R1.01

INITIATION AU DÉVELOPPEMENT

Cours 8 : Introduction à la récursivité

Hervé Blanchon & Anne Lejeune

Université Grenoble Alpes

IUT 2 – Département Informatique

Sommaire

- Premier exemple pratique : fonction factorielle
- Vérifier qu'un entier est pair sans utiliser la division
- Les tours de Hanoï
  - un grand classique
- Les approches possibles pour résoudre un problème en utilisant un algorithme récursif
- Approche diminuer pour régner
  - exemple de la recherche dichotomique
- Approche diviser pour régner
  - exemple pédagogique du maximum

Fonctions intrinsèquement récursives

FACTORIELLE

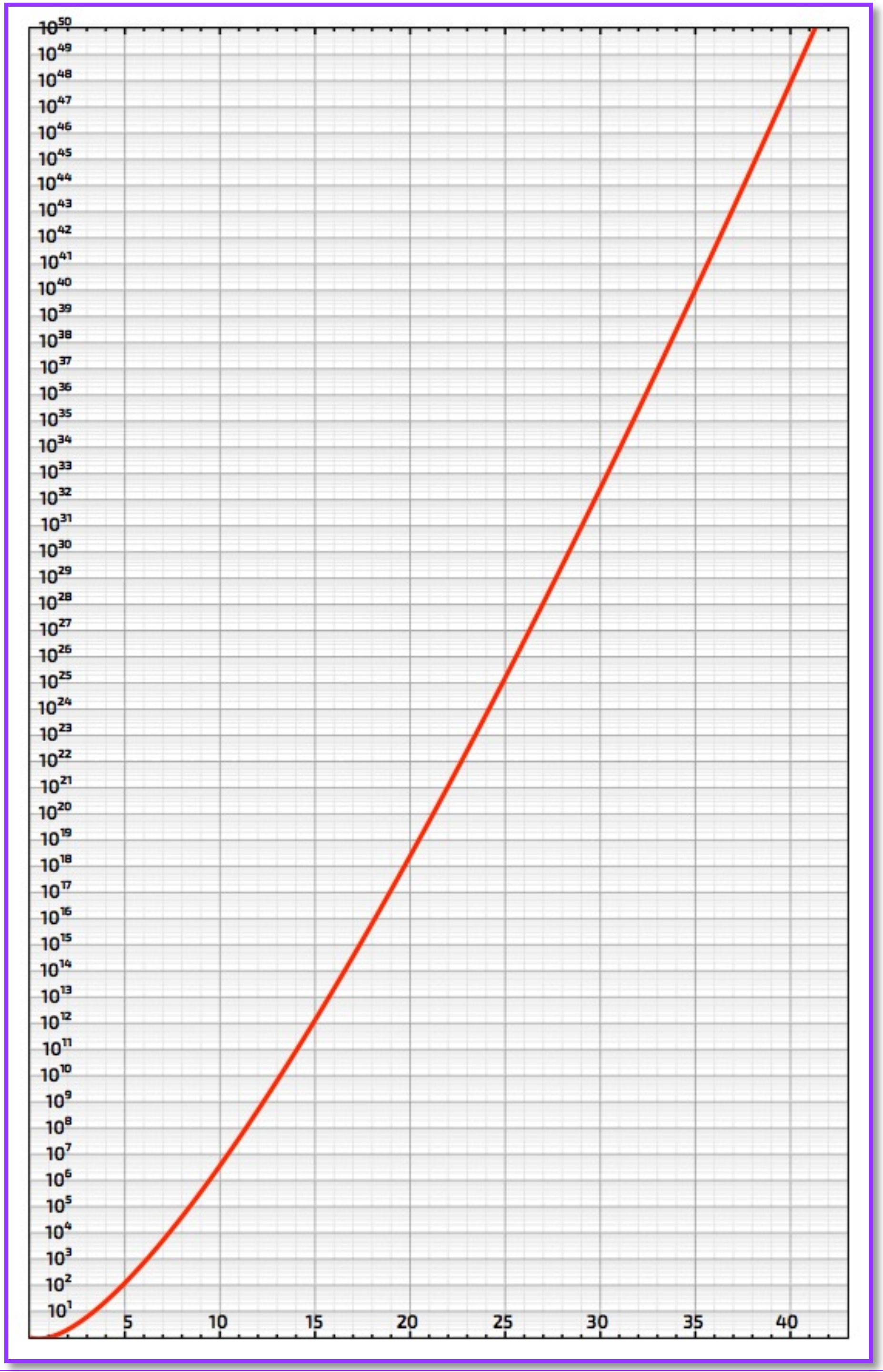
Définition…

- …en langue naturelle
  - La factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n …
  - … elle se note $n!$
- …formelle
  - $n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$

- Note
  - $0! = 1$
    - par convention, le produit vide est égal à l'élément neutre de la multiplication

Une fonction qui croît très vite

- Exemples
  - 0! = 1
  - 5! = 120
  - 10! = 3 628 800
    - 3 millions
  - 15! = 1 307 674 368 000
    - 1 million de millions
  - 20! = 2 432 902 008 176 640 000
    - 2 millions de millions de millions
- Utilité de savoir cela ?
  - $n!$ est le nombre de données que doit manipuler un algorithme qui doit examiner toutes les permutations de n éléments



Un algorithme itératif

Conception de l'algorithme itératif

Invariant	$r = 1 \times 2 \times \ldots \times (i-1)$
Situation Finale	➤ i == n + 1 ⇨ return r; $// r = 1 \times 2 \times \ldots \times (n-1) \times n$

Situation Intermédiaire	➤ i <= n ⇨ r := r × i; // traiter
	i := i + 1; // avancer ➡ **Invariant vérifié**

Itération	while (i <= n) {}
Initialisation	i = 1; r = 1; ➡ **Invariant vérifié** $// r = 0!$

L’implantation itérative en Java

```java
private static
int factorielleIter(int n) {
    int r = 1;
    int i = 1;
    while (i <= n) {
        r = r * i;
        i = i + 1;
    }
    return r;
}
```

Voir les choses autrement

- La définition formelle…
  - $n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$
- … peut se réécrire :
  - $n! = \prod_{i=1}^{n} i = (1 \times 2 \times 3 \times \cdots \times (n-1)) \times n$

- Or : $1 \times 2 \times 3 \times \cdots \times (n-1) = (n-1)!$
- Donc : $n! = (n-1)! \times n$

- En notation fonctionnelle on a :
  - factorielle(n) = factorielle(n−1) × n
    - c'est une définition dite **récursive**, le calcul de la fonction factorielle nécessite l'appel de la fonction factorielle

Qu'est ce qui se passe ?

- On a vu que :
  - factorielle(n) = factorielle(n−1) × n
- Une trace du calcul
  - factorielle(5) = factorielle(4) x 5
    - avec factorielle(4) = factorielle(3) x 4
      - avec factorielle(3) = factorielle(2) x 3
        - avec factorielle(2) = factorielle(1) x 2
          - avec factorielle(1) = factorielle(0) x 1
            - avec factorielle(0) = 1
            - Ouf ! un cas d'arrêt des appels récursifs (calcul trivial) !
          - soit factorielle(1) = 1 x 1 = 1
        - soit factorielle(2) = 1 x 2 = 2
      - soit factorielle(3) = 2 x 3 = 6
    - soit factorielle(4) = 6 x 4 = 24
  - Soit factorielle(5) = 24 x 5 = 120

Descente

Remontée

avoir au moins une situation à résultat trivial

Vers une formalisation

- On a vu :
  - une définition de factorielle qui utilise factorielle
  - une situation triviale (de base) qui permet de produire le résultat sans calcul
- On a donc :
  - factorielle(0) = 1 la **BASE**
  - factorielle(n) = factorielle(n-1) x n la **RÉCURRENCE**
- On peut formaliser comme suit :
  - ➤ n = 0 ⇨ résultat = 1 (**BASE**)
  - ➤ n > 0 ⇨ résultat = factorielle(n-1) x n (**RÉCURRENCE**)

Vers une formalisation

- La formalisation :
  - ➤ $n = 0$ ⇨ ✳ {résultat = 1} (**BASE**)
  - ➤ $n > 0$ ⇨ ✳ {résultat = factorielle(n-1) x n} (**RÉCURRENCE**)

La Preuve !

