

## Outils mathématiques pour l'informatique

CÉLINE ROBARDET

<http://liris.cnrs.fr/celine.robardet/>

Institut National des Sciences Appliquées de Lyon

## Plan

- 1 Introduction
  - Motivation du cours
  - Quelques exemples de problèmes.
- 2 Récursivité et induction

## Pourquoi un cours de mathématiques discrètes ?

- L'informatique ne peut se réduire à la programmation.
- Elle ne peut pas plus se limiter au génie logiciel.

Edsger Dijkstra :

*L'informatique n'est pas plus la science des ordinateurs  
que l'astronomie n'est celle des télescopes*

## Pourquoi un cours de mathématiques discrètes ?

- L'informatique ne peut se réduire à la programmation.
- Elle ne peut pas plus se limiter au génie logiciel.

Edsger Dijkstra :

*L'informatique n'est pas plus la science des ordinateurs  
que l'astronomie n'est celle des télescopes*

**L'informatique est la résolution de problèmes.**

## Pourquoi un cours de mathématiques discrètes ?

- Les mathématiques sont au cœur de la résolution de problèmes.
- Définir le problème nécessite bien souvent la rigueur des mathématiques.
- Le bon usage et l'analyse des modèles, des structures de données et des algorithmes nécessitent une base solide en mathématiques.
- Justifier pourquoi une façon particulière de résoudre le problème est correct ou efficace nécessite une analyse basée sur un modèle mathématique bien défini.

## Pourquoi un cours de mathématiques discrètes ?

- Abstraire le problème est nécessaire afin de pouvoir réutiliser des connaissances.
- Il est rare de rencontrer un problème dans un cadre abstrait (votre patron ne va pas vous demander de trouver un arbre couvrant de poids minimum). Cela va être plutôt votre tâche de modéliser le problème pour le ramener à une solution connue.

## Scénario 1

Une compagnie de taxis vous a embauché pour développer un programme informatique pour automatiser les tâches suivantes.

**Tâche 1** - Dans le premier scénario, les taxis et leur chauffeur sont réservés pour une période de temps déterminée (défini par une date de début et de fin) et sont facturés de manière forfaitaire. Le programme doit être capable de générer un calendrier afin que le nombre de clients pouvant être pris en charge soit maximum.

## Scénario 2

**Tâche 2** - Dans le deuxième scénario, la compagnie de taxis envisage de permettre aux clients de faire une offre à un chauffeur (de sorte que le plus offrant obtient un taxi lorsqu'il n'y a pas assez de taxis disponibles pour tout le monde). Le programme devrait ainsi fournir un calendrier réalisable (i.e. un client au plus par taxi), tandis que dans le même temps, le profit doit être maximisé en choisissant les offres les plus élevées.

## Scénario 3

**Tâche 3** - Dans un troisième scénario, le client est autorisé à spécifier un ensemble de périodes et à faire une offre pour cet ensemble de périodes. Le chauffeur peut quant à lui choisir d'accepter ou de rejeter l'ensemble des périodes. Le calendrier d'affectation doit toujours maximiser le profit.

## Scénario

### Quelle est votre solution ?

- Comment pouvez-vous modéliser de tels scénarios ?
- Quels algorithmes vont vous permettre de résoudre ces scénarios ?
- Comment justifier leur bon fonctionnement ? Permettent-ils de garantir que l'on obtient le profit maximal (solution optimale garantie) ?

## Scénario

- Les fondamentaux étudiés dans ce cours sont la base que vous pourrez utiliser pour résoudre ce type de problèmes.
- Le premier scénario peut être résolu efficacement par un algorithme glouton.
- Le deuxième scénario peut également être résolu efficacement, mais par une technique qui requiert un peu plus de ressources (RAM) : la programmation dynamique.
- Le dernier scénario ne peut pas être résolu efficacement (le problème est NP-complet) par aucune technique connue à ce jour. Pour ce problème, pour garantir l'obtention d'une solution optimale, on doit considérer toutes les possibilités (exponentielle en fonction du nombre de demandes de réservations).

