Semestre 1 / Cours 5 - Recursivité

ITC MPSI & PCSI – Année 2024-2025

Sommaire

Généralités

La factorielle : mode explicite $\leftrightarrow$ programmation impérative

- $\forall n \in \mathbb{N}, \quad u_n = 1 \times 2 \times \cdots \times n = \prod_{k=1}^{n} k \quad$ (convention $u_0 = 1$).

La factorielle : mode explicite $\leftrightarrow$ programmation impérative

- $\forall n \in \mathbb{N}, \quad u_n = 1 \times 2 \times \cdots \times n = \prod_{k=1}^{n} k$ (convention $u_0 = 1$).
- [Code]

```
def factorielle(n):
    """ renvoie la factorielle de l'entier naturel \
    ↪ n. """
    if n  ==  0:
        return 1
    else:
        f = 1
        for k in range (1, n+1):
            f *= k
        return f
```

La factorielle : mode récurrent $\leftrightarrow$ programmation récursive

- $\forall n \in \mathbb{N}^\star, \quad u_n = n \times u_{n-1}, \quad u_0 = 1.$

La factorielle : mode récurrent $\leftrightarrow$ programmation récursive

- $\forall n \in \mathbb{N}^\star, \quad u_n = n \times u_{n-1}, \quad u_0 = 1.$
- Définition faisant appel à la factorielle elle-même mais sur une valeur plus petite, caractéristique du principe **récursif**.

La factorielle : mode récurrent $\leftrightarrow$ programmation récursive

Le calcul de $4!$ se fait alors ainsi :

- **[Phase d'empilement]**

$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times (3 \times 2!) \\ &= 4 \times (3 \times (2 \times 1!)) \\ &= 4 \times (3 \times (2 \times (1 \times 0!))) \\ &= 4 \times (3 \times (2 \times (1 \times 1))). \end{aligned}$$

La factorielle : mode récurrent $\leftrightarrow$ programmation récursive

Le calcul de 4! se fait alors ainsi :

- **[Phase d'empilement]**

$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times (3 \times 2!) \\ &= 4 \times (3 \times (2 \times 1!)) \\ &= 4 \times (3 \times (2 \times (1 \times 0!))) \\ &= 4 \times (3 \times (2 \times (1 \times 1))). \end{aligned}$$

- **[Phase de dépilement]** réaliser les multiplications en commençant par la dernière :

$$\begin{aligned} 4 \times (3 \times (2 \times (1 \times 1))) &= 4 \times (3 \times (2 \times 1)) \\ &= 4 \times (3 \times 2) = 4 \times 6 = 24. \end{aligned}$$

La factorielle : mode récurrent $\leftrightarrow$ programmation récursive

```
def factorielle(n : int) -> int :
    """ renvoie la factorielle de l'entier naturel n \
    ↪ """
    if n == 0:
        return 1
    else :
        return n*factorielle(n-1)
```

La factorielle : mode récurrent $\leftrightarrow$ programmation récursive

```
def factorielle(n : int) -> int :
    """ renvoie la factorielle de l'entier naturel n \
    ↪ """
    if n == 0:
        return 1
    else :
        return n*factorielle(n-1)
```

Exemple :

- l'appel `factorielle(3)` commence et s'interrompt en déclenchant l'appel de `factorielle(2)`;

La factorielle : mode récurrent $\leftrightarrow$ programmation récursive

```
def factorielle(n : int) -> int :
    """ renvoie la factorielle de l'entier naturel n \
    ↪ """
    if n == 0:
        return 1
    else :
        return n*factorielle(n-1)
```

Exemple :

- l'appel `factorielle(3)` commence et s'interrompt en déclenchant l'appel de `factorielle(2)`;
  - l'appel `factorielle(2)` commence et s'interrompt en déclenchant l'appel de `factorielle(1)`;

La factorielle : mode récurrent $\leftrightarrow$ programmation récursive

```
def factorielle(n : int) -> int :
    """ renvoie la factorielle de l'entier naturel n \
    ↪ """
    if n == 0:
        return 1
    else :
        return n*factorielle(n-1)
```

Exemple :

