

R3.02

Développement efficace

Cours 0 – la récursivité

Hervé Blanchon

Université Grenoble Alpes

IUT 2 – Département Informatique

Plan du cours

-  Objectif du module et organisation
-  Des fonctions récursives sur les entiers
 -  `int factorielle(int n)`
 -  `bool estPair(int n)`
-  Les tours de Hanoï
 -  un jeu ultra classique !
-  Approches possibles pour la résolution récursive d'un pb.
 -  diminuer pour régner
 -  recherche dichotomique
 -  diviser pour régner
 -  recherche du maximum

Où j'apprends où je vais & comment !

OBJECTIFS & ORGANISATION

Objectifs

✓ Un module d'algorithmique ...

 *savoir utiliser quelques structures de données avancées, en implanter certaines, et savoir implanter des algorithmes qui les manipulent (PPN)*

✗ ... pas de conception

 c'était l'objectif du R3.04 (qualité de développement)

Donc ...

 on vous fournira des classes spécifiées
 vous devrez implanter et tester des méthodes

Plus précisément ...

Notion de Type de Données Abstrait

↳ liste, ensemble, pile, file, dictionnaire

Liste chaînée

↳ des algorithmes de traitement récursifs et itératifs
pour implanter : liste, ensemble, pile, file

Arbre binaire ordonné (... de recherche)

↳ des algorithmes de traitement récursifs et itératifs
pour implanter : ensemble

Arbre n-aire

↳ des algorithmes itératifs
pour implanter : données géographiques

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

$$0! =$$

$$5! =$$

$$10! =$$

3 millions

$$15! = 1\ 307\ 674\ 368\ 000$$

1 million de millions

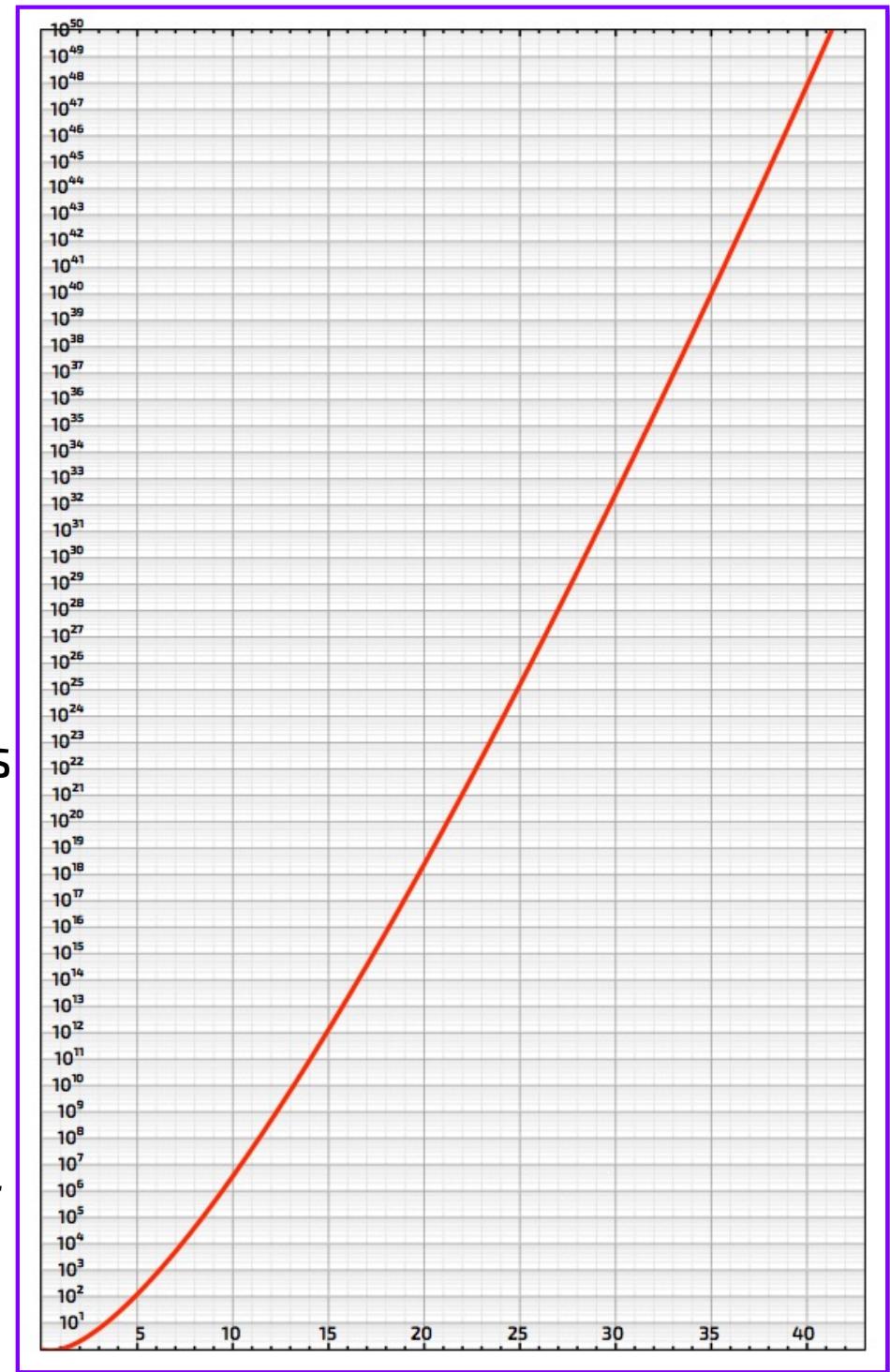
$$20! = 2\ 432\ 902\ 008\ 176\ 640\ 000$$

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

$$\begin{array}{rcl} 1 & \\ 120 & \\ 3\ 628\ 800 & \end{array}$$



Un algorithme itératif

■ Un raisonnement par récurrence

Hypothèse

$$r = 1 \times 2 \times \dots \times (i-1)$$

Cas possibles

- $i = n + 1 \Rightarrow *$ {rendre $r;$ } [$r = 1 \times 2 \times \dots \times (n-1) \times n$]
- $i \leq n \Rightarrow r := r \times i; i := i + 1; \rightarrow H$