## Plan

- 1 Introduction
- 2 Récursivité et induction
  - Récursivité et algorithme
  - Récursivité et mathématiques
  - Diviser pour régner

## Définition récursive et algorithme récursif

Une définition récursive est une définition dans laquelle intervient ce que l'on veut définir. Un algorithme est dit récursif lorsqu'il est défini en fonction de lui-même.

### Récursivité simple

La fonction  $x \rightarrow x^n$  peut être définie récursivement :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n \geq 1 \end{cases}$$

**Fonction** Puissance( $x, n$ : entier) : entier

**Début**

**Si**  $n = 0$  **alors** Retourner 1

**Sinon** Retourner  $x$ . Puissance( $x, n - 1$ )

**Fin**

## Récursivité multiple

Une définition récursive peut contenir plus d'un appel récursif.

### Exemple

Nous voulons calculer ici les combinaisons  $C_n^p$  en se servant de la relation de Pascal :

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon} \end{cases}$$

**Fonction** Combinaison( $n, p$ : entier) : entier

**Début**

**Si**  $p = 0$  ou  $p = n$  **alors**  $r \leftarrow 1$

**Sinon**  $r \leftarrow$  Combinaison( $n - 1, p$ ) + combinaison( $n - 1, p - 1$ )

**Retourner**  $r$

**Fin**

## Récursivité mutuelle

Des définitions sont dites mutuellement récursives si elles dépendent les unes des autres.

### Exemple

On peut définir la parité par :

$$\text{pair}(n) = \begin{cases} \text{vrai} & \text{si } n = 0 \\ \text{impair}(n - 1) & \text{sinon} \end{cases} \quad \text{impair}(n) = \begin{cases} \text{faux} & \text{si } n = 0 \\ \text{pair}(n - 1) & \text{sinon} \end{cases}$$

**Fonction** Pair( $n$ : entier) : booléen

**Début**

**Si**  $n = 0$  **alors** Retourner vrai

**Sinon** Retourner Impair( $n - 1$ )

**Fin**

**Fonction** Impair( $n$ : entier) : booléen

**Début**

**Si**  $n = 0$  **alors** Retourner faux

**Sinon** Retourner Pair( $n - 1$ )

**Fin**

## Récursivité imbriquée

### Exemple

La fonction d'Ackermann est définie comme suit :

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{sinon} \end{cases}$$

**Fonction** Ackermann( $m, n$ : entier) : entier

**Début**

**Si**  $m = 0$  **alors** Retourner  $n + 1$

**Sinon**

**Si**  $n = 0$  **alors** Retourner Ackermann( $m - 1, 1$ )

**Sinon** Retourner Ackermann( $m - 1, A(m, n - 1)$ )

**Fin Si**

**Fin**

## Principe et dangers de la récursivité - 1

## Principe

- Un certain nombre de cas dont la résolution est connue, ces “cas simples” formeront les cas d’arrêt de la récursion
- Un moyen de se ramener d’un cas “compliqué” à un cas “plus simple”.

La récursivité permet d’écrire des algorithmes concis et élégants.

## Difficultés

- La définition peut être dénuée de sens :  
Algorithme  $A(n)$   
retourner  $A(n)$
- Il faut s’assurer que toute exécution mènera à un cas d’arrêt
- Il nous faut s’assurer que la fonction est définie sur tout son domaine d’application.

## Principe et dangers de la récursivité - 2

## Exemple 1

L’algorithme ci-dessous teste si  $a$  est un diviseur de  $b$ .

**Fonction**  $\text{Diviseur}(a, b: \text{entier}) : \text{booléen}$

**Début**

**Si**  $a \leq 0$  **alors** **Retourner** Erreur

**Sinon**

**Si**  $a \geq b$  **alors** **Retourner**  $a = b$

**Sinon** **Retourner**  $\text{Diviseur}(a, b - a)$

**Fin Si**

**Fin**

## Principe et dangers de la récursivité - 3

## Exemple 2

**Fonction**  $\text{Syracuse}(n : \text{entier}) : \text{entier}$

**Début**

**Si**  $n = 0$  ou  $n = 1$  **alors** **Retourner** 1

**Sinon**

**Si**  $n \bmod 2 = 0$  **alors** **Retourner**  $\text{Syracuse}(n/2)$

**Sinon** **Retourner**  $\text{Syracuse}(3 \times n + 1)$

**Fin Si**

**Fin**

## Non décidabilité de la terminaison

## Question :

Peut-on écrire un programme qui vérifie automatiquement si un programme donné  $P$  termine quand il est exécuté sur un jeu de données  $D$  ?