- l'appel `factorielle(3)` commence et s'interrompt en déclenchant l'appel de `factorielle(2)`;
  - l'appel `factorielle(2)` commence et s'interrompt en déclenchant l'appel de `factorielle(1)`;
    - l'appel `factorielle(1)` commence et s'interrompt en déclenchant l'appel de `factorielle(0)`;

La factorielle : mode récurrent $\leftrightarrow$ programmation récursive

```
def factorielle(n : int) -> int :
    """ renvoie la factorielle de l'entier naturel n \
    ↪ """
    if n == 0:
        return 1
    else :
        return n*factorielle(n-1)
```

Exemple :

- l'appel `factorielle(3)` commence et s'interrompt en déclenchant l'appel de `factorielle(2)`;
  - ◇ l'appel `factorielle(2)` commence et s'interrompt en déclenchant l'appel de `factorielle(1)`;
    - — l'appel `factorielle(1)` commence et s'interrompt en déclenchant l'appel de `factorielle(0)`;
      - ⋆ l'appel `factorielle(0)` va jusqu'à son terme et renvoie la valeur 1 à la fonction `factorielle(1)` qui l'a appelée;

La factorielle : mode récurrent $\leftrightarrow$ programmation récursive

```
def factorielle(n : int) -> int :
    """ renvoie la factorielle de l'entier naturel n \
    ↪ """
    if n == 0:
        return 1
    else :
        return n*factorielle(n-1)
```

Exemple :

- l'appel `factorielle(3)` commence et s'interrompt en déclenchant l'appel de `factorielle(2)` ;
  - ◇ l'appel `factorielle(2)` commence et s'interrompt en déclenchant l'appel de `factorielle(1)` ;
    - — l'appel `factorielle(1)` commence et s'interrompt en déclenchant l'appel de `factorielle(0)` ;
      - ⋆ l'appel `factorielle(0)` va jusqu'à son terme et renvoie la valeur 1 à la fonction `factorielle(1)` qui l'a appelée ;
    - — l'appel `factorielle(1)` reprend, termine son calcul $1 \times 1 = 1$, et renvoie cette valeur à l'appel `factorielle(2)` ;
  - ◇ l'appel `factorielle(2)` reprend, termine son calcul $2 \times 1 = 2$, et renvoie cette valeur à l'appel `factorielle(3)` ;
- l'appel `factorielle(3)` reprend, termine son calcul $3 \times 2 = 6$, et renvoie cette valeur, qui est l'unique valeur de retour de l'appel initial.

Pile d'appels récursifs

- Mécanisme consistant à stopper le déroulement d'un appel pour en permettre un autre : gourmand en mémoire.

Pile d'appels récursifs

- Mécanisme consistant à stopper le déroulement d'un appel pour en permettre un autre : gourmand en mémoire.
- Dans les faits, nombre d'appels stockés limité par Python (environ 1000).

Pile d'appels récursifs

- Mécanisme consistant à stopper le déroulement d'un appel pour en permettre un autre : gourmand en mémoire.
- Dans les faits, nombre d'appels stockés limité par Python (environ 1000).
- Si la taille est trop grande, erreur :

  ```
  RecursionError: maximum recursion depth exceeded
  ```

- Possibilité souvent de réecrire des fonctions récursives sans utiliser de pile d'appels ;

Pile d'appels récursifs

- Mécanisme consistant à stopper le déroulement d'un appel pour en permettre un autre : gourmand en mémoire.
- Dans les faits, nombre d'appels stockés limité par Python (environ 1000).
- Si la taille est trop grande, erreur :

  ```
  RecursionError: maximum recursion depth exceeded
  ```

- Possibilité souvent de réecrire des fonctions récursives sans utiliser de pile d'appels ; revient dans la factorielle à ne pas utiliser le symbole `*`

Pile d'appels récursifs

- Mécanisme consistant à stopper le déroulement d'un appel pour en permettre un autre : gourmand en mémoire.
- Dans les faits, nombre d'appels stockés limité par Python (environ 1000).
- Si la taille est trop grande, erreur :

  ```
  RecursionError: maximum recursion depth exceeded
  ```

- Possibilité souvent de réecrire des fonctions récursives sans utiliser de pile d'appels ; revient dans la factorielle à ne pas utiliser le symbole `*` (« récursivité terminale » - non présentée dans ce cours)

Pile d'appels récursifs

- Mécanisme consistant à stopper le déroulement d'un appel pour en permettre un autre : gourmand en mémoire.
- Dans les faits, nombre d'appels stockés limité par Python (environ 1000).
- Si la taille est trop grande, erreur :

  ```
  RecursionError: maximum recursion depth exceeded
  ```

- Possibilité souvent de réecrire des fonctions récursives sans utiliser de pile d'appels ; revient dans la factorielle à ne pas utiliser le symbole `*` (« récursivité terminale » - non présentée dans ce cours)
- Vigilance à avoir sur le cas terminal afin que la série d'appels récursifs s'arrête.