- on a une ou plusieurs situations triviales de base
- la valeur (taille) du paramètre diminue strictement lors des appels récursifs et il y a convergence vers une situation de base
- le calcul de la récurrence est juste

L’implantation récursive en Java

```
private static
int factorielleRec(int n) {
    if (n == 0) {
        // base pour l'arrêt
        return 1;
    } else {
        // utilisation de la définition
        return factorielleRec(n - 1) * n;
    }
}
```

Derniers mots…

- Pour écrire une fonction récursive (`fctRec`) ou une procédure récursive (`procRec`) pour résoudre le problème P…
  - il faut imaginer qu'elle existe déjà et qu'elle calcule la réponse au problème P
  - il faut exprimer la résolution du problème P sur des données **D** en fonction de la résolution du même problème sur des données réduites **d** plus petites (récurrence)
  - il faut trouver une (resp. plusieurs données) **dt** pour laquelle (resp. lesquelles) on connaît la réponse au problème (base) sans calcul
  - il faut s'assurer que les réductions successives des données (passage de **D** à **d**) convergent vers les données **dt**

Derniers mots…

- Réduire la taille des données **D** vers **d** ?
- Quelques pistes…
  - **D** est une valeur numérique
    - **d** est une **valeur numérique plus petite** qui se rapproche d’une valeur **dt** pour laquelle on connaît la réponse au problème P
  - **D** est une vecteur
    - **d** est un **vecteur plus petit** qui se rapproche d’un vecteur **dt** pour lequel on connaît la réponse au problème P
  - **D** est une liste (on verra plus tard)
    - **d** est une **liste plus petite** qui se rapproche d’une liste **dt** pour laquelle on connaît la réponse au problème P

Réflexion récursive sur un problème qui ne semble pas récursif

VÉRIFIER QU'UN ENTIER EST PAIR

SANS UTILISER LA DIVISION

`estPair()` : le problème *(à compléter)*

- On veut écrire la fonction suivante :

```
private static boolean estPair(int val) {
// {val entier positif} =>
// {résultat = true si val est paire, false
//            sinon sans utiliser la division}
```

- Réfléchir… *(indice : ne pas penser en termes de produit)*
  - comment sait-on que val entier est pair ?
  - comment sait-on que val entier n'est pas pair (il est impair) ?

`estPair()` : base et récurrence *(à compléter)*

BASE

- connaît-on un ou plusieurs entiers pour le(s)quel(s) on sait sans calcul qu'il(s) est (sont) pair(s) ?

RÉCURRENCE

- peut-on exprimer la fonction estPair(x) en utilisant la fonction estPair(y) où la valeur de y dépend de la valeur de x ?

estPair() : le code en Java *(à compléter)*

```
private static boolean estPair(int val) {
    // {val entier positif} =>
    // {résultat = true si val est paire, false sinon
    //             sans utiliser la division}








}
```

Un grand classique !

LES TOURS DE HANOÏ

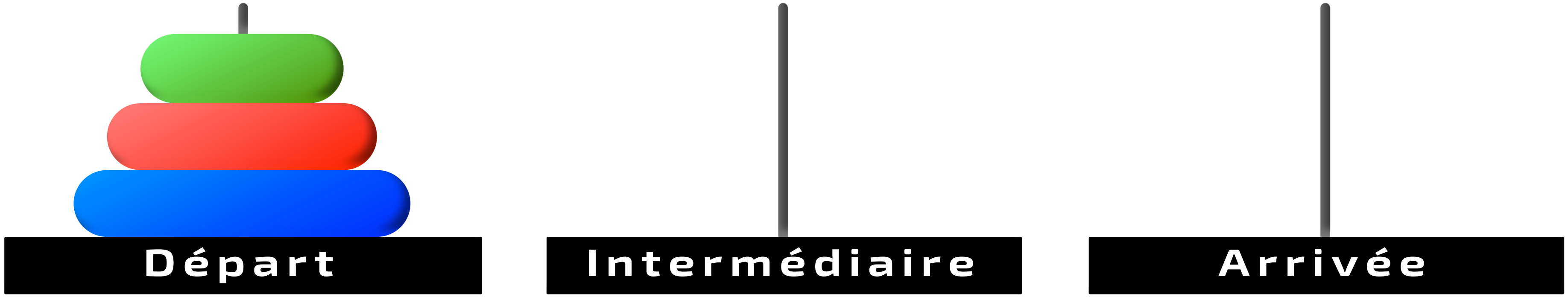
Un jeu, d’apparence anodine…

- Son origine
  - le problème des tours de Hanoï est un jeu de réflexion imaginé par le mathématicien français Édouard Lucas
  - le problème est dû à un de ses amis, N. Claus de Siam, prétendument professeur au collège de Li-Sou-Stian ; une double anagramme de Lucas d'Amiens, sa ville de naissance, et Saint Louis, le lycée où Lucas enseignait
- Le jeux et ses règles
  - déplacer des disques de diamètres différents d'une tour de « départ » à une tour d’« arrivée » en passant par une tour « intermédiaire », et ceci en un minimum de coups, tout en respectant les règles suivantes :
    - **on ne peut déplacer plus d'un disque à la fois,**
    - **on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide**
  - On suppose que cette dernière règle est également respectée dans la configuration de départ

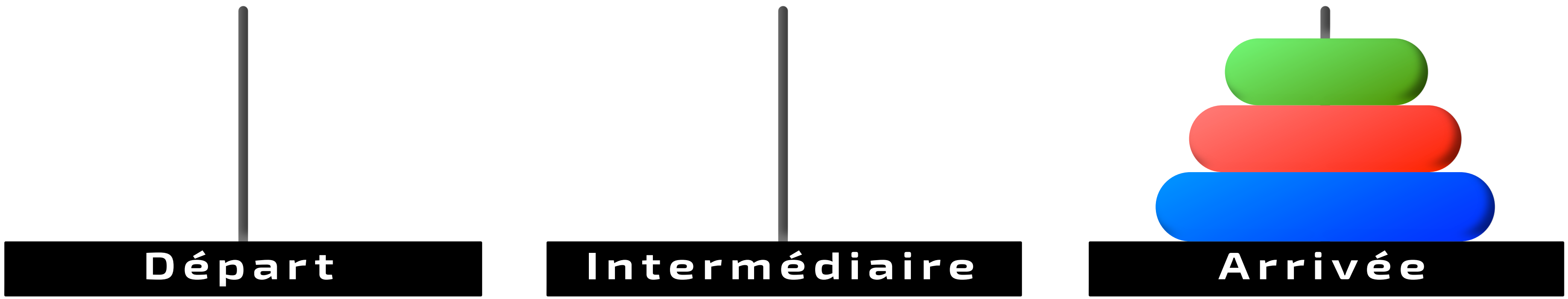
Source : Wikipédia consulté le 20/10/2015