**Entrée** : Un programme  $P$  et un jeu de données  $D$ .

**Sortie** : vrai si le programme  $P$  termine sur le jeu de données  $D$ , et faux sinon.

## Notions d'algorithme correct

Un algorithme est dit totalement correct si pour tout jeu de données il s'arrête et rend le résultat attendu

Un algorithme incorrect peut :

- Ne pas s'arrêter (on dit qu'il boucle)
- S'arrêter en rendant un mauvais résultat
- S'il termine, fournir le bon résultat, mais la terminaison n'est pas garantie (partiellement correct)

Pour certains problèmes difficiles, on peut se contenter d'un algorithme incorrect si l'on sait contrôler son taux d'erreur

## Importance de l'ordre des appels récursifs

Fonction qui affiche les entiers par ordre décroissant

**Procédure** Décroissant( $n$  : entier)

**Début**

**Si**  $n <> 0$  **alors**

afficher  $n$

Décroissant( $n - 1$ )

**Fin Si**

**Fin**

**Procédure** Croissant( $n$  : entier)

**Début**

**Si**  $n <> 0$  **alors**

Croissant( $n - 1$ )

afficher  $n$

**Fin Si**

**Fin**

## Exemple d'algorithme récursif : les tours de Hanoi

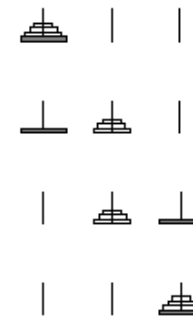
### Le problème

On dispose de trois tiges et de  $n$  disques de diamètres tous différents. Les règles du jeu sont :

- on ne peut déplacer qu'un seul disque à la fois,
- il est interdit de poser un disque sur un disque plus petit.

Le but est de déplacer tous les disques d'une tige à l'autre.

## Exemple d'algorithme récursif : les tours de Hanoi - 1



## Dérécursivation d'une récursion terminale

Dérécursiver, c'est transformer un algorithme récursif en un algorithme équivalent ne contenant pas d'appels récursifs.

## Définition : Récursivité terminale

Un algorithme est dit récursif terminal s'il ne contient aucun traitement après un appel récursif.

Procédure  $X(U)$ 

Début

Si  $C(U)$  alors $D$  $X(\alpha(U))$ Sinon  $T$ 

Fin

Procédure  $Y(U)$ 

Début

Tant que  $C(U)$  faire $D$  $U \leftarrow \alpha(U)$ 

Fin Tant que

 $T$ 

Fin

## Dérécursivation d'une récursion non terminale

Procédure  $X(U)$ 

Début

Si  $C(U)$  alors $D(U)$  $X(\alpha(U))$  $F(U)$ Sinon  $T(U)$ 

Fin

Procédure  $Y(U)$ 

Début

empiler(nouvel\_appel,  $U$ )

Tant que pile non vide faire

dépiler(état,  $V$ ) $U \leftarrow V$ 

Si état = nouvel\_appel alors

Si  $C(U)$  alors $D(U)$ ; empiler(fin,  $U$ )empiler(nouvel\_appel,  $\alpha(U)$ )Sinon  $T(U)$ 

Fin Si

Si état = fin alors

 $F(U)$ 

Fin Si

Fin Tant que

Fin

## Preuves par récurrence

## Théorème 1 :

Soit  $P(n)$  un prédicat qui dépend de l'entier  $n$ . Supposons les deux propriétés suivantes satisfaites