Définition

Fonction récursive
Une fonction *récursive* est une fonction qui s'appelle une ou plusieurs fois dans son corps.

Exercice

```python
def mystere(n):
    if n == 0:
        return 1 # cas de base
    else:
        return 2*mystere(n-1)
```

Conjecturer ce que renvoie cette fonction et compléter sa signature. *On pourra effectuer (au stylo) le déroulé de cette fonction sur, par exemple, l'appel* `mystere(3)`

Solution

Dans l'appel `mystere(3)` :

- puisque $3 \neq 0$, on calcule `2*mystere(2)`,

Solution

Dans l'appel `mystere(3)` :

- puisque $3 \neq 0$, on calcule `2*mystere(2)`,
- puisque $2 \neq 0$, on calcule `2*2*mystere(1)`,

Solution

Dans l'appel `mystere(3)` :

- puisque $3 \neq 0$, on calcule `2*mystere(2)`,
- puisque $2 \neq 0$, on calcule `2*2*mystere(1)`,
- puisque $1 \neq 0$, on calcule `2*2*2*mystere(0)`,
- comme $0 = 0$, on est sur un cas de base, on calcule `2*2*2*1`.
- Phase de dépilement : les multiplications sont réalisées, la fonction renvoie $\boxed{8}$.

Solution

Dans l'appel `mystere(3)` :

- puisque $3 \neq 0$, on calcule `2*mystere(2)`,
- puisque $2 \neq 0$, on calcule `2*2*mystere(1)`,
- puisque $1 \neq 0$, on calcule `2*2*2*mystere(0)`,
- comme $0 = 0$, on est sur un cas de base, on calcule `2*2*2*1`.
- Phase de dépilement : les multiplications sont réalisées, la fonction renvoie $\boxed{8}$.

```
def mystere(n:int)->int:
    """renvoie la valeur de 2**n"""
    if n == 0:
        return 1 # cas de base
    else:
        return 2*mystere(n-1)
```

```
>>> mystere(3)
8
```

Exercice

```python
def mystere(L):
    if len(L) == 0:
        return 0 # cas de base
    else:
        return L[0] + mystere(L[1:])
```

Conjecturer ce que renvoie cette fonction et compléter sa signature. *On pourra effectuer (au stylo) le déroulé de cette fonction sur, par exemple, l'appel* `mystere([1, 4, 3])`

Solution

Dans l'appel `mystere([1, 4, 3])` :

- puisque `[1, 4, 3]` est non vide, on calcule `1 + mystere([4, 3])`,

Solution

Dans l'appel `mystere([1, 4, 3])` :

- puisque `[1, 4, 3]` est non vide, on calcule `1 + mystere([4, 3])`,
- puisque `[4, 3]` est non vide, on calcule `1 + 4 + mystere([3])`,

Solution

Dans l'appel `mystere([1, 4, 3])` :

- puisque `[1, 4, 3]` est non vide, on calcule `1 + mystere([4, 3])`,
- puisque `[4, 3]` est non vide, on calcule `1 + 4 + mystere([3])`,
- puisque `[3]` est non vide, on calcule `1 + 4 + 3 + mystere([])`,
- puisque `[]` est vide, on est sur un cas de base, on calcule `1 + 4 + 3 + 0`,
- Phase de dépilement : les additions sont réalisées, la fonction renvoie $\boxed{8}$.

