Fonctions récursives

Lycée Pierre Corneille – MP

2016-2017

Définition

En mathématiques, une suite $(u_n)_{n\in\mathbb{N}}$ est **récurrente** lorsque le terme u_{n+1} est une fonction du terme u_n.

Définition

En mathématiques, une suite $(u_n)_{n\in\mathbb{N}}$ est **récurrente** lorsque le terme u_{n+1} est une fonction du terme u_n.

En informatique, une fonction f est **récursive** lorsque la définition de f utilise des valeurs de f.

Définition

En mathématiques, une suite $(u_n)_{n\in\mathbb{N}}$ est **récurrente** lorsque le terme u_{n+1} est une fonction du terme u_n.

En informatique, une fonction f est **récursive** lorsque la définition de f utilise des valeurs de f.

Chaque fonction récursive est construite sur une relation de récurrence.

Exemple fondamental

Fonction récursive

```
def f(n):
"""
  n est un entier naturel
"""
  if (n==0):
    return 1
  else:
  # Appel à la fonction f
    return n*f(n-1)
```

Exemple fondamental

Fonction récursive

```
def f(n):
"""
  n est un entier naturel
"""
  if (n==0):
    return 1
  else:
  # Appel à la fonction f
    return n*f(n-1)
```

Traduction mathématique

- Fondation : $f_0 = 1$

Exemple fondamental

Fonction récursive

```
def f(n):
"""
  n est un entier naturel
"""
  if (n==0):
    return 1
  else:
  # Appel à la fonction f
    return n*f(n-1)
```

Traduction mathématique

- Fondation : $f_0 = 1$
- Relation de récurrence

$$\forall\, n \geqslant 1, \quad f_n = n \times f_{n-1}$$

Exemple fondamental

Fonction récursive

```
def f(n):
"""
  n est un entier naturel
"""
  if (n==0):
    return 1
  else:
  # Appel à la fonction f
    return n*f(n-1)
```

Traduction mathématique

- Fondation : $f_0 = 1$
- Relation de récurrence

$$\forall\, n \geqslant 1, \quad f_n = n \times f_{n-1}$$

- Conclusion

$$\forall\, n \in \mathbb{N}, \qquad f_n = n!$$

Construire une fonction récursive

- Formuler une relation de récurrence

Construire une fonction récursive

- Formuler une relation de récurrence
- Préciser la fondation (valeur initiale)

Construire une fonction récursive

- Formuler une relation de récurrence

  Comment descendre une marche ?

- Préciser la fondation (valeur initiale)

Construire une fonction récursive

- Formuler une relation de récurrence

  Comment descendre une marche ?

- Préciser la fondation (valeur initiale)

  Que faire au bas de l'escalier ?

Écriture de la fonction

- Écriture plus compacte que l'écriture impérative (indices, variables locales)

Écriture de la fonction

- Écriture plus compacte que l'écriture impérative (indices, variables locales)
- Parfois indigeste !

Écriture de la fonction

- Écriture plus compacte que l'écriture impérative (indices, variables locales)
- Parfois indigeste !
- Problèmes naturellement récursifs.

Correction et terminaison

- La relation de récurrence utilisée est-elle correcte ?

Correction et terminaison

- La relation de récurrence utilisée est-elle correcte ?
- La fondation est-elle complète ? (Envisager *tous* les cas !)

Correction et terminaison

- La relation de récurrence utilisée est-elle correcte ?
- La fondation est-elle complète ? (Envisager *tous* les cas !)
- Chaque exécution de la fonction doit rapprocher de la fondation.

Complexité

- Relation de récurrence facile à trouver.

Complexité

- Relation de récurrence facile à trouver.
- Ordre de grandeur pas toujours facile à percevoir.

Complexité

- Relation de récurrence facile à trouver.
- Ordre de grandeur pas toujours facile à percevoir.
- Aucun avantage par rapport à une fonction impérative.

Profondeur de récursion

- **Fonction impérative**
  Chaque itération modifie les valeurs de variables locales (*effets de bord*).

Profondeur de récursion

- **Fonction impérative**
  Chaque itération modifie les valeurs de variables locales (*effets de bord*).
- **Fonction récursive**
  Chaque itération appelle la fonction avant de poursuivre le calcul.

