Récursivité

IFT1015: Programmation1

Pascal Vincent et Aaron Courville

Définition de récursif

- Le fait pour un programme ou une méthode de s'appeler lui-même.
- Par exemple
  - Définir la notion de « ancêtre »
    - Les parents sont des ancêtres
    - Les ancêtres des parents sont des ancêtres
  - Définition de « ami »
    - Les gens directement autour de moi sont des amis
    - Les amis de mes amis sont mes amis

Pourquoi la récursivité?

- Certains problèmes en informatiques sont de nature récursive
  - Ne peuvent pas être résolus sans récursion
  - E.g. ancêtre et ami
- Certains problèmes peuvent être formulés de façon plus concise et claire avec récursivité

Un exemple en JavaScript

```
function calcul(val)
    {
        if (val<=0) return 0;
        else return val + calcul(val-1);
    }
```

$$calcul(val) = \begin{cases} 0, & \text{si } val \leq 0 \\ val + calcul(val - 1), & \text{sinon} \end{cases}$$

Quelques fonctions mathématiques

- Fibonacci

$$fibo(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ fibo(n-1) + fibo(n-2), & n > 1 \end{cases}$$

n:	0	1	2	3	4	5	6	7
Fibo(n):	1	1	2	3	5	8	13	21

- Une fonction de nature récursive
  - La valeur de fibo(n) dépend de fibo(n-1) et fibo(n-2)

Définition d'une fonction récursive

$$fibo(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ fibo(n-1) + fibo(n-2), & n > 1 \end{cases}$$

Cas simples

Cas récursif

- Cas simple
  - Problème directement résoluble
  - Pas de récursion
- Cas récursif (cas complexe)
  - Appel à la même fonction
  - Pour des cas plus simples
    - n-1, n-2 sont plus simples que n

Forme d'une méthode récursive

- *Méthode* pour un problème *n*
  - Si *n* est directement résoluble,
    - retourner une réponse ou faire un traitement direct
  - Sinon (cas récursif)
    1. Décomposer le problème *n* en des sous problèmes moins complexes: *n1, n2, …*
    2. Appel à *Méthode* pour traiter *n1, n2, …*
    3. Utiliser les résultats pour *n1, n2, …* pour composer un résultat pour *n*

Exemple Fibonacci

- Fibo(n)
  - Si n=0 ou n=1: cas simple
    - Retourner le résultat = 1
  - Si non (n>1): cas complexe
    - Décomposer le problème en deux parties
      - n-1 et n-2
      - Appel a la méthode pour les traiter: fibo(n-1), fibo(n-2)
      - Composer le résultat pour n: fibo(n-1)+fibo(n-2)

Importants pour réussir la récursion

- Bien définir le cas simple
  - Pour arrêter la récursion
  - Sinon, récursion à l'infinie
- Bien décomposer le problème
  - En sous-problèmes plus simples
  - Si les sous-problèmes sont aussi complexes: récursion à l'infinie

Mauvais exemples

- Sans une bonne condition d'arrêt:

$$fibo(n) = \begin{cases} \\ fibo(n-1) + fibo(n-2), n > 1 \end{cases} \qquad fibo(n) = \begin{cases} 1, \ n = 1 \\ fibo(n-1) + fibo(n-2), n > 1 \end{cases}$$