Itération

tant que $i \leq n$ faire

Initialisation

$i := 1;$
 $r := 1; \rightarrow H [r = 0!]$

L'implantation itérative en C++

```
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 :

❖ $\text{factorielle}(n) = \text{factorielle}(n-1) \times n$

■ Mise en pratique, une trace du calcul

❖ $\text{factorielle}(5) = \text{factorielle}(4) \times 5$

avec $\text{factorielle}(4) = \text{factorielle}(3) \times 4$

avec $\text{factorielle}(3) = \text{factorielle}(2) \times 3$

avec $\text{factorielle}(2) = \text{factorielle}(1) \times 2$

avec $\text{factorielle}(1) = \text{factorielle}(0) \times 1$

avec $\text{factorielle}(0) = 1$

❖ **Ouf ! un cas d'arrêt des appels récursifs (calcul trivial) !**

soit $\text{factorielle}(1) = 1 \times 1 = 1$

soit $\text{factorielle}(2) = 1 \times 2 = 2$

soit $\text{factorielle}(3) = 2 \times 3 = 6$

soit $\text{factorielle}(4) = 6 \times 4 = 24$

Soit $\text{factorielle}(5) = 24 \times 5 = 120$

Descente

Remontée



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 :
 - ↳ $\text{factorielle}(0) = 1$ LA BASE
 - ↳ $\text{factorielle}(n) = \text{factorielle}(n-1) \times n$ LA RÉCURRENCE

- On peut formaliser comme suit :
 - $n = 0 \Rightarrow * \{ \text{résultat} = 1 \}$ (BASE)
 - $n > 0 \Rightarrow * \{ \text{résultat} = \text{factorielle}(n-1) \times n \}$ (RÉCURRENCE)

Vers une formalisation

La formalisation :

- $n = 0 \Rightarrow * \{r\acute{e}sultat = 1\}$ (BASE)
- $n > 0 \Rightarrow * \{r\acute{e}sultat = factorielle(n-1) \times n\}$ (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 C++

```
int factorielleRec(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        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) **d_t** pour laquelle (resp. lesquelles) on connaît trivialement la réponse au problème (**BASE**)
 - ❖ il faut s'assurer que les réductions successives des données (passage de **D** à **d**) convergent vers les données **d_t**

Derniers mots...

- Réduire la taille des données **D** vers **d** ?
- Quelques pistes...
 - **D** est une valeur numérique
 - **d** est une valeur numérique qui se rapproche d'une valeur **d_t** 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 **d_t** 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 **d_t** pour laquelle on connaît la réponse au problème **P**
 - **D** est un arbre (on verra plus tard)
 - **d** est un arbre plus petit qui se rapproche d'un arbre **d_t** pour lequel on connaît la réponse au problème **P**

Un grand classique ! et un rappel de R1.01

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

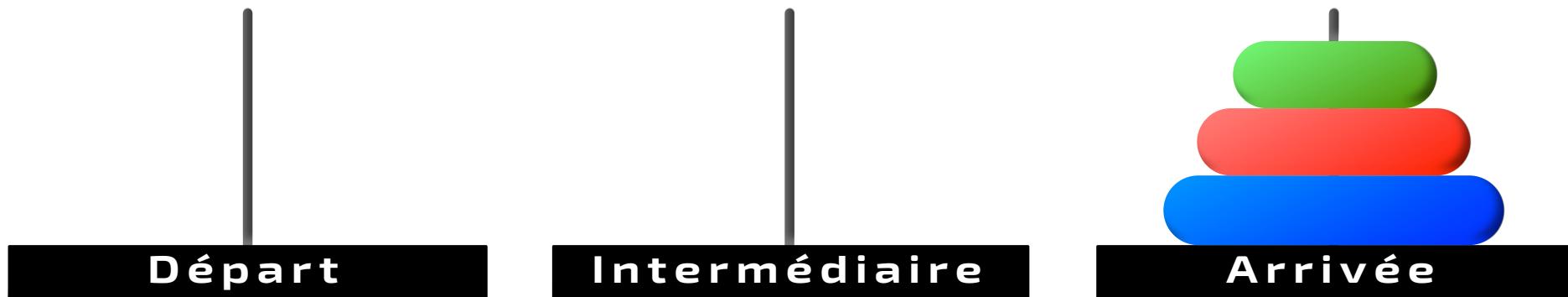
Source : Wikipédia consulté le 20/10/2015

