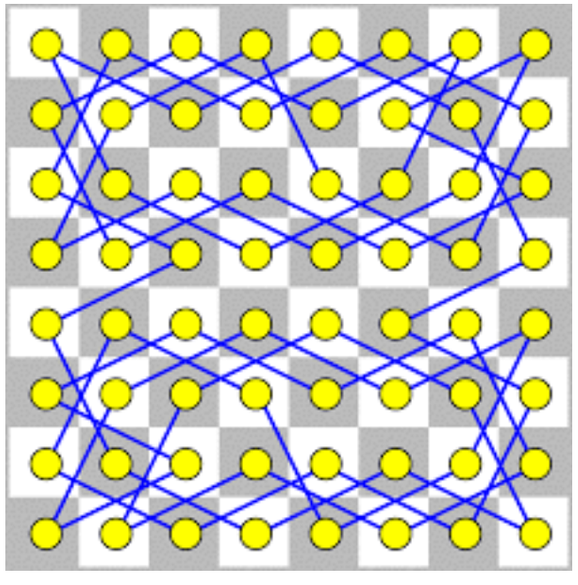
→ **Outch !**

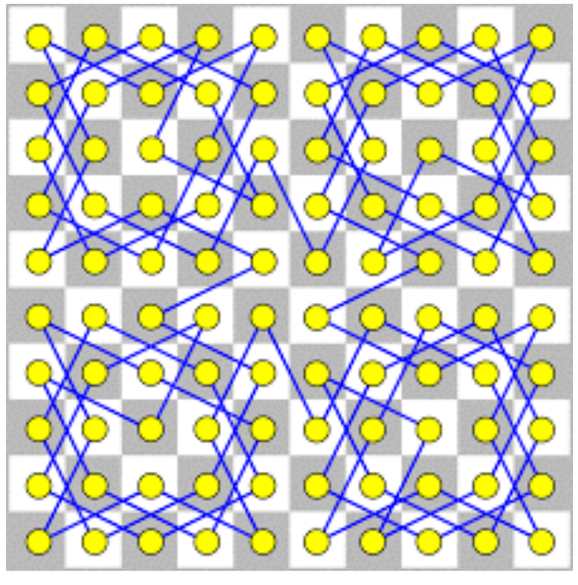
# Le cavalier d’Euler



Passer par toutes les cases  d’un échiquier sans jamais passer deux fois par la même case.







Images prises de Wikipedia

# Le cavalier d’Euler

Que l’on représentera comme ceci:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 54 | 49 | 40 | 35 | 56 | 47 | 42 | 33 |
| 39 | 36 | 55 | 48 | 41 | 34 | 59 | 46 |
| 50 | 53 | 38 | 57 | 62 | 45 | 32 | 43 |
| 37 | 12 | 29 | 52 | 31 | 58 | 19 | 60 |
| 28 | 51 | 26 | 63 | 20 | 61 | 44 | 5 |
| 11 | 64 | 13 | 30 | 25 | 6 | 21 | 18 |
| 14 | 27 | 2 | 9 | 16 | 23 | 4 | 7 |
| 1 | 10 | 15 | 24 | 3 | 8 | 17 | 22 |

où on numérote l’ordre dans lequel les cases sont visitées

# Cavalier d'Euler

## Un peu de codage

```
var nbMax = taille * taille;

var grille = new Array(taille);
for (i=0; i<grille.length; ++i)
      grille[i] = new Array(taille);
```

une grille *taille x taille* où une case est libre si elle vaut **undefined**

```
var afficheSolution = function () {
    for (var i=0; i<grille.length; ++i)
          print (grille[i].join("\t"));
};
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 12 | 9 | 6 | 3 | 14 | 17 | 20 |
| 10 | 7 | 2 | 13 | 18 | 21 | 4 | 15 |
| 31 | 28 | 11 | 8 | 5 | 16 | 19 | 22 |
| 64 | 25 | 32 | 29 | 36 | 23 | 48 | 45 |
| 33 | 30 | 27 | 24 | 49 | 46 | 37 | 58 |
| 26 | 63 | 52 | 35 | 40 | 57 | 44 | 47 |
| 53 | 34 | 61 | 50 | 55 | 42 | 59 | 38 |
| 62 | 51 | 54 | 41 | 60 | 39 | 56 | 43 |

# Cavalier d'Euler

```
var jouer = function (a,b,cavalier) {

    if ( entre(a,0,grille.length-1) &&
         entre(b,0,grille.length-1) &&
         (grille[a][b] === undefined)  ) {

        grille[a][b] = cavalier;
        continuer(a,b,cavalier+1);
        grille[a][b] = undefined; // libère
    }
};
```

jouer(0,0,1);

entre dans *tools.js*

```
var continuer = function(a,b,cavalier) {

    if (cavalier > nbMax)
        afficheSolution();

    else {
        max = Math.max(max,cavalier);
        jouer(a-2,b-1,cavalier);
        jouer(a-2,b+1,cavalier);
        jouer(a-1,b-2,cavalier);
        jouer(a-1,b+2,cavalier);
        jouer(a+1,b-2,cavalier);
        jouer(a+1,b+2,cavalier);
        jouer(a+2,b-1,cavalier);
        jouer(a+2,b+1,cavalier);
    }
};
```

8 déplacements possibles