- Une mauvaise décomposition

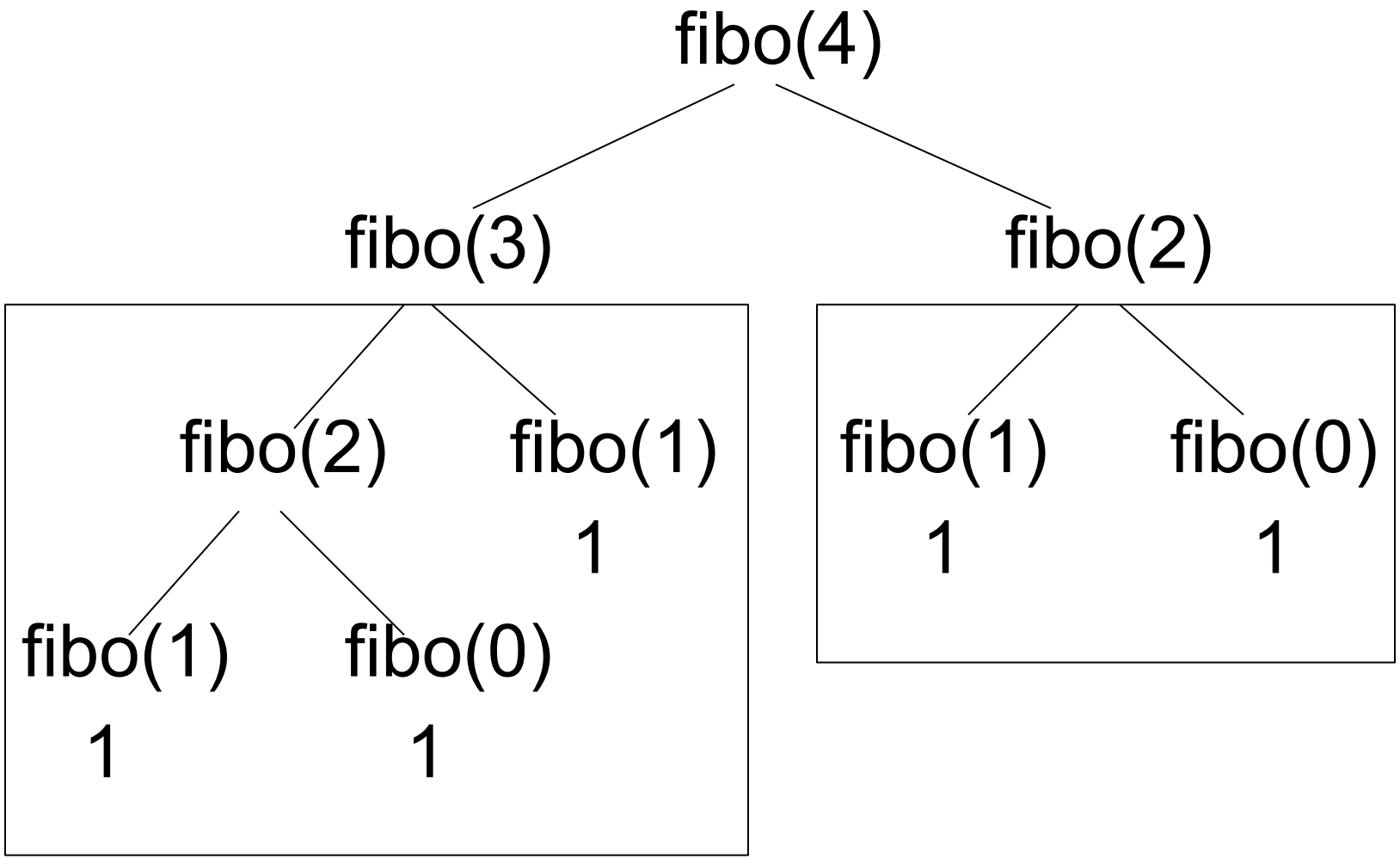
$$fonct(n) = \begin{cases} 1, \ n = 0 \\ 1, \ n = 1 \\ fonct(n) + fibo(n-1), n > 1 \end{cases} \qquad fonct(n) = \begin{cases} 1, \ n = 0 \\ 1, \ n = 1 \\ fonct(n+1) + fibo(n-1), n > 1 \end{cases}$$