Un exemple avec 3 disques

Situation initiale

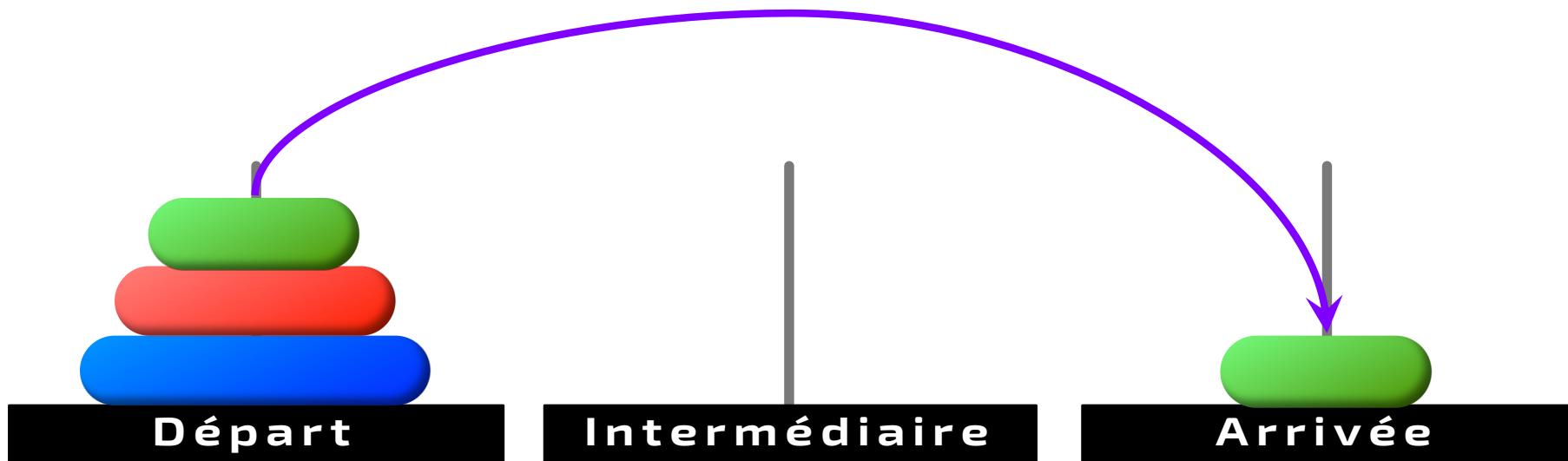


Situation finale



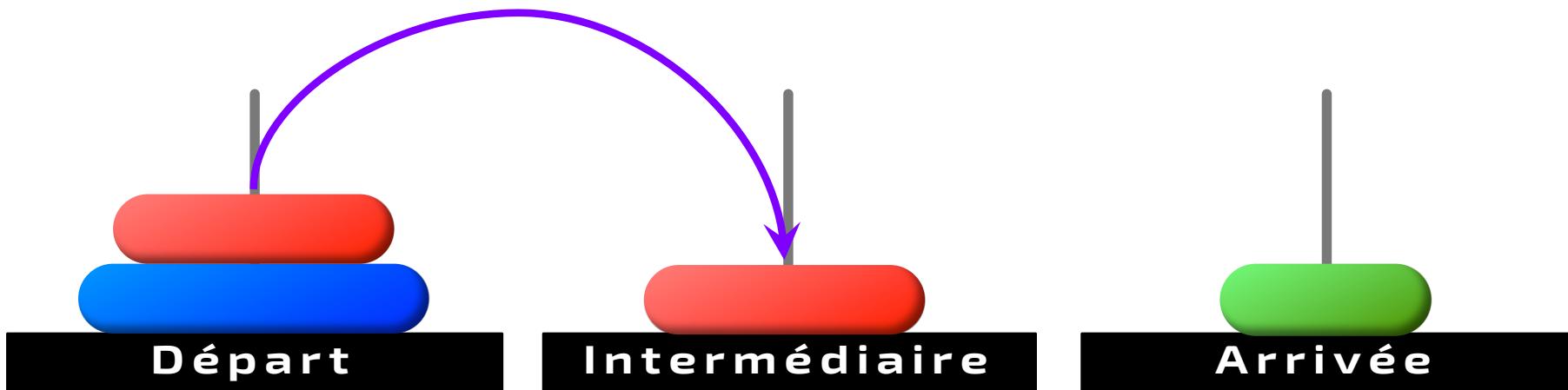
Les étapes de la résolution

Étape 1



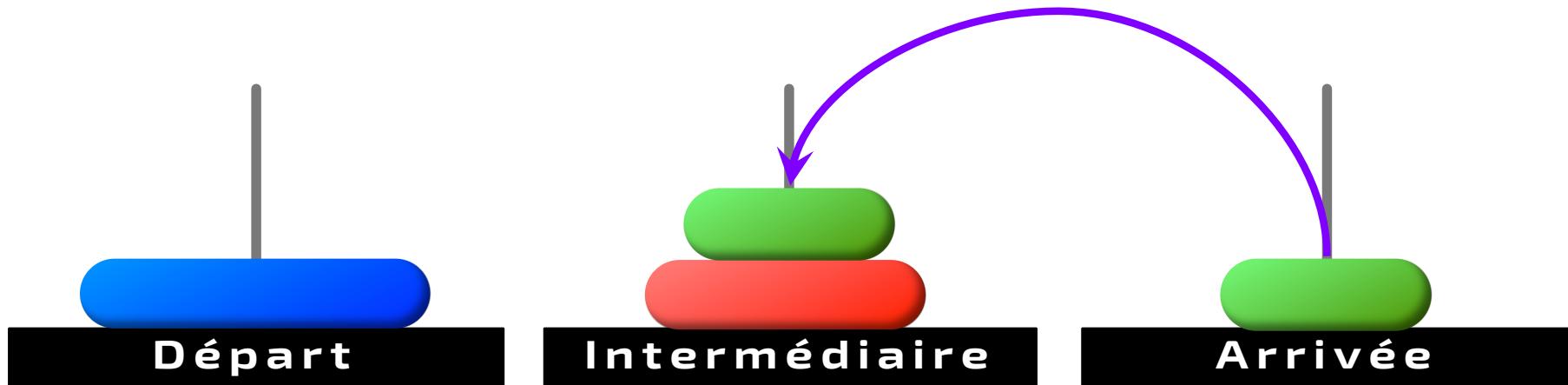
Les étapes de la résolution

Étape 2



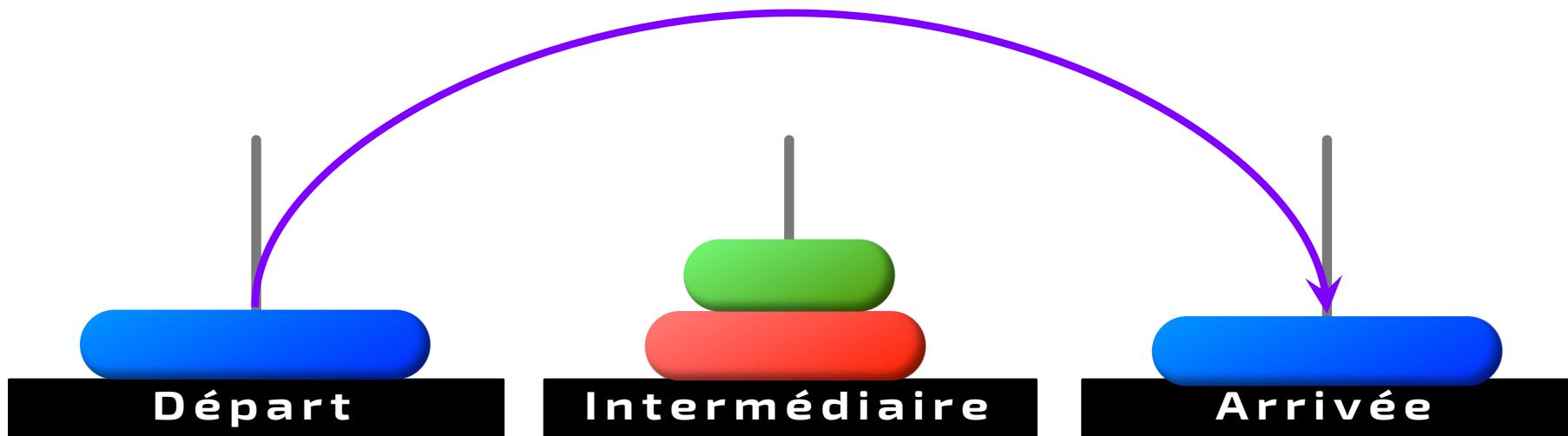
Les étapes de la résolution