(B)  $P(n_0)$  est vrai (base de l'induction)(I)  $\forall n \geq n_0, [P(n) \Rightarrow P(n+1)]$  (induction)alors  $P(n)$  est vrai  $\forall n \geq n_0$ 

## Exemples :

- $S(n) : \sum_{i=0}^n 2^i = 2^{n+1} - 1, n \geq 0$

- Théorème : les nombres harmoniques  $H_k, k \in \mathbb{N}^*$  définis par  $H_k = \sum_{\ell=1}^k \ell^{-1}$  vérifient  $H_{2n} \geq 1 + n/2$  pour tout  $n \in \mathbb{N}$

## Récurrence complète

## Théorème 2 :

Soit  $P(n)$  un prédicat qui dépend d'un entier  $n$ , et soit  $n_0$  un entier positif ou nul. Si les deux conditions suivantes sont vérifiées

(B)  $P(n_0)$  est vrai(I)  $\forall n \geq n_0, [(\forall k, n_0 \leq k \leq n, P(k)) \Rightarrow P(n+1)]$ alors  $P(n)$  est vrai  $\forall n \geq n_0$ .

## Exemple :

- Pour tout entier  $n \geq 3$ , la somme  $S_n$  des angles intérieurs de tout polygone à  $n$  arêtes vaut  $(n-2)\pi$ .
- Toute somme à payer supérieure à 7 peut être décomposée en pièces de 3 et de 5.

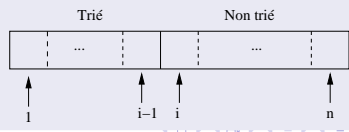
## Prouver les propriétés des programmes

### Invariant de boucle

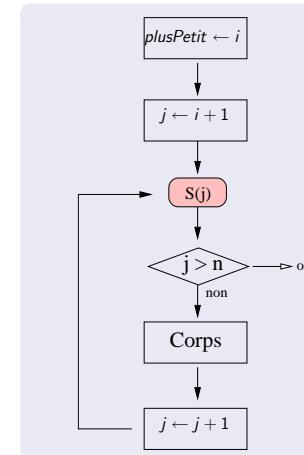
Un invariant de boucle est une propriété qui reste vraie à chaque passage dans la boucle (en un point qui est précisé). On va donc :

- Définir un invariant qui assure qu'à la fin la propriété voulue est satisfaite
- S'assurer qu'il est vrai au début de l'algorithme
- Vérifier par récurrence qu'il reste vrai après chaque exécution de la boucle

### Exemple : Tri par sélection

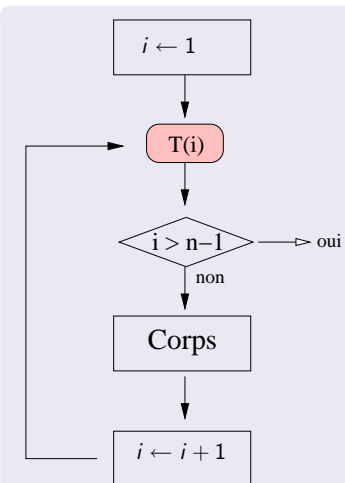


## Invariant de boucle for (1)



$S(j)$  : la valeur de *plusPetit* est l'indice du plus petit élément de  $A[i \dots j - 1]$ .

## Invariant de boucle for (2)



$T(i)$  :

- $A[1 \dots i - 1]$  sont triés
- Toutes les valeurs de  $A[i \dots n]$  sont supérieures ou égales à n'importe laquelle des valeurs de  $A[1 \dots i - 1]$

## Invariant de boucle Tant que

### Procédure Factorielle( $n$ )

#### Début

$i \leftarrow 2$

$f \leftarrow 1$

#### Tant que $i \leq n$ faire

$f \leftarrow f \times i$

$i \leftarrow i + 1$

#### Fin Tant que

#### Fin

- Terminaison de la boucle **Tant que**
- $S(i)$  : si nous atteignons le test " $i \leq n$ " alors la variable  $f$  vaut  $(i - 1)!$ .