En JavaScript

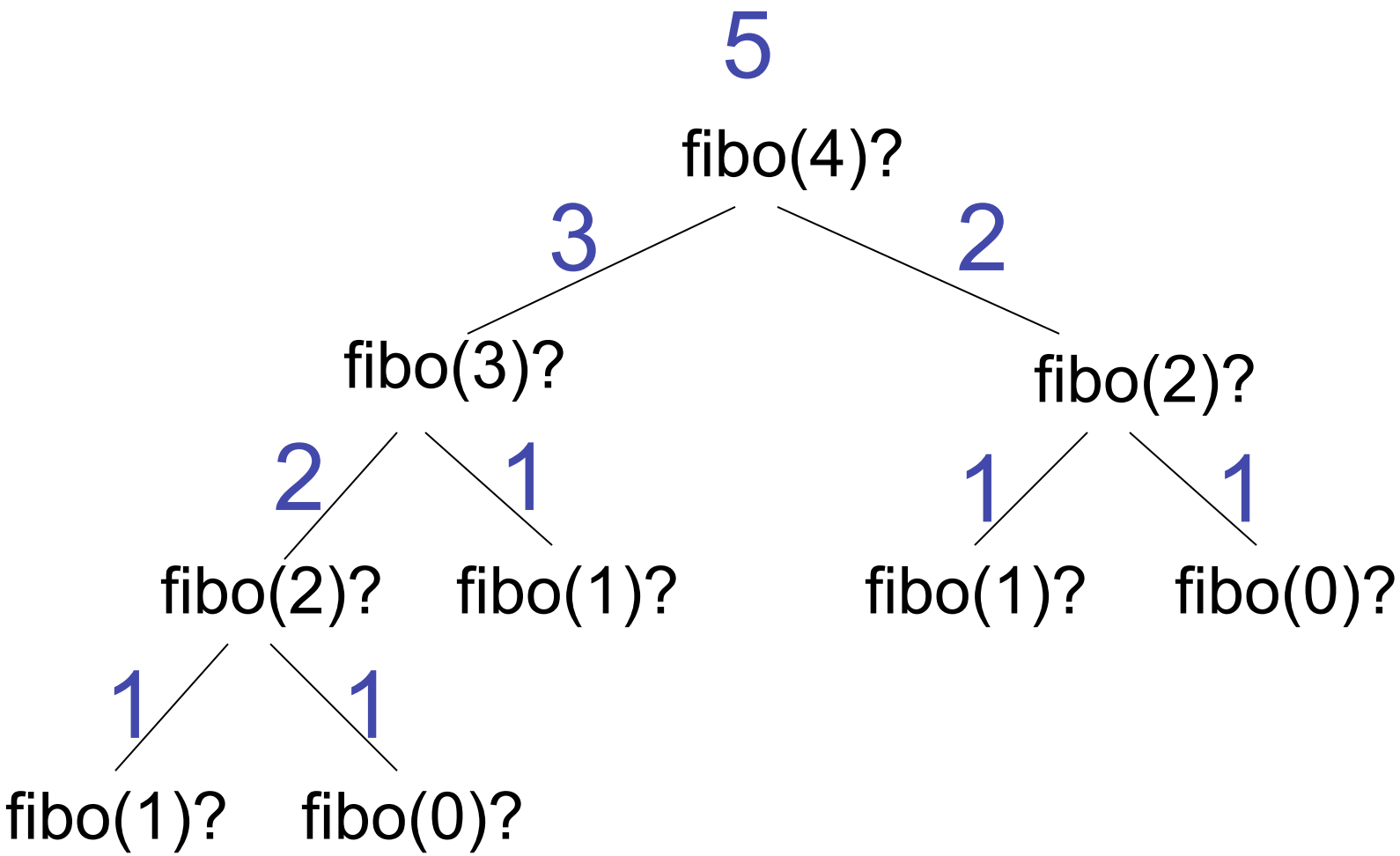
```
function fibo(n)
{
   if (n<=1) return 1;
   else return fibo(n-1)+fibo(n-2);
};
fibo(4)

/* Attention: à chaque récursion, on teste d'abord
   pour traiter le cas simple en premier */
```

Exécution


fibo(4)
fibo(3)
fibo(2)
fibo(2)
fibo(1)
fibo(1)
fibo(0)
1
1
1
fibo(1)
fibo(0)
1
1

Exécution


5
fibo(4)?
3
2
fibo(3)?
fibo(2)?
2
1
1
1
fibo(2)?
fibo(1)?
fibo(1)?
fibo(0)?
1
1
fibo(1)?
fibo(0)?

Fibonacci récursif?

- C’est un exemple pédagogique
- Mais pour calculer un nombre de Fibonacci, cette version récursive naïve est extrèmement inefficace...
- Pourquoi?
- Exercice 1: écrire une version itérative
- Exercice 2: écrire une version récursive plus efficace (inspirée de la version itérative)

14

Fonction transformable en fonction récursive

- Factoriel

$$fac(n) = \prod_{i=1}^{n} i \qquad fac(n) = \begin{cases} 1, & n = 1 \\ n * fac(n-1), & n > 1 \end{cases}$$

n:	1	2	3	4	5	6
Fac(n):	1	2	6	24	120	720

- Somme:

$$somme(n) = \sum_{i=1}^{n} i \qquad somme(n) = \begin{cases} 1, & n = 1 \\ n + somme(n-1), & n > 1 \end{cases}$$

En JavaScript

Versions itératives (avec des boucles)

```
function fact(n)
{
    var i;
    var f=1;
    for (i=1; i<=n; i++) f = f*i;
    return f;
}
function somme(n)
{
    var i;
    var f=0;
    for (i=1; i<=n; i++) f = f+i;
    return f;
}
```