Étape 3



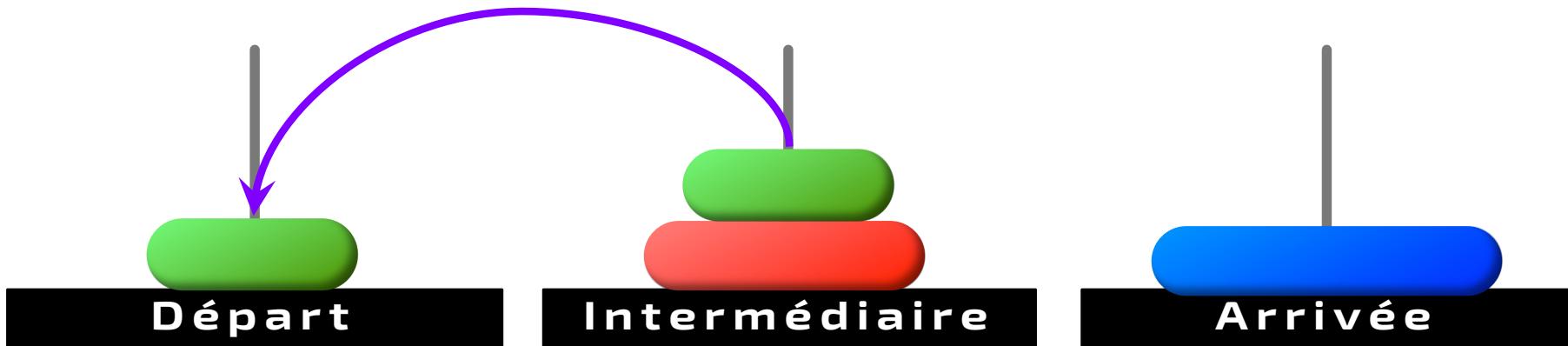
Les étapes de la résolution

■ Étape 4



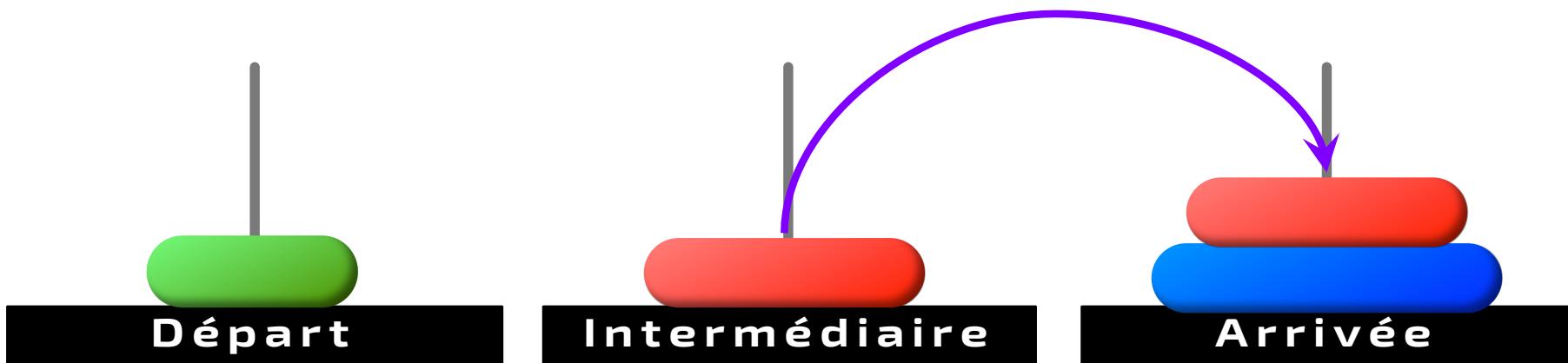
Les étapes de la résolution

Étape 5



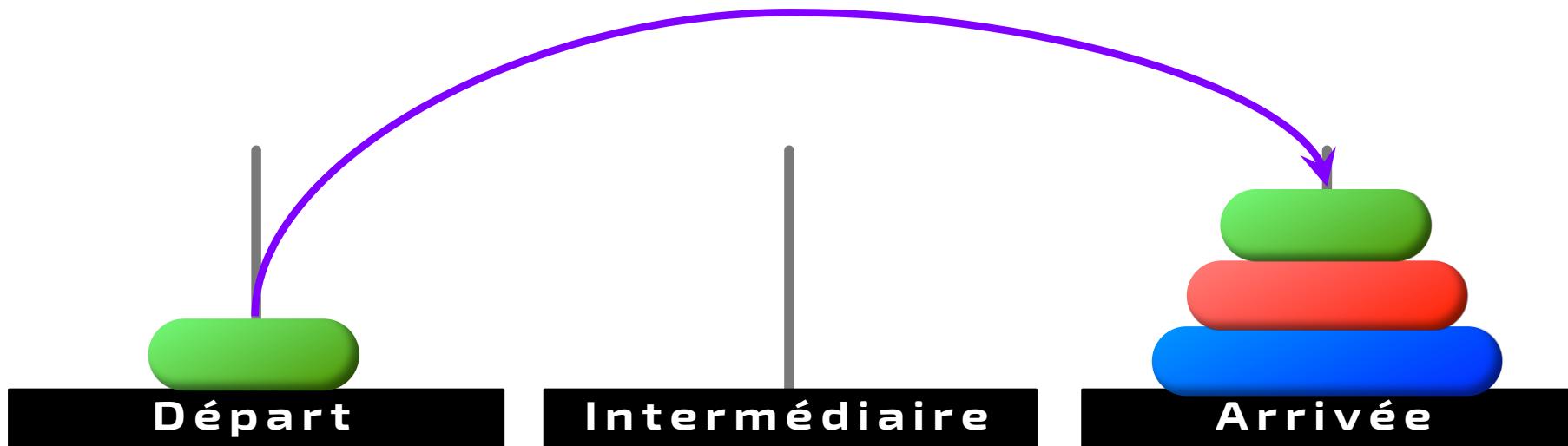
Les étapes de la résolution

Étape 6



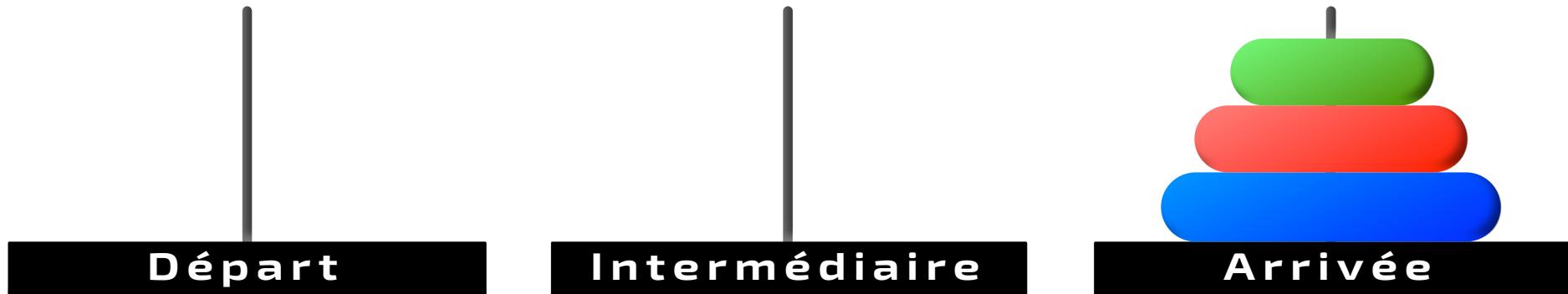
Les étapes de la résolution

Étape 7