Profondeur de récursion

- **Fonction impérative**
  Chaque itération modifie les valeurs de variables locales (*effets de bord*).
- **Fonction récursive**
  Chaque itération appelle la fonction avant de poursuivre le calcul.
  - Le calcul est en *suspens* tant qu'on n'a pas atteint la fondation.

Profondeur de récursion

- **Fonction impérative**
  Chaque itération modifie les valeurs de variables locales (*effets de bord*).
- **Fonction récursive**
  Chaque itération appelle la fonction avant de poursuivre le calcul.
  - Le calcul est en *suspens* tant qu'on n'a pas atteint la fondation.
  - Nombre d'opérations en suspens limité par le langage de programmation.

Profondeur de récursion

- **Fonction impérative**
  Chaque itération modifie les valeurs de variables locales (*effets de bord*).
- **Fonction récursive**
  Chaque itération appelle la fonction avant de poursuivre le calcul.
  - Le calcul est en *suspens* tant qu'on n'a pas atteint la fondation.
  - Nombre d'opérations en suspens limité par le langage de programmation.
  - Python : si plus de 1 000 parenthèses ouvertes simultanément,

    RuntimeError : maximum recursion depth exceeded

Conclusion

La programmation récursive est un style qu'on peut éviter...

Conclusion

La programmation récursive est un style qu'on peut éviter... sauf pour certains problèmes spécifiques (informatique fondamentale).

Somme d'une liste de nombres

Relation de récurrence

$$\sum_{k=0}^{n} x_k = x_0 + \sum_{k=1}^{n} x_k$$

pour tout $n \geqslant 1$ et par convention

$$\sum_{k=1}^{0} x_k = 0.$$

Somme d'une liste de nombres

Relation de récurrence

$$\sum_{k=0}^{n} x_k = x_0 + \sum_{k=1}^{n} x_k$$

pour tout $n \geqslant 1$ et par convention

$$\sum_{k=1}^{0} x_k = 0.$$

Code récursif

```
def somme(L):
  if (len(L)==0):
    return 0
  else:
    return L[0]+somme(L[1:])
```

Maximum d'une liste d'entiers

On dispose d'une fonction pour calculer le maximum de *deux* entiers.

```
def max2(x, y):
  if (x<y):
    return y
  else:
    return x
```

Maximum d'une liste d'entiers

On dispose d'une fonction pour calculer le maximum de *deux* entiers.

```
def max2(x, y):
  if (x<y):
    return y
  else:
    return x
```

Comment calculer le maximum d'une liste (non vide) d'entiers ?

Maximum d'une liste d'entiers

Cas d'un singleton :

$$\max(a_0) = a_0$$

Maximum d'une liste d'entiers

Cas d'un singleton :

$$\max(a_0) = a_0$$

Cas général : pour tout $n \geqslant 1$,

$$\max(a_0, a_1, \ldots, a_n) = \max\{a_0, \max(a_1, \ldots, a_n)\}$$

Maximum d'une liste d'entiers

Cas d'un singleton :

$$\max(a_0) = a_0$$

Cas général : pour tout $n \geqslant 1$,

$$\max(a_0, a_1, \ldots, a_n) = \max\{a_0, \max(a_1, \ldots, a_n)\}$$

```
def max_liste(L):
  if (len(L)==1):
    return L[0]
  else:
    return max2(L[0], max_liste(L[1:]))
```

Insertion dans une liste triée

Données

- Réel a
- Liste réelle L triée $L_0 \leqslant L_1 \leqslant \cdots \leqslant L_{n-1}$

Insertion dans une liste triée

Données

- Réel a
- Liste réelle L triée $L_0 \leqslant L_1 \leqslant \cdots \leqslant L_{n-1}$

➙ Où insérer a dans L pour que la liste reste triée ?

Insertion dans une liste triée

Données

- Réel a
- Liste réelle L triée $L_0 \leqslant L_1 \leqslant \cdots \leqslant L_{n-1}$

➙ Où insérer a dans L pour que la liste reste triée ?

- Fondation 1 : Si L est vide, alors (a) est triée.