Solution

Dans l'appel `mystere([1, 4, 3])` :

- puisque `[1, 4, 3]` est non vide, on calcule `1 + mystere([4, 3])`,
- puisque `[4, 3]` est non vide, on calcule `1 + 4 + mystere([3])`,
- puisque `[3]` est non vide, on calcule `1 + 4 + 3 + mystere([])`,
- puisque `[]` est vide, on est sur un cas de base, on calcule `1 + 4 + 3 + 0`,
- Phase de dépilement : les additions sont réalisées, la fonction renvoie $\boxed{8}$.

```
def mystere(L:list)->float:
    """ renvoie la somme des éléments de L """
    if len(L) == 0:
        return 0 # cas de base
    else:
        return L[0] + mystere(L[1:])
```

```
>>> mystere([1, 4, 3])
8
```

Méthodologie

Pour concevoir une fonction récursive, il faut :

- Rechercher le ou les cas de bases et leur donner une solution.

Méthodologie

Pour concevoir une fonction récursive, il faut :

- Rechercher le ou les cas de bases et leur donner une solution.
- Décomposer le problème général en sous-problèmes identiques de « taille inférieure ».

Exemple : reste de la division euclidienne

Soient $a, b \in \mathbb{N}$ deux entiers avec $b \neq 0$. Rappelons qu'il existe un unique couple $(q_{a,b}, r_{a,b})$ d'entiers tels que :

$$a = bq_{a,b} + r_{a,b}, \quad \text{et :} \quad 0 \leqslant r_{a,b} < b.$$

Exemple : reste de la division euclidienne

Soient $a, b \in \mathbb{N}$ deux entiers avec $b \neq 0$. Rappelons qu'il existe un unique couple $(q_{a,b}, r_{a,b})$ d'entiers tels que :

$$a = bq_{a,b} + r_{a,b}, \quad \text{et :} \quad 0 \leqslant r_{a,b} < b.$$

- L'entier $r_{a,b}$ est appelé *reste de la division euclidienne de a par b.*

Exemple : reste de la division euclidienne

Soient $a, b \in \mathbb{N}$ deux entiers avec $b \neq 0$. Rappelons qu'il existe un unique couple $(q_{a,b}, r_{a,b})$ d'entiers tels que :

$$a = bq_{a,b} + r_{a,b}, \quad \text{et :} \quad 0 \leqslant r_{a,b} < b.$$

- L'entier $r_{a,b}$ est appelé *reste de la division euclidienne de a par b.*
- Notons que si $a < b$, alors $r_{a,b} = a$ puisque $a = 0 \times b + a$ et que $0 \leqslant a < b$.

Exemple : reste de la division euclidienne

Cherchons une fonction récursive qui renvoie $r_{a,b}$.

- **[Cas de base]** Si $a < b$, il faut renvoyer a.

Exemple : reste de la division euclidienne

Cherchons une fonction récursive qui renvoie $r_{a,b}$.

- **[Cas de base]** Si $a < b$, il faut renvoyer a.
- **[Récursivité]** On remarque que $r_{a,b} = r_{a-b,b}$, car :

$$a = bq_{a,b} + r_{a,b} \iff a - b = b(q_{a,b} - 1) + \underbrace{r_{a,b}}_{=r_{a-b,b}}.$$

Exemple : reste de la division euclidienne

Cherchons une fonction récursive qui renvoie $r_{a,b}$.

- **[Cas de base]** Si $a < b$, il faut renvoyer a.
- **[Récursivité]** On remarque que $r_{a,b} = r_{a-b,b}$, car :

$$a = bq_{a,b} + r_{a,b} \iff a - b = b(q_{a,b} - 1) + \underbrace{r_{a,b}}_{=r_{a-b,b}}.$$

Cela mène à la fonction récursive ci-après.

```
def reste(a:int,b:int)->int:
    if a < b:
        return a
    else:
        return reste(a-b, b)
```

1er paramètre abaissé (changé en `a-b < a`) : point crucial qui permet d'arriver sur un case de base.

Exemple : reste de la division euclidienne

Cherchons une fonction récursive qui renvoie $r_{a,b}$.