Versions récursives

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


function somme(n)
{
    if(n==1) return 1;
    else return n+somme(n-1);
}
```

Exemple graphique:
procédé artistique: mise en abyme

17

Dessiner une boite de vache qui rit

```
function dessineBoite(g, x, y, r)
{

        // Dessiner le contour de la boîte
        dessineCercle(g,x,y,r);
        // Dessiner la tête
        dessineVache(g,x,y,r*0.90);
        // Boucle d'oreille gauche
        dessineBoite(g, x-r/2, y+r/2, r/10);
        // Boucle d'oreille droite
        dessineBoite(g, x+r/2, y+r/2, r/10);

}
```

g est le contexte graphique

Attention: récursion infinie!

18

Dessiner une boite de vache qui rit

```
function dessineBoite(g, x, y, r)
{
    if (r > 5)
      {  // Dessiner le contour de la boîte
         dessineCercle(g,x,y,r);
         // Dessiner la tête
         dessineVache(g,x,y,r*0.90);
         // Boucle d'oreille gauche
         dessineBoite(g, x-r/2, y+r/2, r/10);
         // Boucle d'oreille droite
         dessineBoite(g, x+r/2, y+r/2, r/10);
      }
}
```

19

Recherche dichotomique *(binary search)*

- C'est un *principe récursif* pour **rechercher** la position d'une certaine valeur dans un **tableau trié**
- On compare la valeur recherchée à celle du milieu du tableau: si c'est la valeur recherchée, on a fini!
- Si la valeur recherchée est plus petite que la valeur milieu, alors **recherchons** la dans la ***première moitié*** du tableau.
- Si elle est plus grande que la valeur milieu, alors **recherchons** la dans la ***deuxième moitié*** du tableau.

20

Recherche dichotomique *version itérative (boucle)*

```
function rechercheDichotomiqueIterative(val, tab, debut, fin) {
    // Valeurs par défaut pour les arguments debut et fin
    if (arguments.length<3) debut = 0;
    if (arguments.length<4) fin = tab.length-1;

    while (debut <= fin) {
        // position du milieu
        var milieu = Math.floor((fin+debut) / 2);
        if (tab[milieu] == val) return milieu;
        if (tab[milieu] > val) fin = milieu-1;
        else                   debut = milieu+1;
    }

    // Valeur jamais trouvée: on retourne une valeur negative
    return -debut-1; // qui indique aussi la place ou l'inserer
}
```

Recherche dichotomique *version récursive*

```
function rechercheDichotomiqueRecursive(val, tab, debut, fin) {
    // Valeurs par défaut pour les arguments debut et fin
    if (arguments.length<3) debut = 0;
    if (arguments.length<4) fin = tab.length-1;

    // On n'a pas trouvé la valeur val dans le tableau:
    if (debut > fin)
        return -debut-1;

    // Comparons a la valeur a la position milieu
    var milieu = Math.floor((debut+fin)/2);
    if (tab[milieu] == val)
        return milieu;
            else if (tab[milieu] > val)
        return rechercheDichotomiqueRecursive(val, tab, debut, milieu-1);
            else // tab[milieu] < val
        return rechercheDichotomiqueRecursive(val, tab, milieu+1, fin);
}
```

Un autre exemple: Traverser un tableau

- Problème initial: traverser le tableau à partir de l'indice 0 (jusqu'à la fin)
- Le problème est-il décomposable ?
  - Pour traverser à partir de l'indice *i*
    - Traiter l'élément courant *i*
    - Traverser le reste du tableau – à partir de *i+1*
  - *Attention:* condition d'arrêt!
    - Traverser le reste – à partir de *i+1* si il en reste

Traverser

- Structure d'une méthode
  - traverser(i, tableau)
    - Traiter i
    - Si pas à la fin du tab, traverser(i+1, tableau)
- Appel pour traverser complètement un tableau