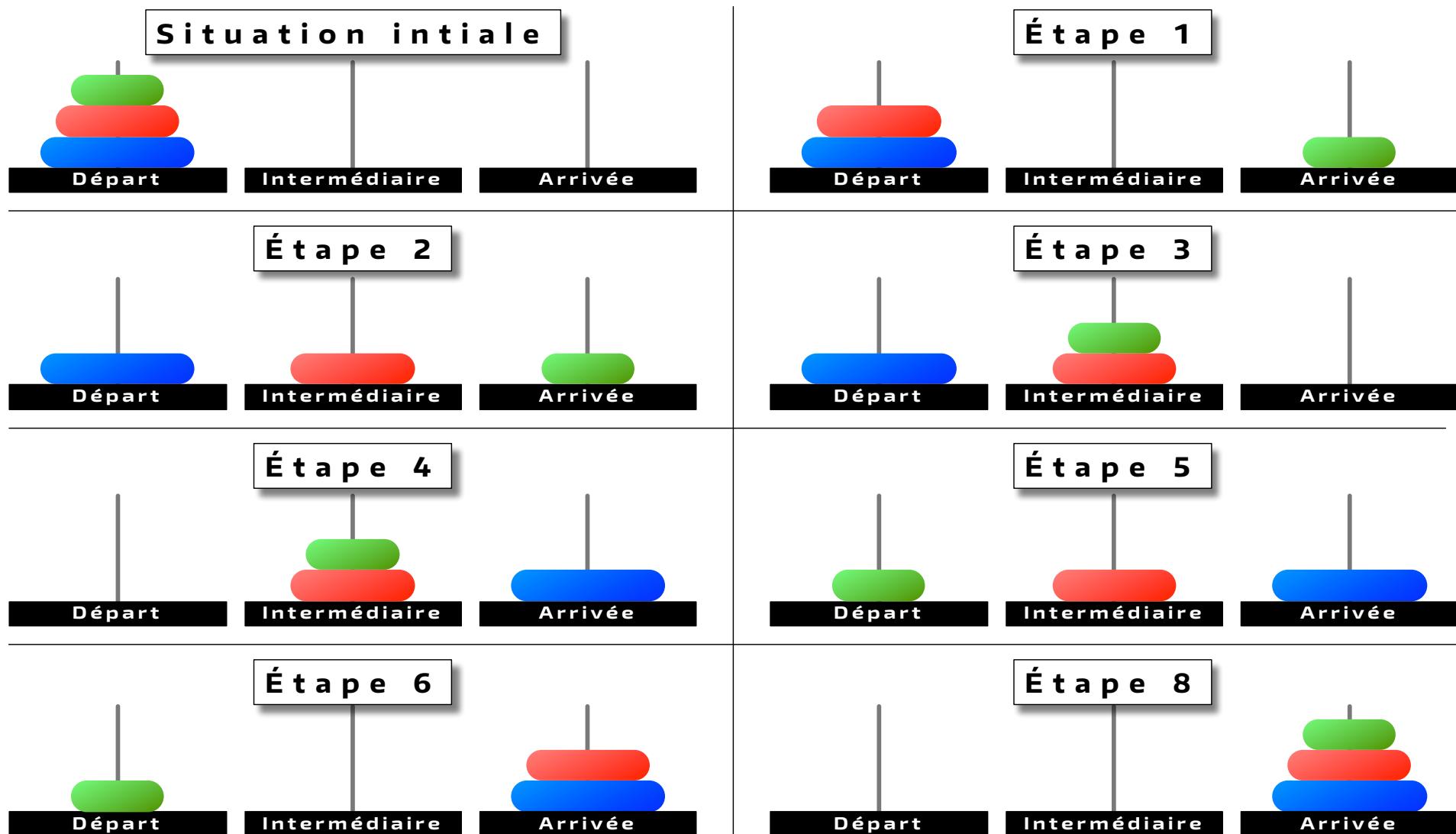
Les étapes de la résolution

 Situation finale



Les étapes de la résolution (récapitulatif)

■ Avec 3 disques on a les 7 étapes suivantes :



La procédure à réaliser

Objectif

Écrire la procédure

-  `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 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