- **[Cas de base]** Si $a < b$, il faut renvoyer a.
- **[Récursivité]** On remarque que $r_{a,b} = r_{a-b,b}$, car :

$$a = bq_{a,b} + r_{a,b} \iff a - b = b(q_{a,b} - 1) + \underbrace{r_{a,b}}_{=r_{a-b,b}}.$$

Cela mène à la fonction récursive ci-après.

```
def reste(a:int,b:int)->int:
    if a < b:
        return a
    else:
        return reste(a-b, b)
```

1er paramètre abaissé (changé en `a-b < a`) : point crucial qui permet d'arriver sur un case de base.

```
>>> reste(10, 3)
1
>>> reste(3, 10)
3
```

Exercice

Écrire une fonction récursive `pgcd(a:int, b:int)->int` qui renvoie le plus grand diviseur commun de 2 entiers. *On rappelle que* $pgcd(a, b) = pgcd(b, r)$ *où* r *est le reste de la division euclidienne de* a *par* b. *On pourra utiliser la fonction* `reste` *codée précédemment*

Solution

Rappel : le pgcd est le dernier reste non nul dans l'algorithme d'Euclide

Solution

Rappel : le pgcd est le dernier reste non nul dans l'algorithme d'Euclide

```
def pgcd(a:int, b:int)->int:
    if b == 0 :
        return a
    return pgcd(b, reste(a, b))
```

Solution

Rappel : le pgcd est le dernier reste non nul dans l'algorithme d'Euclide

```
def pgcd(a:int, b:int)->int:
    if b == 0 :
        return a
    return pgcd(b, reste(a, b))
```

```
>>> pgcd(3, 6)
3
>>> pgcd(6, 3)
3
>>> pgcd(3, 1)
1
```

Exercice

Écrire une fonction récursive `produit(L:list)->float` qui renvoie le produit des éléments d'une liste.

Solution

```
def produit(L:list)->float:
    """ renvoie la somme des éléments de L """
    if len(L) == 0:
        return 1 # cas de base
    else:
        return L[0] * produit(L[1:])
```

Solution

```
def produit(L:list)->float:
    """ renvoie la somme des éléments de L """
    if len(L) == 0:
        return 1 # cas de base
    else:
        return L[0] * produit(L[1:])
```

```
>>> produit([1, 4, 3])
12
```

Types de récursivité

Récursivité simple et multiple

Définition

- Une fonction récursive est dite *simple* si elle ne contient au plus qu'un seul appel récursif (éventuellement un par sous-cas qu'elle traite).

Récursivité simple et multiple

Définition

- Une fonction récursive est dite *simple* si elle ne contient au plus qu'un seul appel récursif (éventuellement un par sous-cas qu'elle traite).
- Dans le cas contraire, elle est dite *multiple*.

Exemple de récursivité multiple

On considère la suite de Fibonnaci définie par :

$$u_0 = a, \quad u_1 = b, \quad \forall n \in \mathbb{N}, \quad u_{n+2} = u_{n+1} + u_n.$$

Exemple de récursivité multiple

On considère la suite de Fibonnaci définie par :

$$u_0 = a, \quad u_1 = b, \quad \forall n \in \mathbb{N}, \quad u_{n+2} = u_{n+1} + u_n.$$

- Il y a 2 cas de bases.

Exemple de récursivité multiple

On considère la suite de Fibonnaci définie par :

$$u_0 = a, \quad u_1 = b, \quad \forall n \in \mathbb{N}, \quad u_{n+2} = u_{n+1} + u_n.$$

- Il y a 2 cas de bases.
- Le cas général se divise en deux sous cas donnés par la formule de récurrence.

Exemple de récursivité multiple

On considère la suite de Fibonnaci définie par :

$$u_0 = a, \quad u_1 = b, \quad \forall n \in \mathbb{N}, \quad u_{n+2} = u_{n+1} + u_n.$$

- Il y a 2 cas de bases.
- Le cas général se divise en deux sous cas donnés par la formule de récurrence.

```
def fibonacci(a:int, b:int, n : int) -> int:
    """ renvoie le terme d'indice n de la suite de fibonacci """
    if n == 0:
        return a
    elif n == 1:
        return b
    else :
        return fibonacci(a, b, n-1) + fibonacci(a, b, n-2)
```