```
tableau = [ ... ];
traverser(0, tableau);
```

Traverser

```
function parcoursCroissant(tableau, index) {
            if( index >= 0 && index < tableau.length ) {
                print( tableau[index]);
                parcoursCroissant(tableau, index + 1);
            }
};
function parcoursDecroissant(tableau, index) {
      if( index >= 0  && index < tableau.length) {
          parcoursDecroissant(tableau, index + 1);
          print( tableau[ index ] );
      }
};
function essaiParcours() {
      tableau = ["a", "b", "c", "d", "e", "f"];
      print("parcoursCroissant:");
      parcoursCroissant( tableau, 0);
      print("parcoursDecroissant:");
      parcoursDecroissant( tableau, 0);
};
essaiParcours();
```

Appeler la fonction de parcours, sans le paramètre supplémentaire

- Quand on appelle une fonction pour parcourir complètement un tableau, on ne devrait pas avoir besoin de spécifier l'index du début (0).
  - Appel normal: parcours(tab)
- Solution possible:
  - Écrire une autre fonction pour l'utilisateur, prenant seulement le paramètre tableau:

```
function parcours(tableau)
  { parcoursCroissant(tableau, 0); }
```

Appeler la fonction de parcours, sans le paramètre supplémentaire

- Autre solution: gérer le cas spécial où la fonction a été appelée avec moins de paramètres:

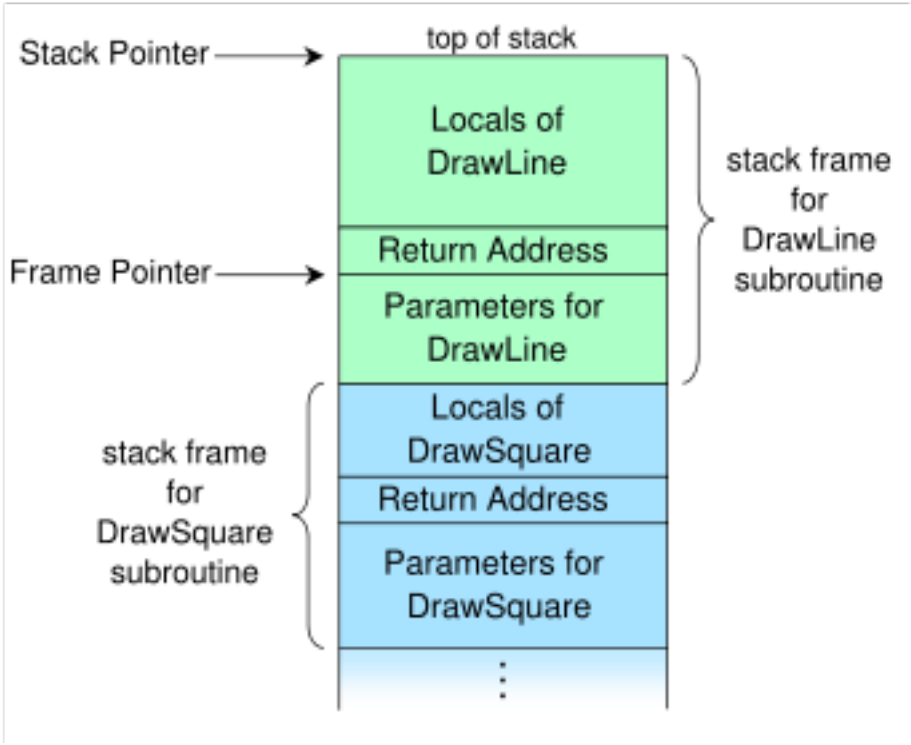
```
function parcoursCroissant(tableau, index)
  {
    if (arguments.length<2)
       parcoursCroissant(tableau, 0);
    else if( index >= 0 && index < tableau.length )
       {
       print( tableau[index]);
       parcoursCroissant(tableau, index + 1);
       }
  }
```

ou bien:

```
function parcoursCroissant(tableau, index)
  {
    if (arguments.length<2)
       index = 0;
    if( index >= 0 && index < tableau.length )
       {
       print( tableau[index]);
       parcoursCroissant(tableau, index + 1);
       }
  }
```

Que se passe-t-il lors d'un appel récursif?

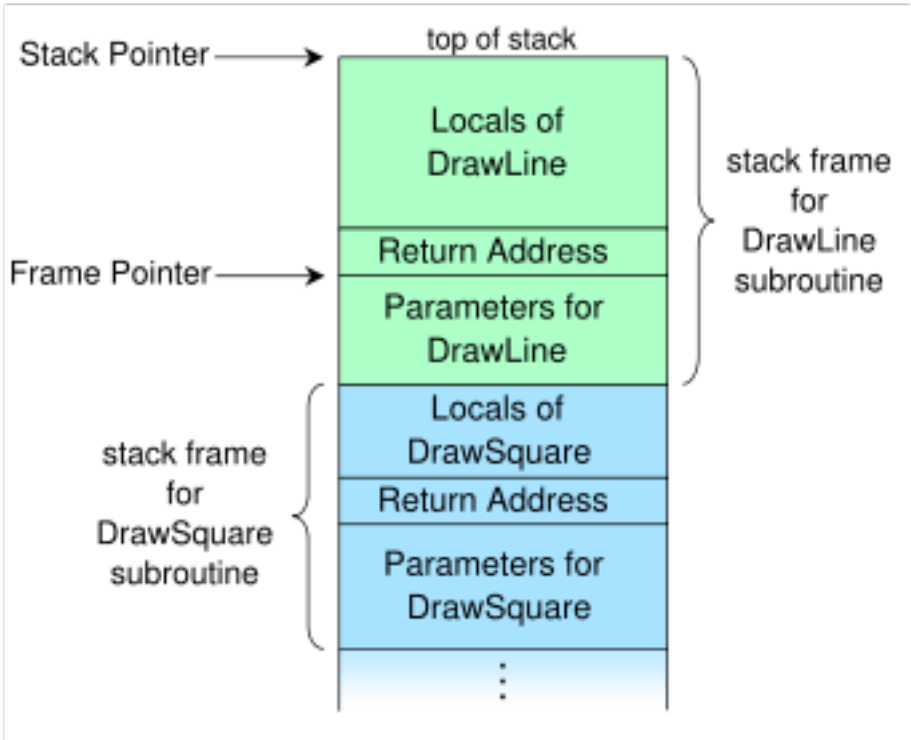
- Notion de **pile d'appel** (call stack)
- Lorsqu'une ***fonction appelante*** appelle une ***fonction appelée***, le contexte de la fonction appelante demeure sur la pile.



28

Que se passe-t-il lors d'un appel récursif?

- La **pile d'appel** contient:
  - toutes les variables locales et paramètres de la fonction appelante, incluant *this*
  - le point (adresse) de retour (là où l'exécution doit se poursuivre dans la fonciton appelante une fois que la fonciton appelée a termriné).



29

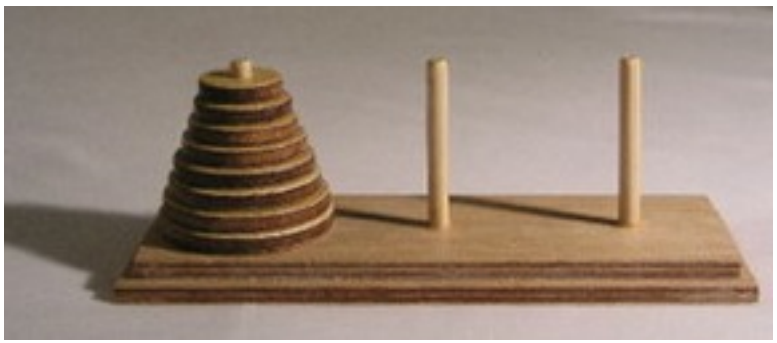
Limites de récursivité

- Attention la pile d'appel peut se remplir rapidement!
- On obtient alors une erreur de **dépassement de pile (stack overflow)**