Un exemple avec 3 disques

- Situation initiale

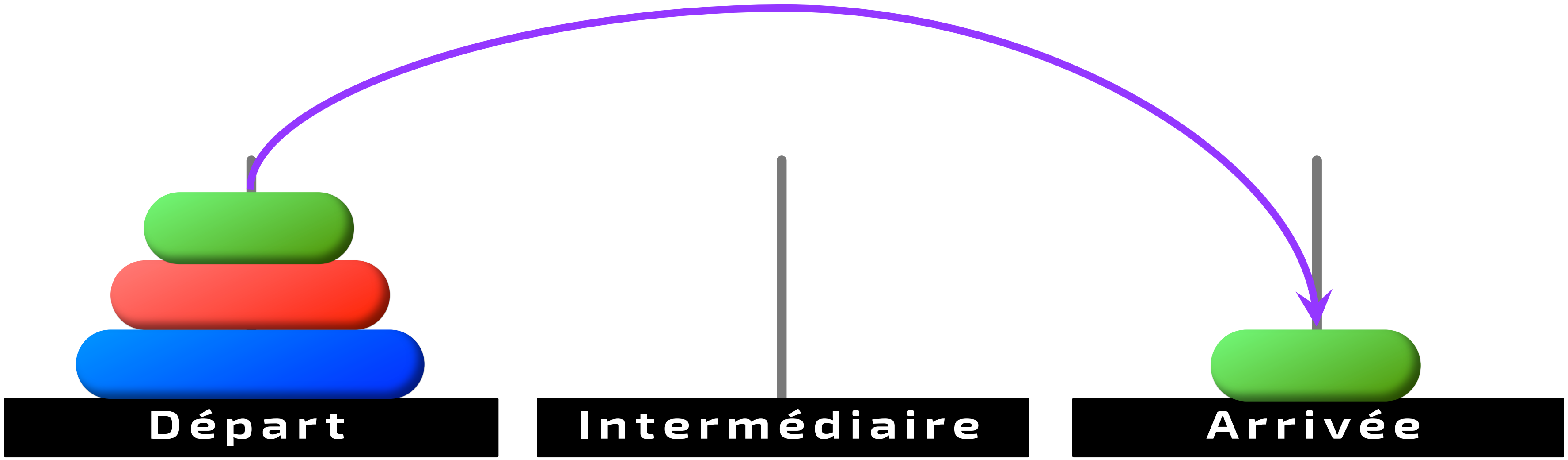


- Situation finale



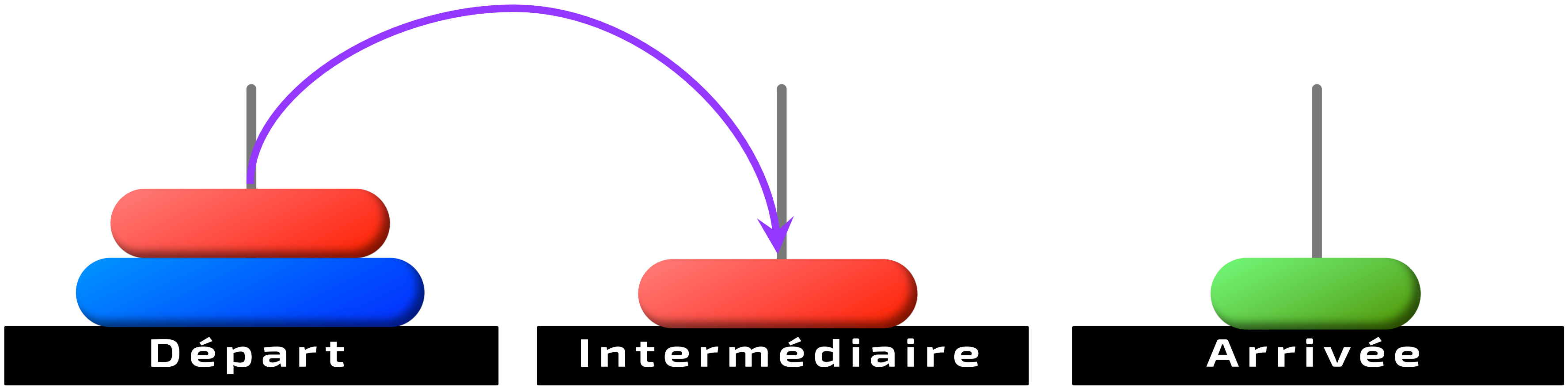
Les étapes de la résolution

- Étape 1



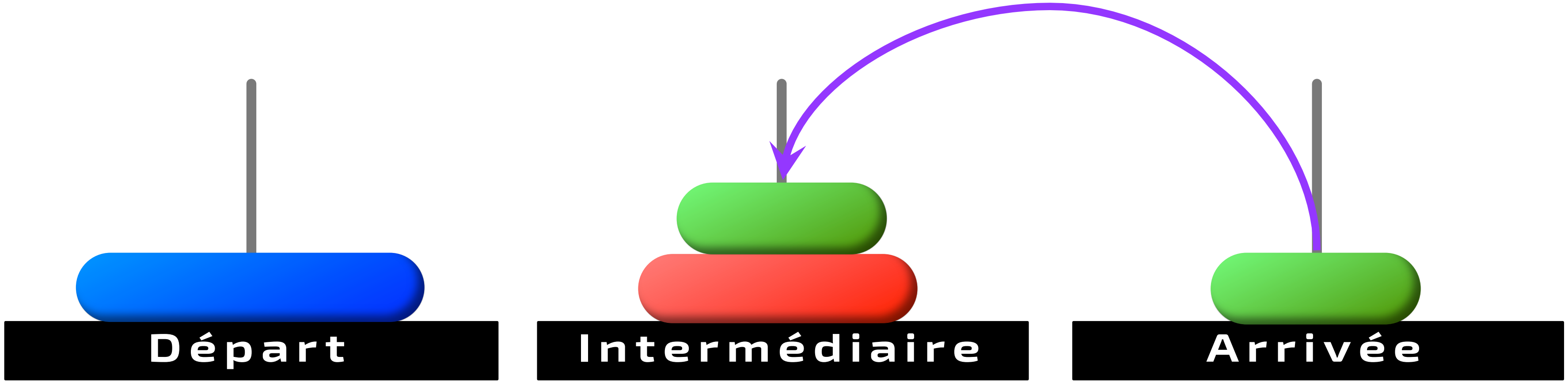
Les étapes de la résolution

- Étape 2



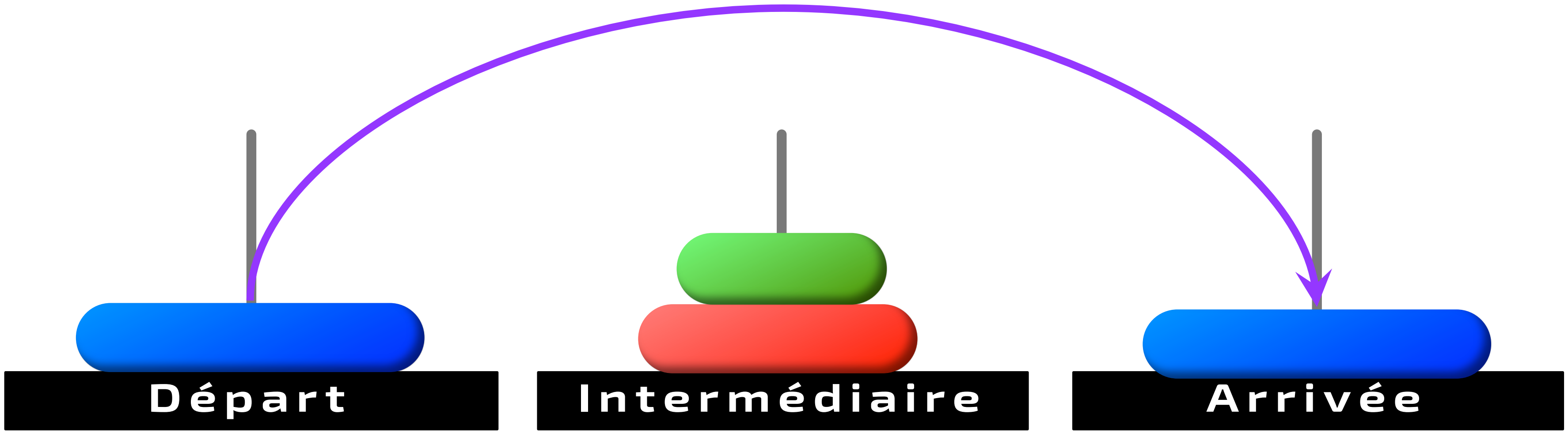
Les étapes de la résolution

- Étape 3



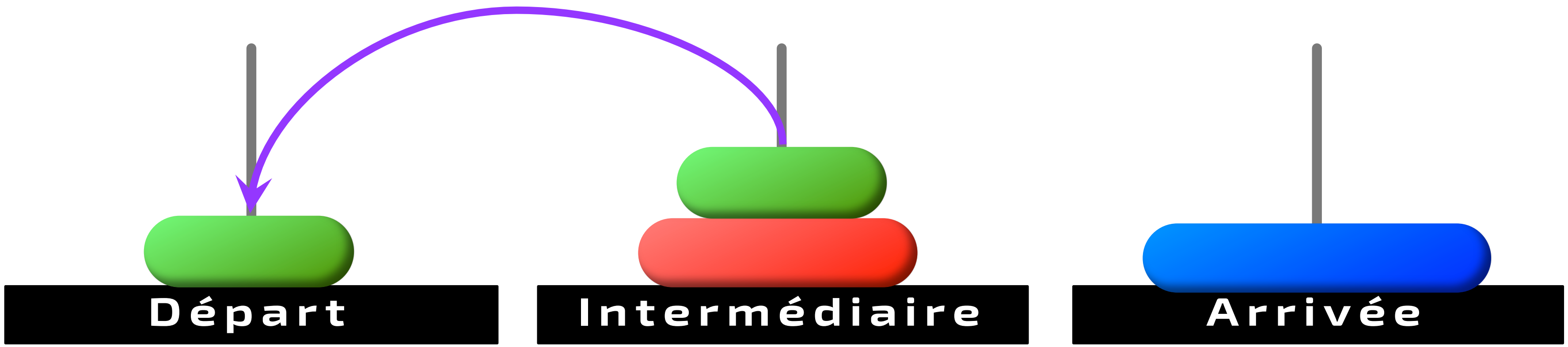
Les étapes de la résolution

- Étape 4



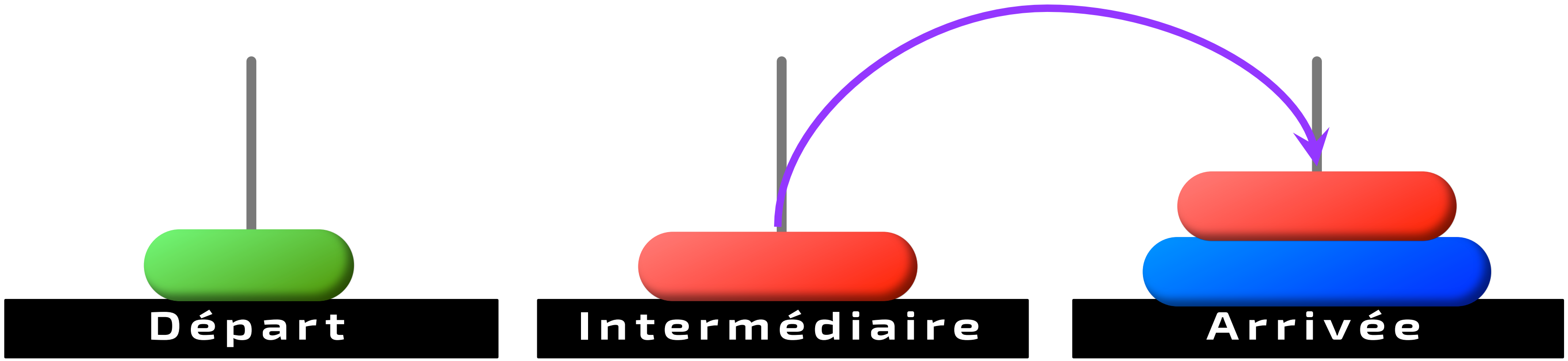
Les étapes de la résolution

- Étape 5



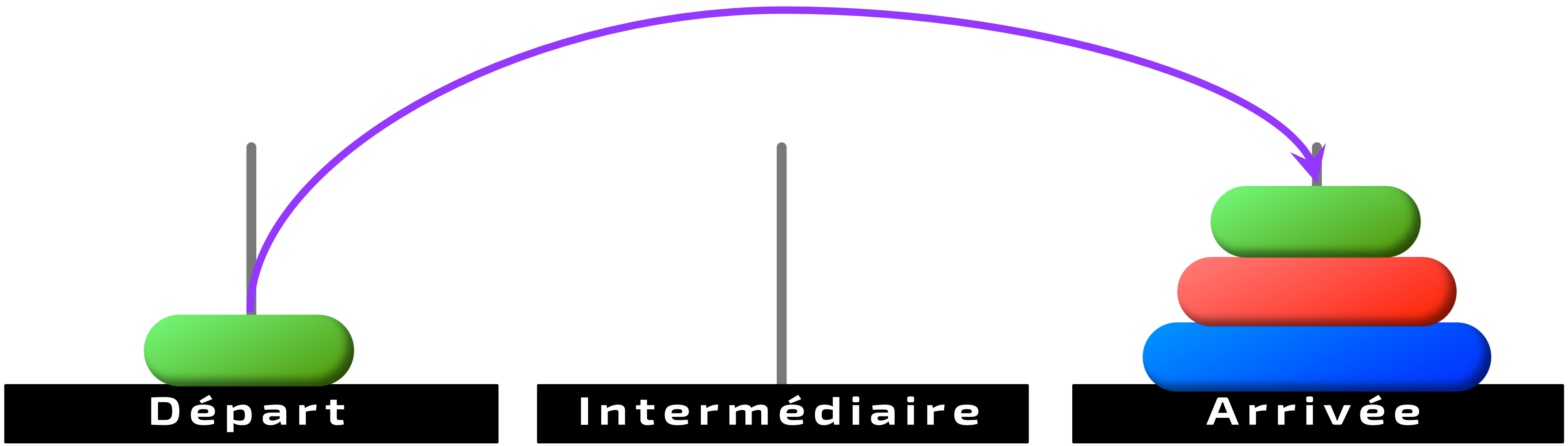
Les étapes de la résolution

- Étape 6



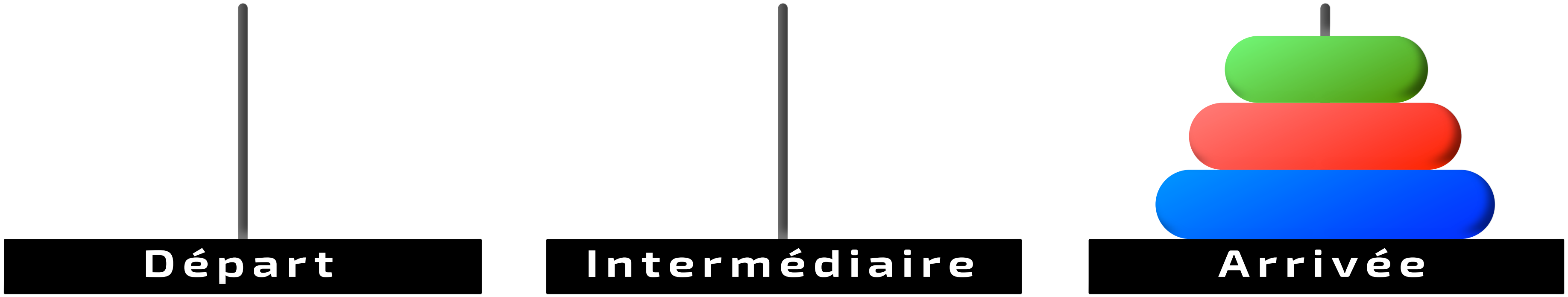
Les étapes de la résolution

- Étape 7



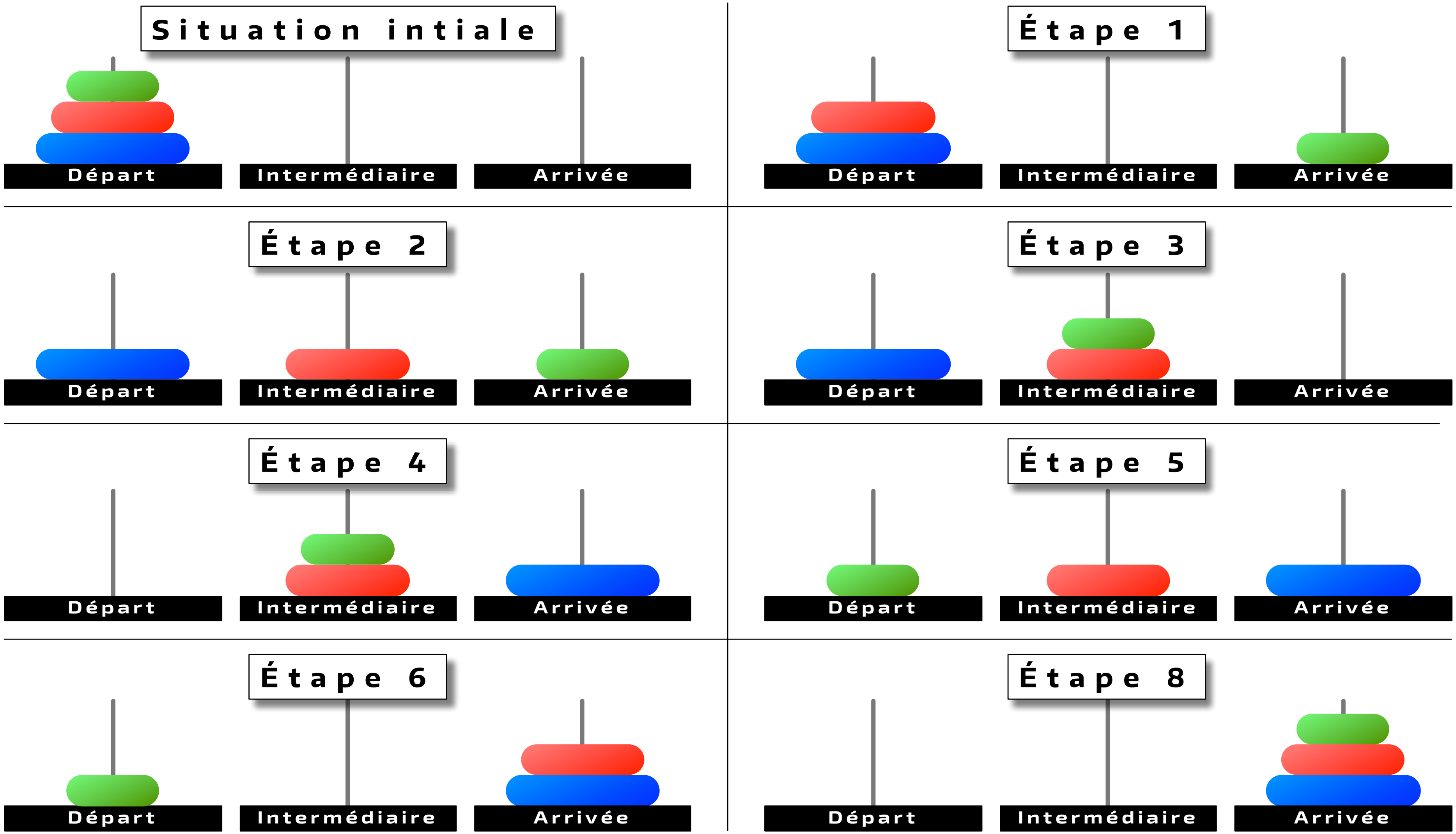
Les étapes de la résolution

- Situation finale