Insertion dans une liste triée

Données

- Réel a
- Liste réelle L triée $L_0 \leqslant L_1 \leqslant \cdots \leqslant L_{n-1}$

➙ Où insérer a dans L pour que la liste reste triée ?

- Fondation 1 : Si L est vide, alors (a) est triée.
- Fondation 2 : Si $a < L_0$, alors $(a, L_0, L_1, \ldots, L_{n-1})$ est triée.

Insertion dans une liste triée

Données

- Réel a
- Liste réelle L triée $L_0 \leqslant L_1 \leqslant \cdots \leqslant L_{n-1}$

➙ Où insérer a dans L pour que la liste reste triée ?

- Fondation 1 : Si L est vide, alors (a) est triée.
- Fondation 2 : Si $a < L_0$, alors $(a, L_0, L_1, \ldots, L_{n-1})$ est triée.
- Cas général : Si $L_k \leqslant a$, il faut insérer a dans la sous-liste

$$(L_{k+1}, \ldots, L_{n-1})$$

Insertion dans une liste triée

```
def inserer(a, L):
  if (L==[]):                  # Fondation 1
    return [a]
  elif (a<L[0]):               # Fondation 2
    return [a] + L
  else:
    return [L[0]] + inserer(a,L[1:])
```

Résolution de $f(x) = 0$ par dichotomie

Propriétés de la fonction f :

- strictement monotone
- continue sur le segment $[a, b]$
- changement de signe : $f(a)f(b) < 0$.

Résolution de $f(x) = 0$ par dichotomie

Propriétés de la fonction f :

- strictement monotone
- continue sur le segment $[a, b]$
- changement de signe : $f(a)f(b) < 0$.

L'équation $f(x) = 0$ admet une, et une seule, solution $x_0 \in [a, b]$.

➙ Comment trouver α et β tels que

$$\alpha \leqslant x_0 \leqslant \beta \quad \text{et} \quad \beta - \alpha \leqslant \varepsilon \ ?$$

Résolution de $f(x) = 0$ par dichotomie

Propriétés de la fonction f :

- strictement monotone
- continue sur le segment $[a, b]$
- changement de signe : $f(a)f(b) < 0$.

L'équation $f(x) = 0$ admet une, et une seule, solution $x_0 \in [a, b]$.
➙ Comment trouver α et β tels que

$$\alpha \leqslant x_0 \leqslant \beta \quad \text{et} \quad \beta - \alpha \leqslant \varepsilon \ ?$$

On pose $\alpha_0 = a$, $\beta_0 = b$ et on itère :

$$\forall\ n \geqslant 1, \quad f(\alpha_n)f(\beta_n) < 0 \quad \text{et} \quad \beta_n - \alpha_n = \frac{\beta_{n-1} - \alpha_{n-1}}{2}.$$

Résolution de $f(x) = 0$ par dichotomie

Code récursif

```
def resoudre(f, a, b, epsilon=0.01):
  if (b-a < epsilon):
    return (a,b)
  else:
    c = (b+a)/2
    if (f(a)*f(c) > 0):
      return resoudre(f, c, b, epsilon)
    else:
      return resoudre(f, a, c, epsilon)
```

Recherche dichotomique dans une liste triée

- **Données**
  - Réel x
  - Liste réelle

$$\mathsf{L} = (L_0, L_1, \ldots, L_{n-1})$$

triée par ordre croissant

Recherche dichotomique dans une liste triée

- **Données**
  - Réel x
  - Liste réelle

    $$\mathtt{L} = (L_0, L_1, \ldots, L_{n-1})$$

    triée par ordre croissant
- **Objectif**

  Trouver k tel que

  $$L_k \leqslant x < L_{k+1}$$

  à moins que

  $$x < L_0 \qquad \text{ou que} \qquad x > L_{n-1}.$$

Recherche dichotomique dans une liste triée

Traitement des cas exceptionnels : $[x < L_0]$, $[L_{n-1} \leqslant x]$ puis du cas général

$$[L_0 \leqslant x < L_{n-1}]$$

```
def recherche_dicho(x, L):
  n = len(L)
  if (x<L[0]):
    return (None,0)
  elif (x>=L[-1]):
    return (n-1,None)
  else:
    return rech_rec(x, L, 0, n-1)
```