- Les implémentations de certains langages permettent dans **certains cas** des récursions qui ne remplissant pas la pile. (optimisation de «récursion terminale», *tail call optimisation*)

30

Un problème plus complexe

- Tours de Hanoi:



- Problème: déplacer les n disques d'une tour à une autre
- Contraintes
  - Déplacer un disque à la fois
  - Plus gros disque jamais sur de plus petits

Origine

Une légende raconte qu'un groupe de moines, gardien des trois tours, doivent déplacer **64 disques d'or** d'une tour à une autre en respectant la règle unique qui stipule qu'un anneau ne doit pas reposer sur un anneau plus petit. Lorsqu'ils auront déplacé les 64 anneaux...

Ne vous inquiétez pas!

- On n’a jamais trouvé le temple ni les moines...
  à Hanoi au Vietnam,

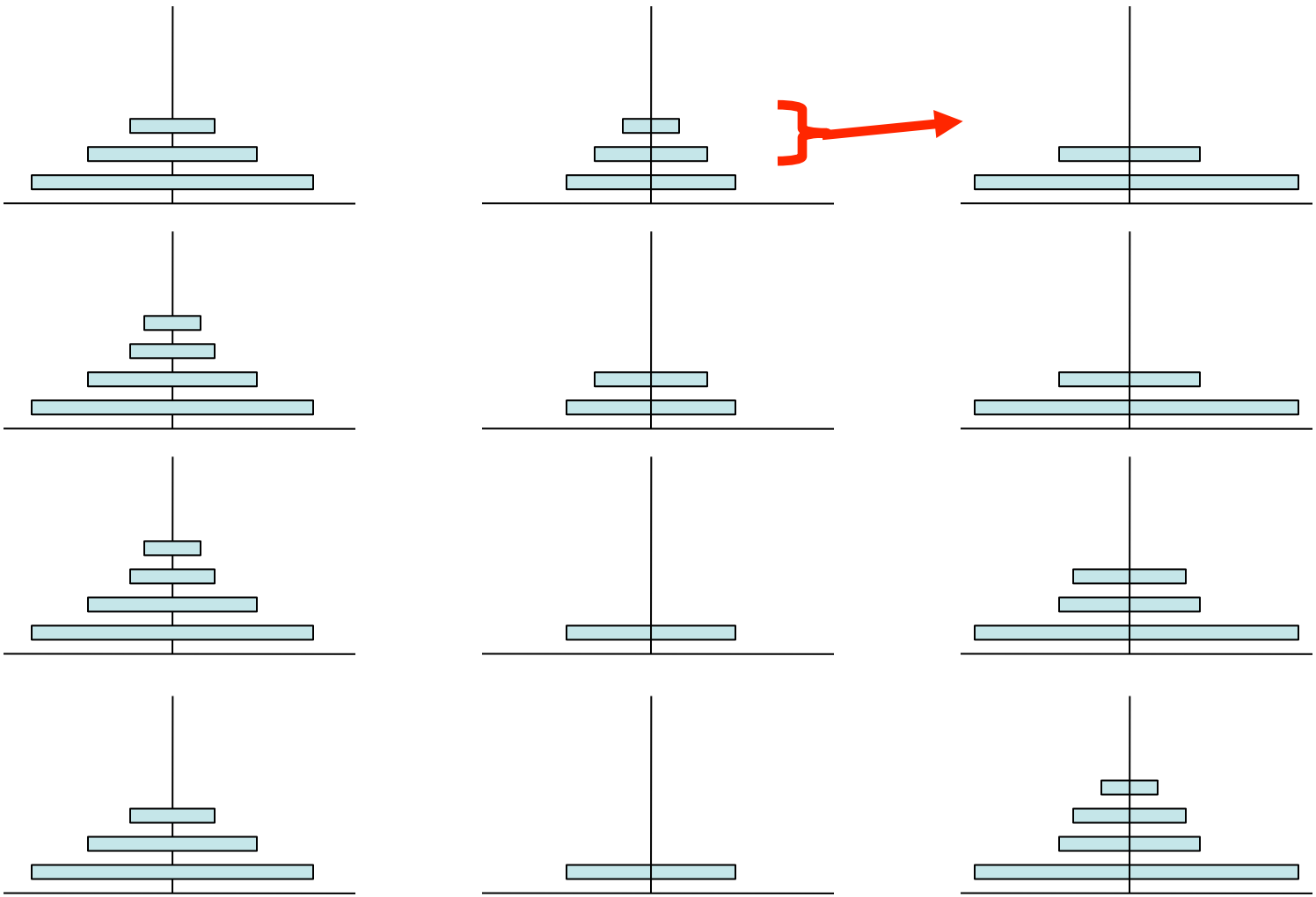
Demeurez-vous inquiets?

- Déplacer les 64 disques nécessite un minimum de 2^{64}-1 déplacements
- Si déplacer un disque prend une seconde...
- Déplacer les 64 disques prendra environ 584,5 milliards d'années
  (43 fois l'âge estimé de l'univers)

Et puis la légende a (probablement) été *inventée* par un mathématicien Français Édouard Lucas