Mise en garde concernant les récursivités multiples

Inefficacité

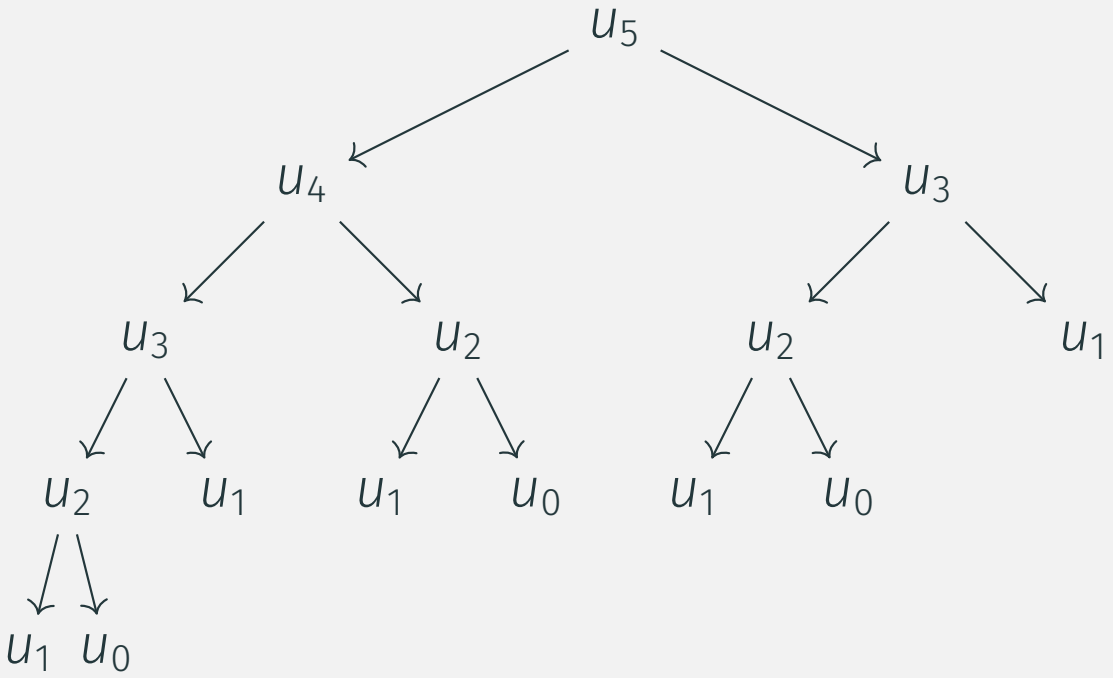
Fonctions récursives multiples : code souvent plus simple, mais parfois très inefficace !

Mise en garde concernant les récursivités multiples

Inefficacité

Fonctions récursives multiples : code souvent plus simple, mais parfois très inefficace !

- Exemple de `fibonacci(5)` :