Recherche dichotomique dans une liste triée

On pose $i_0 = 0$, $j_0 = n - 1$ et on itère :

$$\forall\, n \geqslant 1, \quad L_{i_n} \leqslant x < L_{j_n} \quad \text{et} \quad j_n - i_n = \frac{j_{n-1} - i_{n-1}}{2}$$

```
def rech_rec(x, L, i, j):
  if (j-i==1):
    return (i,j)
  else:
    m = (j+i)//2
    if (x<L[m]):
      return rech_rec(x, L, i, m)
    else:
      return rech_rec(x, L, m, j)
```

Position d'un élément dans une liste

L'objet x appartient-il à la liste L ?

$$\texttt{L} = (L_0, \ldots, L_{i-1}, \texttt{x}, L_{i+1}, \ldots, L_{n-1})$$
$$\iff \texttt{L[i:]} = (\texttt{x}, L_{i+1}, \ldots, L_{n-1})$$

Position d'un élément dans une liste

L'objet x appartient-il à la liste L ?

$$\texttt{L} = (L_0, \ldots, L_{i-1}, \texttt{x}, L_{i+1}, \ldots, L_{n-1})$$
$$\iff \texttt{L[i:]} = (\texttt{x}, L_{i+1}, \ldots, L_{n-1})$$

- Le compteur `i` est initialisé à 0.

Position d'un élément dans une liste

L'objet x appartient-il à la liste L ?

$$\texttt{L} = (L_0, \ldots, L_{i-1}, \texttt{x}, L_{i+1}, \ldots, L_{n-1})$$
$$\iff \texttt{L[i:]} = (\texttt{x}, L_{i+1}, \ldots, L_{n-1})$$

- Le compteur `i` est initialisé à 0.
- Invariant de boucle : À chaque appel à `position`,

$$\texttt{L} \leftarrow \texttt{L[1:]} \quad \text{et} \quad \texttt{i} \leftarrow \texttt{i} + 1$$

donc la somme `i+len(L)` est constante.

Position d'un élément dans une liste

L'objet x appartient-il à la liste L ?

$$\texttt{L} = (L_0, \ldots, L_{i-1}, \texttt{x}, L_{i+1}, \ldots, L_{n-1})$$
$$\iff \texttt{L[i:]} = (\texttt{x}, L_{i+1}, \ldots, L_{n-1})$$

- Le compteur `i` est initialisé à 0.
- Invariant de boucle : À chaque appel à `position`,

$$\texttt{L} \leftarrow \texttt{L[1:]} \quad \text{et} \quad \texttt{i} \leftarrow \texttt{i} + 1$$

  donc la somme `i+len(L)` est constante.
- Terminaison 1 : on arrive à vider la liste sans trouver `x`.

Position d'un élément dans une liste

L'objet x appartient-il à la liste L ?

$$\texttt{L} = (L_0, \ldots, L_{i-1}, \texttt{x}, L_{i+1}, \ldots, L_{n-1})$$
$$\iff \texttt{L[i:]} = (\texttt{x}, L_{i+1}, \ldots, L_{n-1})$$

- Le compteur `i` est initialisé à 0.
- Invariant de boucle : À chaque appel à `position`,

$$\texttt{L} \leftarrow \texttt{L[1:]} \quad \text{et} \quad \texttt{i} \leftarrow \texttt{i} + 1$$

  donc la somme `i+len(L)` est constante.
- Terminaison 1 : on arrive à vider la liste sans trouver `x`.
- Terminaison 2 : on trouve `x` en position `0` dans la liste `L[i:]`.

Position d'un élément dans une liste

```
def position(x, L, i=0):
  if (L==[]):              Terminaison 1
    return None
  elif (x==L[0]):          Terminaison 2
    return i
  else:
    return position(x, L[1:], i+1)
```

Décomposition en base 2

Cas particuliers : Si $n = 0$ ou $n = 1$, alors $n = \varepsilon_0$.