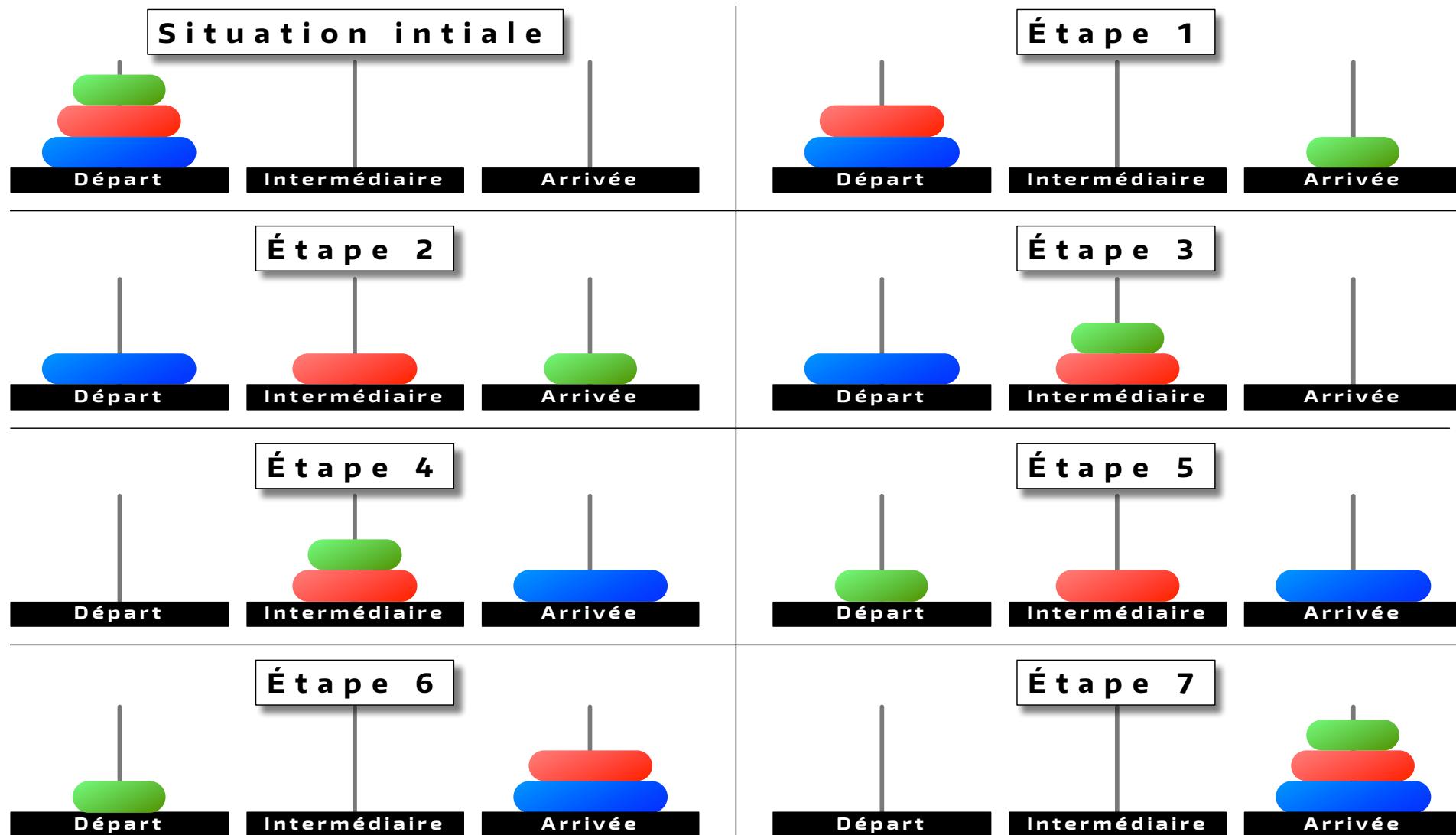
Déplacement disque 1 du piquet D vers le piquet A	-- étape 1
Déplacement disque 2 du piquet D vers le piquet I	-- étape 2
Déplacement disque 1 du piquet A vers le piquet I	-- étape 3
Déplacement disque 3 du piquet D vers le piquet A	-- étape 4
Déplacement disque 1 du piquet I vers le piquet D	-- étape 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**

- Connait-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 d_1 & d_2 (telles que $d_1 \cup d_2 = D$ et $d_1 \cap d_2 = \emptyset$) et utiliser les résultats obtenus pour obtenir le résultat sur D (on coupe habituellement en 2 mais pas toujours !)
 - ❖ on va voir un exemple...

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

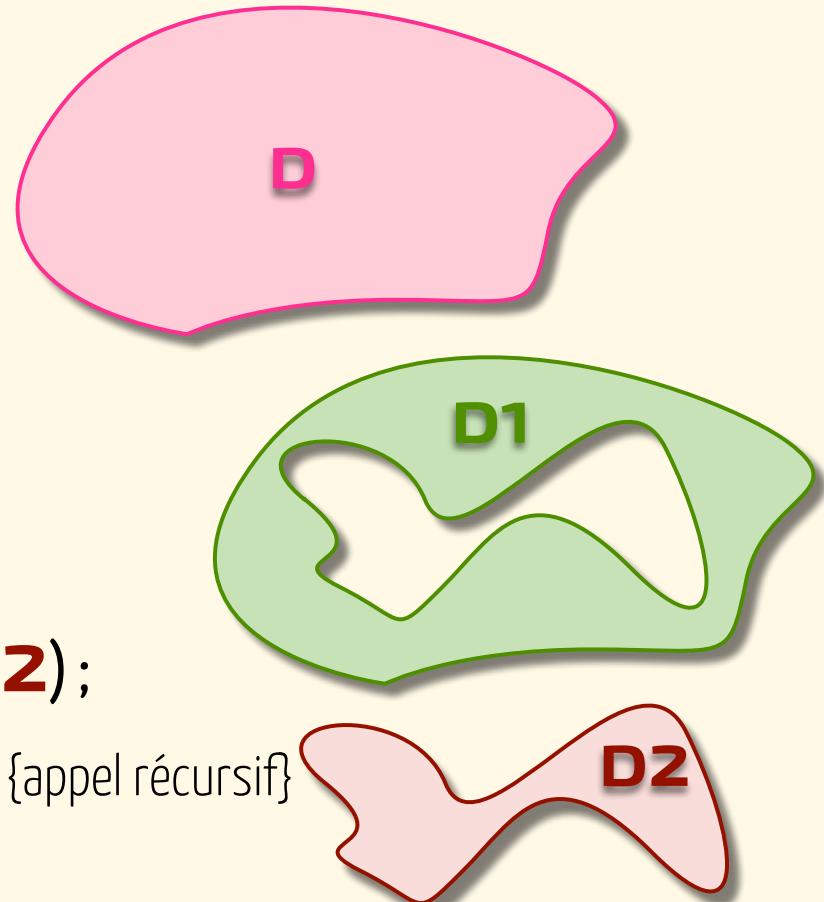
(D1, D2) ← Diviser(D);