Les étapes de la résolution (récapitulatif)

- Avec 3 disques on a les 7 étapes suivantes :



La procédure à réaliser

- Procédure demandée

```
private static void hanoi(int nbDisques,
                          char depart,
                          char intermediaire,
                          char arrivee)
```

  - qui écrit sur le terminal les étapes de la résolution du problème
    - où **`nbDisques`** est le nombre de disques en jeu dans le problème & **`depart`**, **`intermédiaire`** et **`arrivee`** sont les noms respectifs des piquets de départ, intermédiaire et d'arrivée
- Appel pour **`3`** disques & des piquets **`D`**, **`I`**, **`A`**
  - **`hanoi(3, 'D', 'I', 'A');`**

La procédure à réaliser

- Appel
  - `hanoi(3, 'D', 'I', 'A');`

- Trace d'exécution

```
Déplacement disque 1 du piquet D vers le piquet A    -- étape 1
Déplacement disque 2 du piquet D vers le piquet I    -- étape 2
Déplacement disque 1 du piquet A vers le piquet I    -- étape 3
Déplacement disque 3 du piquet D vers le piquet A    -- étape 4
Déplacement disque 1 du piquet I vers le piquet D    -- étape 5
Déplacement disque 2 du piquet I vers le piquet A    -- étape 7
Déplacement disque 1 du piquet D vers le piquet A    -- étape 8
```

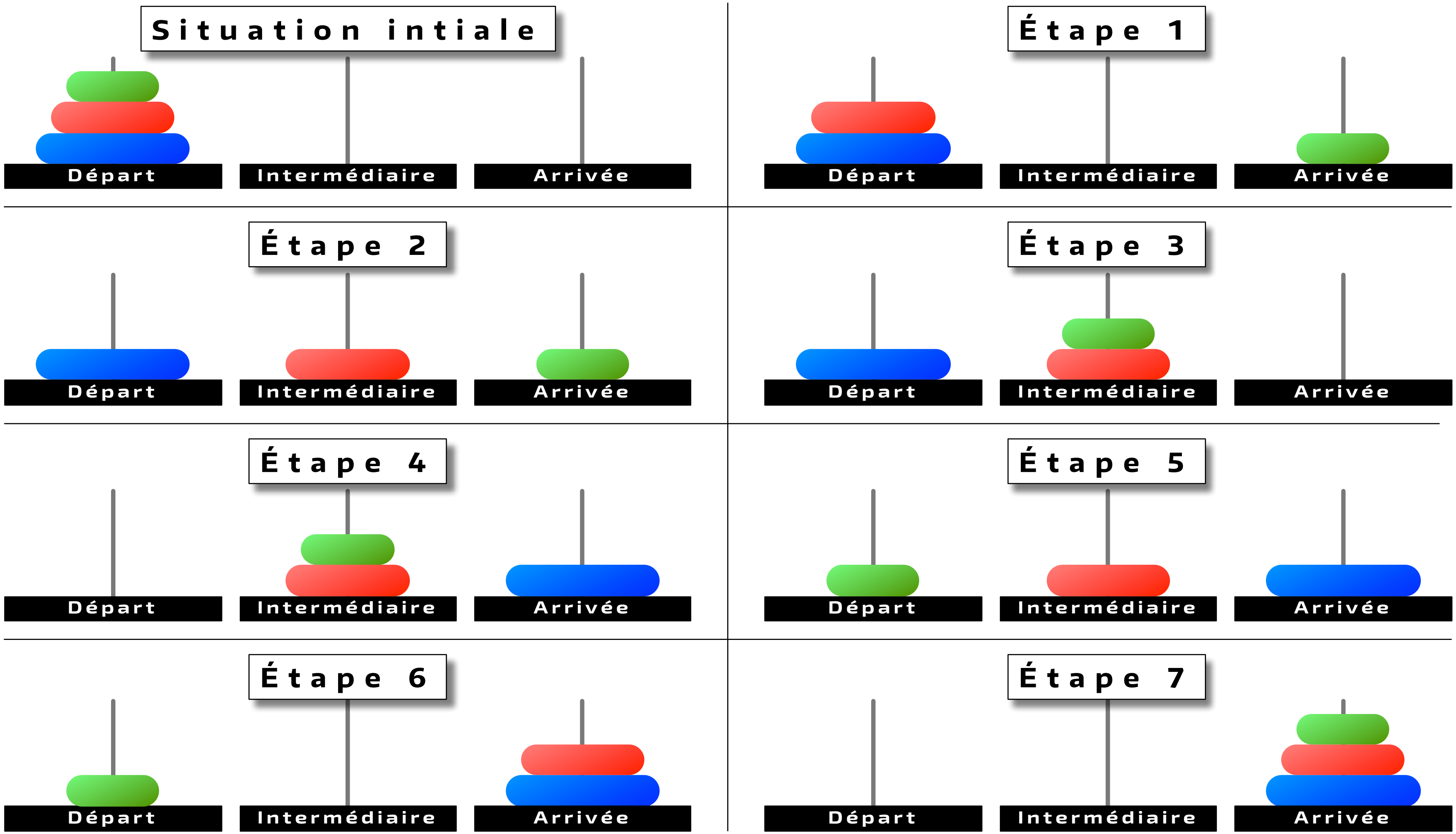
Vers une version récursive : Base

- Connaît-on une situation des données pour laquelle la résolution du problème est triviale ?
- Oui !
  - laquelle ? (à vous …)

- Version algorithmique de la base :

Vers une version récursive : Récurrence

- États intermédiaires intéressants ?