Décomposition en base 2

Cas particuliers : Si $n = 0$ ou $n = 1$, alors $n = \varepsilon_0$.
Cas général :

$$\mathscr{B}(n) = (\varepsilon_d, \ldots, \varepsilon_1, \varepsilon_0) \iff n = \sum_{k=0}^{d} \varepsilon_k 2^k$$

$$= (\varepsilon_d, \ldots, \varepsilon_1), (\varepsilon_0) \qquad = \left(\sum_{k=0}^{d-1} \varepsilon_{k+1} 2^k \right) \times 2 + \varepsilon_0$$

Décomposition en base 2

Cas particuliers : Si $n = 0$ ou $n = 1$, alors $n = \varepsilon_0$.
Cas général :

$$\mathscr{B}(n) = (\varepsilon_d, \ldots, \varepsilon_1, \varepsilon_0) \iff n = \sum_{k=0}^{d} \varepsilon_k 2^k$$

$$= (\varepsilon_d, \ldots, \varepsilon_1), (\varepsilon_0) \qquad = \left(\sum_{k=0}^{d-1} \varepsilon_{k+1} 2^k \right) \times 2 + \varepsilon_0$$

```
def binaire(n):
  if (n<2):
    return [n]
  else:
    return binaire(n//2)+[n%2]
```

Exponentiation rapide

Si

$$n = \sum_{k=0}^{d} \varepsilon_k 2^k,$$

alors

$$x^n = \prod_{\substack{0 \leqslant k \leqslant d \\ \varepsilon_k = 1}} x^{(2^k)}$$

sachant que $x^{2^0} = x$ et que

$$x^{(2^{k+1})} = x^{(2 \times 2^k)} = \left(x^{(2^k)}\right)^2$$

pour tout $k \in \mathbb{N}$.

Exponentiation rapide

Si

$$n = \sum_{k=0}^{d} \varepsilon_k 2^k,$$

alors

$$x^n = \prod_{\substack{0 \leqslant k \leqslant d \\ \varepsilon_k = 1}} x^{(2^k)}$$

sachant que $x^{2^0} = x$ et que

$$x^{(2^{k+1})} = x^{(2 \times 2^k)} = \left(x^{(2^k)}\right)^2$$

pour tout $k \in \mathbb{N}$.

Code récursif

```
def puissance(x, n):
  if (n==1):
    return x
  else:
    y = puissance(x, n//2)
    if (n%2==1):
      return x*y*y
    else:
      return y*y
```

Algorithme d'Euclide

Cas particulier :

$$a \wedge 0 = a$$

Algorithme d'Euclide

Cas particulier :

$$a \wedge 0 = a$$

Cas général pour $b \neq 0$:

$$a \wedge b = b \wedge r$$

où $0 \leqslant r < |b|$ est le reste de la division euclidienne de a par b.

Algorithme d'Euclide

Cas particulier :

$$a \wedge 0 = a$$

Cas général pour $b \neq 0$:

$$a \wedge b = b \wedge r$$

où $0 \leqslant r < |b|$ est le reste de la division euclidienne de a par b.

```
def pgcd(a, b):
  if (b==0):
    return a
  else:
    return pgcd(b, a%b)
```

Algorithme d'Euclide pour une liste d'entiers

Cas particulier :

$$\mathrm{pgcd}(a_0, a_1) = a_0 \wedge a_1$$

Algorithme d'Euclide pour une liste d'entiers

Cas particulier :

$$\mathrm{pgcd}(a_0, a_1) = a_0 \wedge a_1$$

Cas général pour $n \geqslant 3$:

$$\mathrm{pgcd}(a_0, a_1, \ldots, a_{n-1}) = a_0 \wedge \mathrm{pgcd}(a_1, \ldots, a_{n-1})$$

Algorithme d'Euclide pour une liste d'entiers

Cas particulier :

$$\mathrm{pgcd}(a_0, a_1) = a_0 \wedge a_1$$

Cas général pour $n \geqslant 3$:

$$\mathrm{pgcd}(a_0, a_1, \ldots, a_{n-1}) = a_0 \wedge \mathrm{pgcd}(a_1, \ldots, a_{n-1})$$

```
def pgcd_liste(L):
  if (len(L)==2):
    return pgcd(L[0], L[1])
  else:
    return pgcd(L[0], pgcd_liste(L[1:]))
```