34

Situations typique



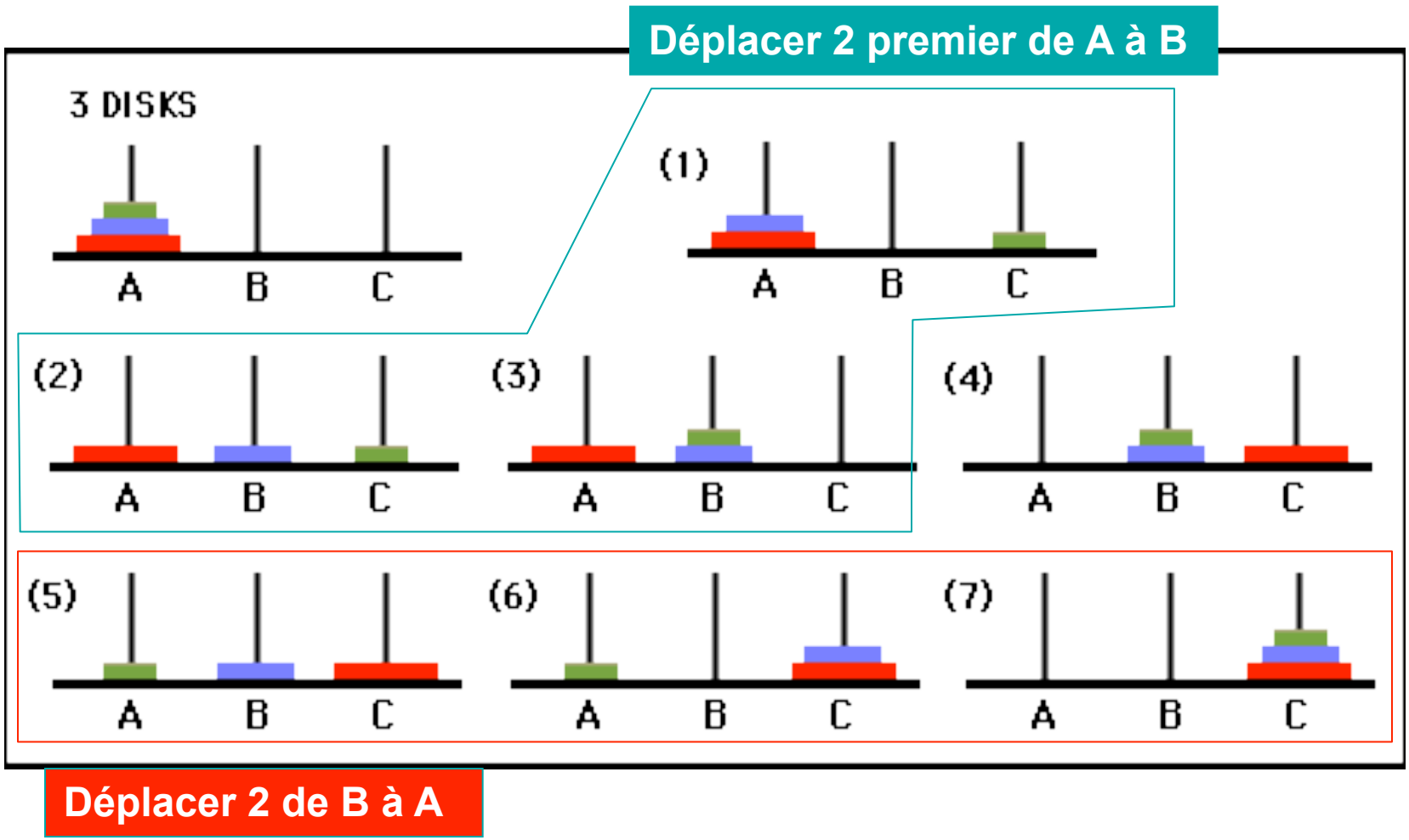
- Libérer le dessus
- Déplacer le dernier voulu
- Déplacer les autres

Décomposition

Déplacer n disque de tour A à C

- Si n=1: déplacer le disque directement
  - Cas simple, condition d'arrêt de récursion
- Sinon
  - Déplacer n-1 disque de A à B
    - Problème moins complexe
  - Déplacer 1 disque de A à C
    - Problème résoluble directement
  - Déplacer n-1 disque de B à C
    - Problème moins complexe

Illustration



Peudocode pour JavaScript

- Stocker n disques dans un tableau de int
- Utiliser des entiers pour désigner la grandeur des disques
- E.g.:
  - tour[1] = [4,3,2,1];
  - Mieux: le générer automatiquement
- Supposons 3 tours: tour[1], tour[2], tour[3]
- function deplacer(n,i,j): // déplacer n disque de la tour i à la tour j
    - Si n=1: déplacer un disque de la tour[i] à la tour[j]
    - Sinon:
      - deplacer(n-1, i, 6-i-j)
      - deplacer(1, i, j)
      - deplacer(n-1, 6-i-j, j)
- Utilisation (Test)
  - Générer 3 tours
  - Appeler déplacer