Vers une version récursive : Récurrence

- Version algorithmique de la référence (à vous …)

Écriture d’algorithmes récursifs

LES APPROCHES POSSIBLES

Pour aller plus loin…

- Les algorithmes récursifs sont habituellement rangés en deux classes
- Ce qui distingue ces deux classes est le constat suivant
  - pour résoudre le problème initial sur des données D, il suffit de résoudre le problème sur des données d_i plus « petites »
    - c’est le cas de la recherche dichotomique
  - pour résoudre le problème initial sur des données D, il faut résoudre deux fois le problème sur des données d1 & d2 (telles que $d1 \cup d2 = D$ et $d1 \cap d2 = \emptyset$) et utiliser les résultats obtenus pour obtenir le résultat sur D (on coupe habituellement en 2 mais pas toujours !)
    - on va voir un exemple pédagogique

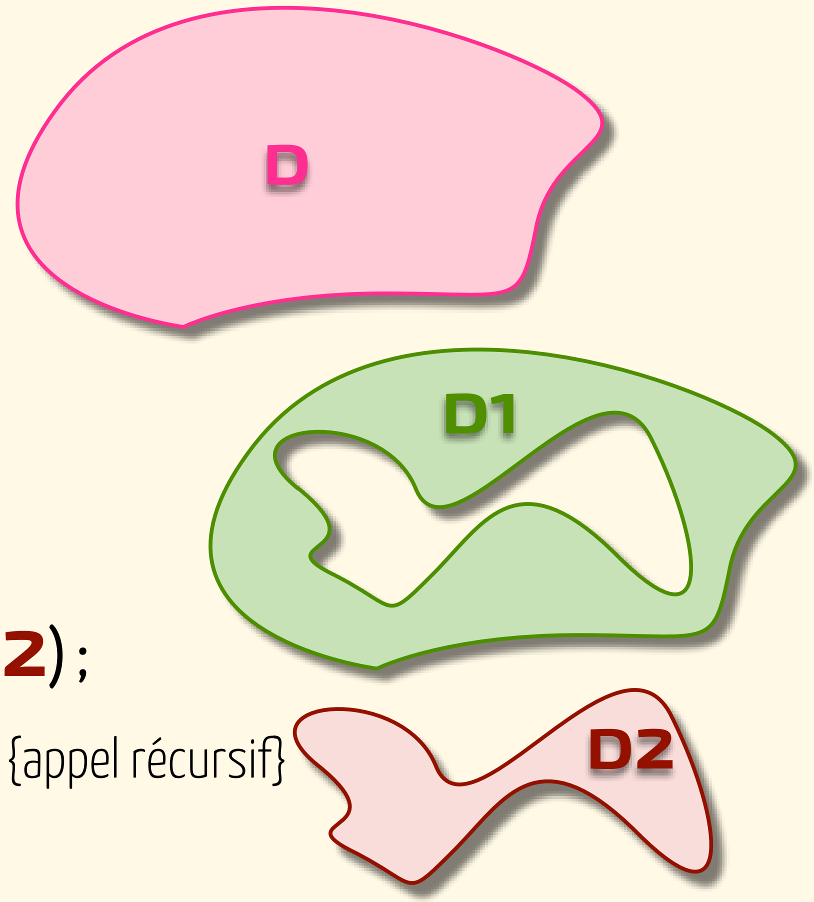
Quand il suffit que je m’occupe de données plus petites

DIMINUER POUR RÉGNER

Principe

- Modèle d'algorithme pour résoudre le_problème sur des données D

```
Résoudre_le_problème(D)
 début
    si estTriviale(D) alors
       Résultat ← résultat donné sans calcul ;
    sinon
       (D1, D2) ← Diviser(D) ;
       Dx ← Choisir_sous_problème(D1, D2) ;
       Résultat ← Résoudre_le_problème(Dx) ; {appel récursif}
    finsi ;
 fin ;
```



recherche dichotomique
en diminuer pour régner

Rappel du problème (cf. cours 6 — partie 3)

- C'est une recherche de la position la plus à gauche d'une valeur `val` dans un v vecteur trié
  - résultat = position la plus à gauche de `val` si `val` est dans v
  - résultat = -position_que_val_devrait_occuper sinon
- On a déjà résolu ce problème avec une approche itérative

Version itérative : le code

```
private static int indiceValDichoIterative(ArrayList<Integer> v, int val) {
// {v trié croissant non vide} =>
// {résultat = indice le plus à gauche de val si val est dans v ;
//              -indice que val devrait occuper si val n’est pas dans v}
  if (v.get(v.size() - 1) < val) {  // v.[v.size()-1] < val
    return -v.size();    // val devrait occuper l’indice v.size()
  } else {                          // v.[v.size()-1] ≥ val
    int inf = 0;
    int sup = v.size() - 1;
    // v[0 .. -1] < val ≤ v[v.size()-1 .. v.size()-1] -> invariant vérifié
    int m;
    while (inf < sup) {
      m = (inf + sup) / 2;
      if (v.get(m) >= val) {     // v[m] ≥ val
        sup = m;     // poursuivre la recherche à gauche sur [inf..m-1]
      } else {                   // v[m] < val
        inf = m + 1; // poursuivre la recherche à droite sur [m+1..sup-1]
      }
      // v[0 .. inf-1] < val ≤ v[sup .. v.size()-1] -> invariant vérifié
    }
    // inf = sup ; v[0 .. sup-1] < val ≤ v[sup .. v.size()-1] -> invariant
    // production du résultat
    if (v.get(sup) == val) {
      return sup;   // val trouvée
    } else {
      return -sup;  // val pas trouvée, val aurait été à l'indice sup
    }
  }
}
```

Version récursive : le modèle

- Contrairement à la version itérative
  - une fonction « point d'entrée »
    - `int indiceValDichoRec(ArrayList<Integer> v, int val)`
    - qui cherche l'indice sur tout le vecteur v
  - une fonction récursive qui fait le travail (worker)
    - `int indiceValDichoRec(ArrayList<Integer> v, int val, int borneInf, int borneSup)`
    - qui cherche l'indice sur une tranche [borneInf .. borneSup] du vecteur v
- En effet
  - la « recherche » a besoin de connaître la borne **`inf`** et la borne **`sup`** de l'intervalle de recherche qui va changer à chaque appel récursif
- Note
  - la situation v[v.size()-1] < val (résultat = -(v.size()+1)) sera traitée dans la fonction « point d'entrée »

Version récursive : le modèle

- On aura donc le point d’entrée :

```
private static
int indiceValDichoRec(ArrayList<Integer> v, int val) {
  if (v.get(v.size() - 1) < val) {
    // inutile de chercher (le plus grand de v < val)
    return -v.size();
  } else {
    // on cherche initialement sur tout le vecteur
    // v[0..v.size()-1]
    return indiceValDichoRecWorker(v, val, 0, v.size() - 1);
  }
}
```

Version récursive : le worker

- Rappels
  - précondition du worker
    - position de val dans v ≤ v.size() (présente ou absente)
  - recherche entre inf et sup
- Raisonnement
  - ➤ inf > sup ⇨ (BASE, recherche dans intervalle vide)
    - ➤➤ v[inf] = val ⇨ résultat = inf
    - ➤➤ v[inf] ≠ val ⇨ résultat = -inf
  - ➤ inf ≤ sup ⇨ (RÉCURRENCE)
    - m ← (inf+sup)/2
    - ➤➤ v[m] < val ⇨ résultat = indiceDichoRecWorker(v, m+1, sup, val)
      - suite de la recherche à droite de m
    - ➤➤ v[m] ≥ val ⇨ résultat = indiceDichoRecWorker(v, inf, m, val)
      - suite de la recherche à gauche de m, m compris