## Généralisation du principe de récurrence - 1

## Définition : Ensembles bien fondés

Un ensemble  $E$  muni d'une relation d'ordre  $\leq$  est bien fondé s'il n'y a pas de suite infinie strictement décroissante d'éléments de  $E$ .

**Proposition :** Un ensemble ordonné  $(E, \leq)$  est bien fondé ssi toute partie non vide de  $E$  admet au moins un élément minimal.

## Exemples :

- $\mathbb{N}$  muni de l'ordre naturel.
- L'ordre produit sur  $\mathbb{N}^2$  :  $(x_1, x_2) \leq (y_1, y_2)$  ssi  $x_1 \leq y_1$  et  $x_2 \leq y_2$ .
- L'ordre lexicographique  $<$  sur  $\mathbb{N}^2$  :  $(x, y) < (x', y')$  ssi  $(x < x')$  ou  $(x = x'$  et  $y < y')$ .

## Généralisation du principe de récurrence - 2

## Théorème :

Soit  $\leq$  un ordre bien fondé sur un ensemble  $E$  et  $P$  un prédicat qui dépend d'un élément  $x$  de  $E$ . Si on a :

- (B)  $P(x_0)$  vrai pour tout  $x_0$  élément minimal de  $E$
- (I)  $\forall x \in E, [(\forall y < x, P(y)) \Rightarrow P(x)]$

alors  $P(x)$  est vrai pour tout  $x$  de  $E$ .

## Exemple :

- Terminaison de la fonction d'Ackermann :
  - $A(0, n) = n + 1$
  - $A(m, 0) = A(m - 1, 1)$
  - $A(m, n) = A(m - 1, A(m, n - 1))$

## Ensembles définis inductivement

## Définition :

Soit  $E$  un ensemble. On définit inductivement une partie  $X$  de  $E$  par

$$B \subseteq E \text{ et } \phi_i : E^{a(\phi_i)} \rightarrow E, i = 1 \dots k$$

$X$  est le plus petit ensemble vérifiant :

- (B)  $\forall x \in B, x \in X$
- (I)  $\forall i, \forall x_1, \dots, x_{a(\phi_i)} \in X \Rightarrow \phi_i(x_1, \dots, x_{a(\phi_i)}) \in X$

## Induction dans les ensembles définis inductivement

## Théorème :

Soit  $X$  un ensemble défini inductivement et soit  $P(x)$  un prédicat exprimant une propriété de l'élément  $x$  de  $X$ . Si les conditions suivantes sont vérifiées :

- (B)  $P(x)$  vrai pour tout  $x \in B$
- (I)  $(P(x_1), \dots, P(x_{a(\phi_i)})) \Rightarrow P(\phi_i(x_1, \dots, x_{a(\phi_i)}))$  pour tout  $\phi_i, i = 1 \dots k$

alors  $P(x)$  est vrai pour tout  $x$  de  $X$ .

## Exemple d'induction

## Arbre binaire

L'ensemble *ArbreBin* des arbres binaires étiquetés sur  $\Sigma$  est :

- Base :  $ABvide \in ArbreBin$
- Induction :  $\forall a \in \Sigma$  et  $\forall g, d \in ArbreBin : (a, g, d) \in ArbreBin$

## Preuve inductive

On définit la hauteur d'un arbre binaire étiqueté sur  $\Sigma$  par :

- $h(ABvide) = -1$
- $a \in \Sigma$  et  $g, d \in ArbreBin, h((a, g, d)) = 1 + \max(h(g), h(d))$

Le nombre de nœuds d'un arbre binaire est :

- $n(ABvide) = 0$
- $a \in \Sigma$  et  $g, d \in ArbreBin, n((a, g, d)) = 1 + n(g) + n(d)$

**Prouver que  $n(a, g, d) \leq 2^{h(a, g, d)+1} - 1$**

## Récursivité et approche "Diviser pour régner"

## Récursion

- Appel direct ou indirect
- Fonction récursive vs itérative
- Les appels se font avec des arguments de taille plus petite

## Diviser pour régner

- Découpage du problème en sous-problèmes similaires
- Les sous-problèmes sont "plus petits"
- Un des sous-problèmes peut-être résolu