Dx ← Choisir_sous_problème(D1, D2);

Résultat ← Résoudre_le_problème(Dx); {appel récursif}

finsi;

fin;



Exemples

- Recherche dichotomique de val dans v vecteur
 - ❖ Diviser coupe **v** en deux en fournissant l'indice **m**
 - ❖ Choisir un sous-problème (moitié inférieure ou moitié supérieure) en comparant **v[m]** à **val**
 - ❖ Résoudre le sous-problème choisi : recherche dichotomique sur une moitié de vecteur

- Recherche de val dans un arbre binaire de recherche (cf. cours 2)

Fonction récursive sur les vecteurs

LA RECHERCHE DICHOTOMIQUE

Le problème

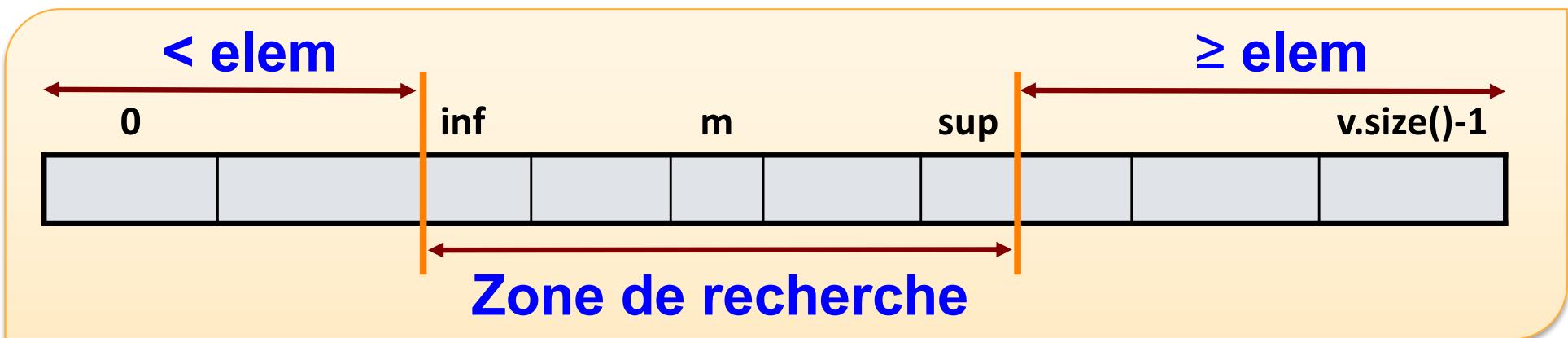
- Écrire la fonction « point d'entrée » suivante :
 - ❖ **int dichoIter(vector<T> v ; T val)**
 - ❖ si $val \in v$, retourne la position de la première occurrence de val dans v (positions numérotées à partir de 0)
 - ❖ si $val \notin v$, retourne l'opposé de la position que devrait occuper val dans v
 - ❖ **Note :** cf. fct. `lower_bound(...)` de `<algorithm>`
 - Il s'agit donc de trouver un indice m tel que :



- ❖ si $v(m) = val$ alors retourner m
- ❖ si $v(m) > val$ alors retourner $-m$

Mode opératoire (cf. cours 6 pt. 3 R1.01)

Un dessin pour l'invariant d'itération



Mode opératoire (cf. cours 6 pt. 3 R1.01)

- Choisir un indice au milieu de l'intervalle des indices du segment considéré de V ([inf...sup])
 - ❖ Si $V(m) < val$ alors poursuivre la recherche dans l'intervalle $[m+1...sup]$
 - ❖ Si $V(m) \geq val$ alors poursuivre la recherche dans l'intervalle $[inf...m-1]$
- La recherche s'arrête lorsque l'intervalle de recherche devient vide, soit lorsque $inf=sup+1$
 - ❖ si $V(inf) = val$ alors retourner inf
 - ❖ si $V(inf) > val$ alors retourner $-inf$
- **NOTE**
 - ❖ À l'initialisation prévenir le débordement de inf à droite

Version itérative : vers le code

 Écrire la fonction suivante

 `int function dichoIter(vector<T> v ; T val)`

 `//{v trié croissant} => {`

- `- si $val \in v$: résultat = position la plus à gauche occupée par val , on a $résultat \in [1..v.size()]$;`
- `- si $val \notin v$: résultat = opposé de la position que devrait occuper val , on a $résultat \in [-(v.size())+1]..-1]$ }`

Version itérative : le code

```
template<class T>
int dichoIter(vector<T> v, T val) {
    int sup = v.size()-1;           // borne sup des indices
    if (v[sup] < val) {           // val > v[v.size()-1]
        return -(sup+1);
    } else {
        int m;                   // indice milieu
        int inf = 0;              // borne inf des indices
        while (inf < sup) {
            m = (inf + sup) / 2; // point milieu
            if (v[m] >= val) {
                sup = m;          // chercher à gauche v[inf..m-1]
            } else {
                inf = m + 1;       // chercher à droite v[m+1..sup]
            }
        }
        if (v[inf] == val) {
            return inf;           // trouvé
        } else {
            return -inf;          // absent
        }
    }
}
```

Version récursive : le modèle

- Contrairement à la version itérative
 - ❖ une fonction « point d'entrée »
 - ❖ `int dichoRec(vector<T> v, T val)`
 - ❖ une fonction récursive qui fait le travail (worker)
 - ❖ `int dichoRecWorker(vector<T> v, int inf, int sup, T val)`
- En effet
 - ❖ la « recherche » a besoin de connaître la **borne inf (inf)** et la **borne sup (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

```
template<class T>
int dichoRec(vector<T> v , T val) {
    if (v[v.size()-1] < val) {
        // inutile de chercher (plus grand de v < val)
        return -v.size();
    } else {
        // on cherche initialement sur tout le vecteur
        return dichoRecWorker(v, 0, v.size() - 1, val);
    }
}
```