Version récursive : code du worker

```
private static
int indiceValDichoRecWorker(ArrayList<Integer> v,
                            int val,
                            int borneInf,
                            int borneSup) {
  if (borneInf == borneSup) {                   // BASE
    if (v.get(borneInf) == val) {
      return borneInf;          // trouvé
    } else {
      return -borneInf;         // absent
    }
  } else {                                      // RÉCURRENCE
    int m = (borneInf + borneSup) / 2; // point milieu
    if (v.get(m) >= val) {
      // chercher dans la moitié inférieure (à gauche)
      return indiceValDichoRecWorker(v, val, borneInf, m);
    } else {
      // chercher dans la moitié supérieure (à droite)
      return indiceValDichoRecWorker(v, val, m + 1, borneSup);
    }
  }
}
```

Procédure main de trace

```java
public static void main(String[] args) {
  ArrayList<Integer> vDicho = new ArrayList<>();
  for (int i = 1; i < 500; i = i + 2) {
    if (i % 9 != 0) {vDicho.add(i);} if (i % 7 == 0) {vDicho.add(i); // 9 fois}
  }
  System.out.println("le vecteur : " + vDicho);
  System.out.printf("%-12s %3s %-5s\n",
                    "Place de 1", ":", indiceValDichoRec(vDicho, 1));
  System.out.printf("%-10s %3s %-5s\n",
                    "Place de 900", ":", indiceValDichoRec(vDicho, 900));
  System.out.printf("%-12s %3s %-5s\n",
                    "Place de 35", ":", indiceValDichoRec(vDicho, 35));
  System.out.printf("%-10s %3s %-5s\n",
                    "Place de 243", ":", indiceValDichoRec(vDicho, 243));
  System.out.printf("%-10s %3s %-5s\n",
                    "Place de 123", ":", indiceValDichoRec(vDicho, 123));
  System.out.printf("%-10s %3s %-5s\n",
                    "Place de 342", ":", indiceValDichoRec(vDicho, 342));
}
```

Trace obtenue

```
le vecteur : [1, 3, 5, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 11, 13, 15, 17, 19, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 23, 25, 29, 31, 33, 35, 35, 35, 35,
35, 35, 35, 35, 35, 35, 37, 39, 41, 43, 47, 49, 49, 49, 49, 49, 49, 49, 49, 49, 51, 53, 55, 57, 59, 61, 63, 63, 63, 63, 63, 63, 63, 63, 63, 65, 67, 69, 71,
73, 75, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 79, 83, 85, 87, 89, 91, 91, 91, 91, 91, 91, 91, 91, 91, 93, 95, 97, 101, 103, 105, 105, 105, 105, 105,
105, 105, 105, 105, 105, 107, 109, 111, 113, 115, 119, 119, 119, 119, 119, 119, 119, 119, 119, 119, 121, 123, 125, 127, 129, 131, 133, 133, 133, 133,
133, 133, 133, 133, 133, 133, 137, 139, 141, 143, 145, 147, 147, 147, 147, 147, 147, 147, 147, 147, 147, 149, 151, 155, 157, 159, 161, 161, 161, 161,
161, 161, 161, 161, 161, 161, 163, 165, 167, 169, 173, 175, 175, 175, 175, 175, 175, 175, 175, 175, 175, 177, 179, 181, 183, 185, 187, 189, 189, 189,
189, 189, 189, 189, 189, 189, 191, 193, 195, 197, 199, 201, 203, 203, 203, 203, 203, 203, 203, 203, 203, 203, 205, 209, 211, 213, 215, 217, 217, 217,
217, 217, 217, 217, 217, 217, 217, 219, 221, 223, 227, 229, 231, 231, 231, 231, 231, 231, 231, 231, 231, 231, 233, 235, 237, 239, 241, 245, 245, 245,
245, 245, 245, 245, 245, 245, 245, 247, 249, 251, 253, 255, 257, 259, 259, 259, 259, 259, 259, 259, 259, 259, 259, 263, 265, 267, 269, 271, 273, 273,
273, 273, 273, 273, 273, 273, 273, 273, 275, 277, 281, 283, 285, 287, 287, 287, 287, 287, 287, 287, 287, 287, 287, 289, 291, 293, 295, 299, 301, 301,
301, 301, 301, 301, 301, 301, 301, 301, 303, 305, 307, 309, 311, 313, 315, 315, 315, 315, 315, 315, 315, 315, 315, 317, 319, 321, 323, 325, 327, 329,
329, 329, 329, 329, 329, 329, 329, 329, 329, 331, 335, 337, 339, 341, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 345, 347, 349, 353, 355, 357,
357, 357, 357, 357, 357, 357, 357, 357, 357, 359, 361, 363, 365, 367, 371, 371, 371, 371, 371, 371, 371, 371, 371, 371, 373, 375, 377, 379, 381, 383,
385, 385, 385, 385, 385, 385, 385, 385, 385, 385, 389, 391, 393, 395, 397, 399, 399, 399, 399, 399, 399, 399, 399, 399, 401, 403, 407, 409, 411,
413, 413, 413, 413, 413, 413, 413, 413, 413, 413, 415, 417, 419, 421, 425, 427, 427, 427, 427, 427, 427, 427, 427, 427, 427, 429, 431, 433, 435, 437,
439, 441, 441, 441, 441, 441, 441, 441, 441, 441, 443, 445, 447, 449, 451, 453, 455, 455, 455, 455, 455, 455, 455, 455, 455, 457, 461, 463, 465,
467, 469, 469, 469, 469, 469, 469, 469, 469, 469, 469, 471, 473, 475, 479, 481, 483, 483, 483, 483, 483, 483, 483, 483, 483, 483, 485, 487, 489, 491,
493, 497, 497, 497, 497, 497, 497, 497, 497, 497, 497, 499]
Place de 1     : 0
Place de 900   : -546
Place de 35    : 33
Place de 243   : -261
Place de 123   : 135
Place de 342   : -368
```

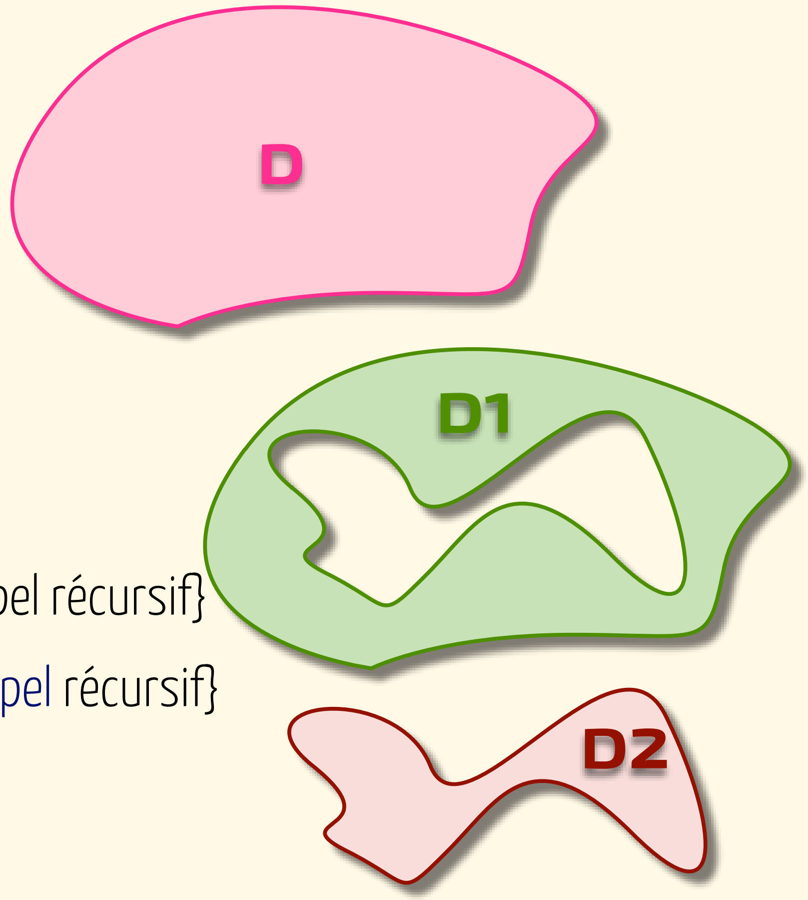
Quand il faut quand même traiter toutes les données

DIVISER POUR RÉGNER

Principe

- Modèle d'algorithme pour résoudre le_problème sur des données D

```
Résoudre_le_problème(D)
 début
   si estTriviale(D) alors
     Résultat ← résultat donné sans calcul ;
   sinon
     (D1, D2) ← Diviser(D) ;
     R1 ← Résoudre_le_problème(D1) ; {appel récursif}
     R2 ← Résoudre_le_problème(D2) ; {appel récursif}
     Résultat ← Fusionner(R1, R2) ;
   finsi ;
 fin ;
```



maximum d’un vecteur en diviser pour régner

Exemple à vertu pédagogique uniquement !