## Procédure récursive : le tri par sélection

- (B) Si  $i = n$ , il ne reste qu'un élément à trier, il n'y a donc plus rien à faire
- (R) Si  $i < n$ , alors trouver le plus petit élément de  $A[i \dots n]$ , l'échanger avec  $A[i]$  et trier récursivement  $A[i + 1 \dots n]$

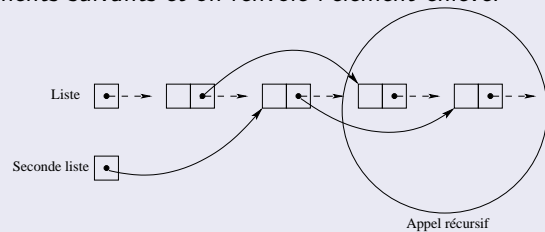
## Procédure récursive : le tri par fusion

- 1 Séparation de la liste à trier en deux listes de même taille
- 2 Chacune des listes est triée séparément
- 3 Fusion des listes triées

## Tri fusion : séparation (1)

On enlève un élément sur deux de la liste que l'on met dans une seconde liste.

- (B) Si la longueur de la liste est 0 ou 1, il n'y a rien à faire.
- (R) Lorsque la liste contient au moins deux éléments, on enlève de la liste le second élément, on sépare récursivement les éléments suivants et on renvoie l'élément enlevé.



## Tri fusion : séparation (2)

```

Fonction Separation(liste: TypeListe): TypeListe;
Variable liste2: TypeListe
Début
  Si liste est vide alors Retourner vide
  Sinon
    Si liste-suivant est vide alors Retourner vide
    Sinon
      liste2 ← liste-suivant
      liste-suivant ← liste2-suivant
      liste2-suivant ← Separation(liste2-suivant)
    Retourner liste2
  Fin Si
Fin Si
Fin
  
```

## Tri fusion : fusion des deux listes triées

```

Fonction Fusion(liste1, liste2: TypeListe): TypeListe
Début
  Si liste1 est vide alors Retourner liste2
  Sinon
    Si liste2 est vide alors Retourner liste1
    Sinon
      Si (liste1-élément ≤ liste2-élément) alors
        liste1-suivant ← Fusion(liste1-suivant, liste2)
        Retourner liste1
      Sinon
        liste2-suivant ← Fusion(liste1, liste2-suivant)
        Retourner liste2
    Fin Si
  Fin Si
Fin Si
Fin
  
```

## Tri fusion (1)

L'algorithme de tri :

- (B) Si la longueur de la liste est 0 ou 1, elle est déjà triée.
- (R) Si la liste est au moins de longueur 2, on sépare la liste en deux. Chacune des deux listes est ensuite triée récursivement, puis les deux listes triées sont fusionnées.

## Tri fusion (2)

```
Function Tri_Fusion(liste: TypeListe): TypeListe
Variable liste2: TypeListe
Début
  Si liste n'est pas vide alors
    Si liste-suivant n'est pas vide alors
      liste2 ← Separation(liste);
      Tri_Fusion(liste);
      Tri_Fusion(liste2);
    Retourner Fusion(liste, liste2)
  Fin Si
Fin Si
Fin
```

## Prouver des propriétés de programme

Notion de “taille d'arguments”

- valeur d'un argument
- longueur d'une liste
- une fonction des arguments

La taille des arguments doit décroître à chaque appel.

$S(n)$  : lorsque TRI\_FUSION est appelée sur une liste LISTE de longueur  $n$  alors elle retourne LISTE, une version triée de la liste.

## Résumé

- Les preuves par récurrence, les définitions récursives et les programmes récursifs sont des notions proches
- Les algorithmes récursif sont souvent plus simples à concevoir et à implémenter
- Il n'existe pas de programme sachant tester si une fonction récursive termine
  - Existence de preuve de terminaison dans certains cas
  - Mais pas toujours de preuve
- La récurrence est utile pour prouver qu'un programme ou un morceau de programme fonctionne correctement
- Généralisation du principe de récurrence à d'autres modèles de données