Version récursive : le worker

Rappels

précondition du worker

 position de val dans $v \leq v.size()$ (présente ou absente)

recherche entre inf et sup

Raisonnement

➤ $\text{inf} = \text{sup} \Rightarrow (\text{BASE}, \text{recherche dans intervalle vide})$

➤➤ $v[\text{inf}] = \text{val} \Rightarrow * \{\text{résultat} = \text{inf}\}$

➤➤ $v[\text{inf}] \neq \text{val} \Rightarrow * \{\text{résultat} = -\text{inf}\}$

➤ $\text{inf} < \text{sup} \Rightarrow (\text{RÉCURRENCE})$

$m \leftarrow (\text{inf}+\text{sup})/2$

➤➤ $v[m] < \text{val} \Rightarrow \text{résultat} = \text{dichoRecWorker}(v, m+1, \text{sup}, \text{val})$

➤➤ $V[m] \geq \text{val} \Rightarrow \text{résultat} = \text{dichoRecWorker}(v, \text{inf}, m-1, \text{val})$

Version récursive : code du worker

```
template<class T>
int dichoRecWorker(vector<T> v, int inf, int sup, T val) {
    if (inf == sup) {                                // BASE
        if (v[inf] == val) {
            return inf;                            // trouvé
        } else {
            return -inf;                           // absent
        }
    } else {                                         // RÉCURRENCE
        int m = (inf + sup) / 2; // point milieu
        if (v[m] >= val) {
            return dichoRecWorker(v, inf, m, val);
                // poursuivre la recherche à gauche
        } else {
            return dichoRecWorker(v, m+1, sup, val);
                // chercher dans la moitié supérieure
        }
    }
}
```

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) ;

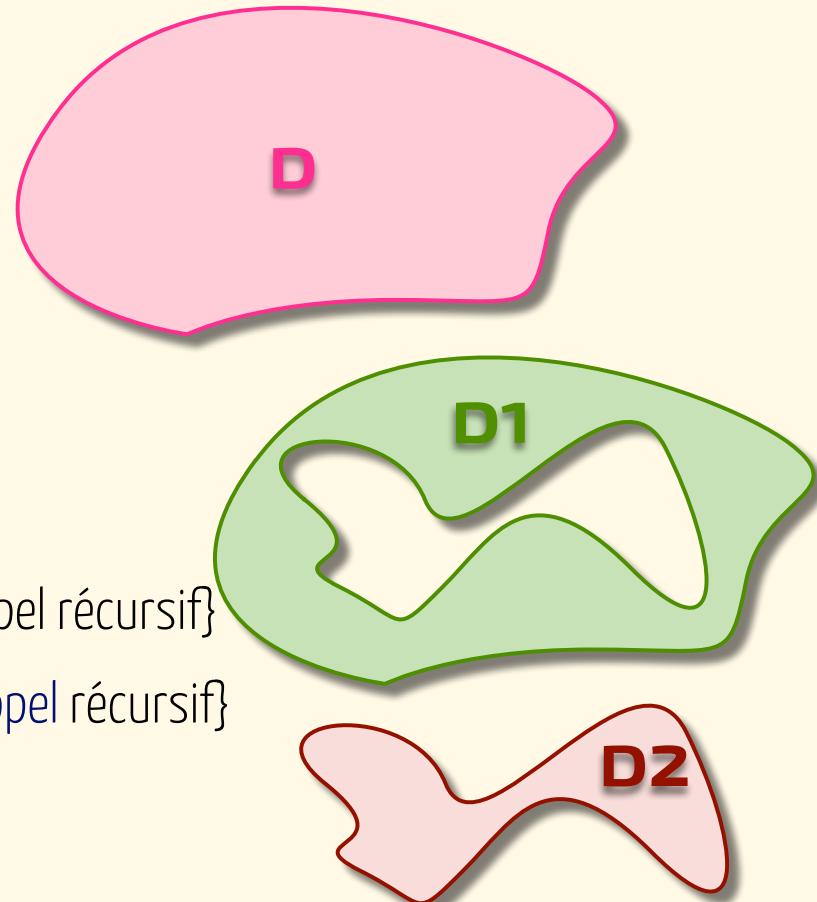
R1 ← Résoudre_le_problème(D1) ; {appel récursif}

R2 ← Résoudre_le_problème(D2) ; {appel récursif}

Résultat ← Fusionner(R1, R2) ;

finsi ;

fin ;



Fonction récursive sur les vecteurs

RECHERCHE DU MAXIMUM

Le problème

- On considère la précondition suivante
 - ❖ // {v non vide}
- Version itérative
 - ❖ sur le vecteur complet cf. R1.01 cours 5
 - ❖ sur un intervalle cf. R1.01 cours 7
- Version récursive en diviser pour régner
 - ❖ une fonction « point d'entrée » suivante :
 - ❖ `T maxRec(const vector<T>& v)`

Version récursive : le modèle

■ On aura donc

```
template<class T>
T maxVectRect(const vector<T>& v) {
    if (v.size()==1) {
        // vecteur de taille 1
        return v[0];
    } else {
        int inf = 0;
        int sup = (int) v.size() - 1;
        return maxVectWorker(v, inf, sup);
    }
}
```

Réflexion sur le worker

- T maxVect(vector<T>& v, int inf, int sup)
 - inf = sup \Rightarrow * {résultat = v[inf]} (BASE)
 - inf < sup \Rightarrow (RÉCURRENCE)
 - [diviser en 2 : m ; résoudre 2 sous-pbs : maxVect ; fusionner : max2]
 - $m = (inf + sup) / 2$
 - résultat = max2(maxVect(v, inf, m),
maxVect(v, m+1, sup));
- ❖ Commentaires
 - ❖ le max d'un vecteur à un élément (inf=sup) est cet élément
 - ❖ le max d'un autre vecteur (inf<sup) est le plus grand des max des demi vecteurs inférieur ([inf..m]) et supérieur ([m+1..sup])

Version récursive : code du worker

```
template<class T>
T maxVectWorker(const vector<T>& v, int inf, int sup) {
    if (inf == sup) { // BASE SUR UN VECTEUR NON VIDE
        return v[inf];
    } else {           // RÉCURRENCE
        int m = (inf + sup) / 2;
        return max(maxVectWorker(v, inf, m),
                   maxVectWorker(v, m + 1, sup));
    }
}
```