Rappel du problème

- On cherche la plus grande valeur dans un **v** vecteur trié
  - résultat = plus grande valeur de **v**
- On a déjà résolu ce problème avec une approche itérative en TP

Version itérative : le code

```
private static int maximumIter(ArrayList<Integer> v) {
// {v non vide} => {résultat = plus grande valeur de v}

    // le vecteur n'est pas vide, on peut initialiser
    // le max avec le premier élément de v
    int max = v.get(0);
    // on commence le parcours de v à l'indice 1
    int i = 1;
    // invariant -> max = plus grand de v[0 .. i-1]

    // itération de parcours complet
    while (i < v.size()-1) {
        // le maximum est-il à mettre à jour ?
        if (v.get(i) > max) {
             max = v.get(i);
        }
        // avancer
        i = i + 1;
        // invariant -> max = plus grand de v[0 .. i-1]
    }
    // i = v.size() et max = plus grand de v[0 .. v.size() - 1]
    return max;
}
```

Version récursive : le modèle

- Contrairement à la version itérative
  - une fonction « point d'entrée »
    - int maximumDPR(ArrayList<Integer> v)
    - qui cherche le maximum sur tout le vecteur v
  - une fonction récursive qui fait le travail (worker)
    - ```
      int maximumDPRWorker(ArrayList<Integer> v,
                           int borneInf,
                           int borneSup)
      ```
    - qui cherche l'indice sur une tranche `[borneInf .. borneSup]` du vecteur v
- En effet
  - la « recherche » a besoin de connaître la borne inférieure et la borne supérieur de l'intervalle de recherche qui va changer à chaque appel récursif
      ```

Version récursive : le modèle

- On aura donc le point d'entrée :

```
private static
int maximumDPR(ArrayList<Integer> v) {
// {v non vide} => {résultat = plus grande valeur de v}

  // on cherche initialement sur tout le vecteur
  // avec le worker sur l'intervalle [0 .. v.size()-1]
  return maximumDPRWorker(v, 0, v.size()-1);
}
```

Version récursive : le worker

- Rappels
  - précondition du worker
    - v[borneInf .. borneSup] non vide
  - recherche entre borneInf et borneSup
- Raisonnement
  - ➤ borneInf = borneSup (BASE, vecteur de taille 1)
    - résultat = v[borneInf]
  - ➤ borneInf < borneSup ⇨ (RÉCURRENCE)
    - m ← (inf+sup)/2
    - maxGauche = maximumDPRWorker(v, borneInf, m);
    - maxDroit = maximumDPRWorker(v, m+1, bornesup);
    - résultat = max(maxGauche, maxDroit)

Version récursive : code du worker

```java
private static int maximumDPRWorker(ArrayList<Integer> v,
                                    int borneInf,
                                    int borneSup) {
  if (borneInf == borneSup) {                  // BASE

    return v.get(borneInf);

  } else {                                     // RÉCURRENCE

    int m = (borneInf + borneSup) / 2; // point milieu

    // résoudre deux problèmes 2 fois plus petits
    // sur le demi-vecteur gauche
    int maxGauche = maximumDPRWorker(v, borneInf, m);
    // sur le demi-vecteur droit
    int maxDroit = maximumDPRWorker(v, m+1, borneSup);

    // fusionner : maximum du vecteur v est le plus grand
    // parmi maxGauche et maxDroit
    return Math.max(maxGauche, maxDroit);
  }
}
```

Procédure main de trace

```
public static void main(String[] args) {

  ArrayList<Integer> v = new ArrayList<>();

  for (int i = 1; i < 500; i = i + 1) {
    v.add((int) (Math.random() * 500));
  }

  System.out.println("Le vecteur : " + v);

  System.out.println("en itératif, le maximum de v est :  »
                    + maximumIter(v));
  System.out.println("en diviser pour régner, le maximum de v est : "
                    + maximumDPR(v));
}
```

Trace obtenue

Le vecteur : [336, 105, 254, 360, 2, 104, 419, 445, 301, 328, 46, 57, 191, 465, 340, 425, 478, 236, 491, 259, 75, 96, 386, 84, 66, 349, 237, 286, 2, 380, 164, 121, 310, 478, 297, 89, 458, 355, 57, 341, 38, 1, 336, 191, 189, 433, 402, 390, 335, 225, 472, 477, 28, 374, 19, 391, 253, 204, 25, 484, 384, 435, 68, 202, 112, 481, 272, 457, 5, 245, 384, 114, 27, 390, 414, 54, 222, 78, 20, 451, 458, 124, 48, 233, 34, 326, 392, 147, 193, 467, 108, 450, 492, 90, 20, 8, 284, 136, 314, 439, 482, 444, 492, 491, 110, 380, 335, 481, 131, 43, 408, 453, 337, 51, 484, 148, 179, 375, 169, 444, 189, 14, 360, 202, 286, 74, 133, 262, 71, 383, 37, 388, 375, 360, 246, 88, 182, 197, 455, 23, 220, 258, 402, 164, 422, 264, 353, 408, 146, 409, 431, 93, 493, 45, 167, 189, 284, 396, 181, 455, 218, 57, 471, 447, 5, 128, 350, 92, 259, 407, 135, 401, 229, 457, 314, 490, 486, 195, 28, 251, 473, 337, 309, 362, 441, 150, 201, 297, 95, 428, 4, 120, 396, 481, 111, 98, 345, 220, 51, 74, 190, 48, 80, 76, 426, 325, 402, 316, 107, 115, 298, 69, 424, 257, 242, 356, 224, 6, 318, 486, 74, 110, 56, 235, 244, 191, 102, 326, 452, 188, 450, 421, 481, 387, 304, 451, 324, 266, 479, 471, 279, 415, 112, 140, 493, 264, 358, 10, 278, 186, 84, 59, 106, 283, 66, 253, 46, 288, 40, 417, 34, 27, 456, 395, 251, 294, 213, 197, 31, 300, 136, 320, 150, 20, 449, 485, 26, 374, 486, 208, 300, 109, 421, 42, 12, 151, 326, 253, 26, 405, 405, 372, 139, 160, 352, 318, 310, 250, 315, 199, 213, 81, 97, 477, 377, 424, 304, 348, 471, 75, 288, 36, 387, 103, 385, 198, 79, 357, 108, 219, 270, 310, 116, 454, 463, 58, 492, 378, 360, 476, 60, 134, 193, 45, 194, 146, 112, 346, 313, 301, 260, 1, 16, 146, 305, 94, 61, 168, 218, 194, 205, 443, 276, 124, 210, 353, 292, 474, 123, 69, 329, 187, 300, 170, 166, 177, 467, 86, 115, 25, 491, 477, 300, 240, 398, 261, 380, 429, 87, 228, 248, 439, 91, 50, 123, 176, 148, 484, 67, 307, 330, 244, 319, 470, 375, 290, 49, 407, 398, 204, 486, 121, 102, 426, 107, 253, 142, 287, 12, 314, 309, 179, 297, 305, 213, 154, 272, 469, 243, 399, 64, 93, 104, 303, 188, 292, 324, 240, 444, 130, 255, 266, 24, 257, 150, 92, 52, 428, 75, 68, 317, 382, 45, 359, 341, 31, 406, 136, 484, 285, 221, 339, 216, 120, 44, 0, 253, 126, 224, 110, 235, 353, 75, 241, 216, 177, **497**, 425, 236, 259, 396, 254, 396, 386, 492, 280, 354, 84, 434, 43, 371, 403, 393, 395, 44, 160, 172, 379, 119, 205, 148, 24, 310, 461, 261, 264, 343, 32, 110]

en itératif, le maximum de v est : 497

en diviser pour régner, le maximum de v est : 497