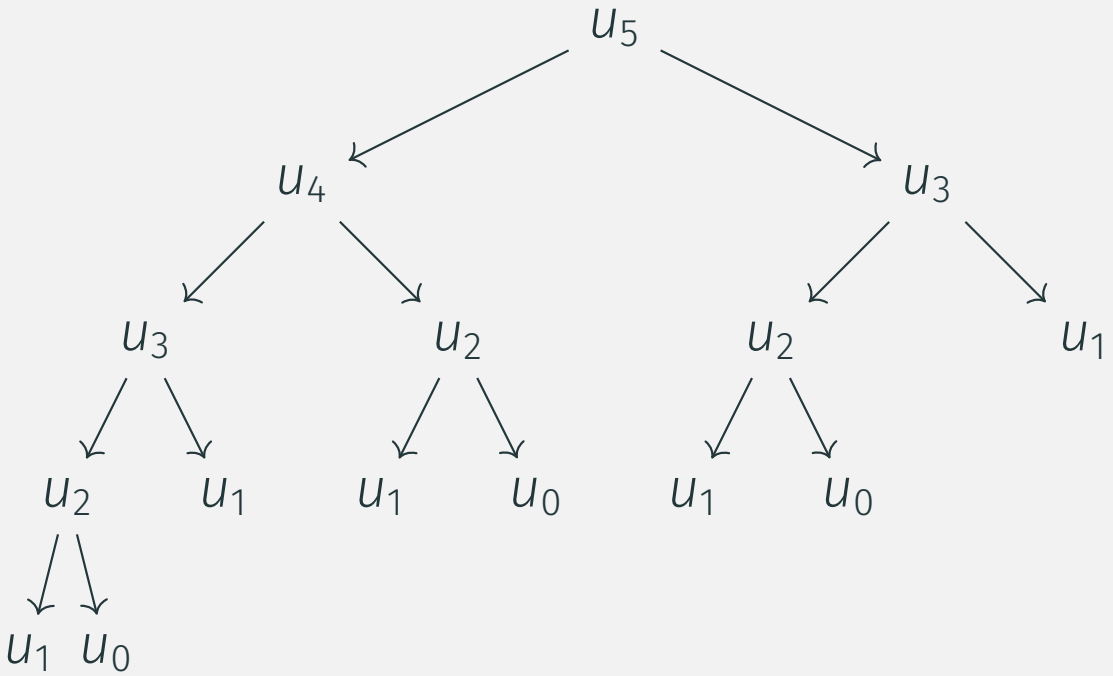


Mise en garde concernant les récursivités multiples

Inefficacité

Fonctions récursives multiples : code souvent plus simple, mais parfois très inefficace !

- Exemple de `fibonacci(5)` :



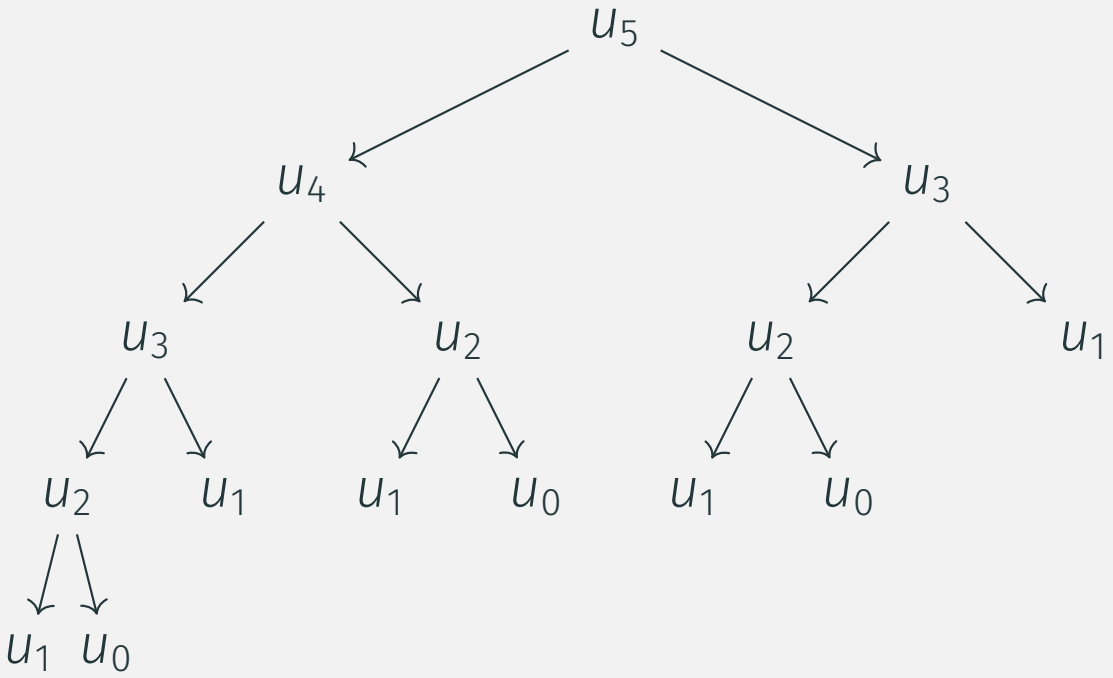
u_1 par exemple a été évalué 5 fois

Mise en garde concernant les récursivités multiples

Inefficacité

Fonctions récursives multiples : code souvent plus simple, mais parfois très inefficace !

- Exemple de `fibonacci(5)` :



u_1 **par exemple a été évalué** 5 **fois**

- Contournement possible par programmation dynamique (2ème année).

Récursivité mutuelle (ou croisée)

Définition
On dit que deux fonctions récursives sont *mutuellement récursives* si l'une fait appel à l'autre et vice versa. On parle alors de *récursivité croisée*.

Exercice

```python
def pair(n:int)->int:
    if n == 0:
        return True
    else:
        return \
        ↪ impair(n-1)
```

```python
def impair(n:int)->int:
    if n == 0:
        return False
    else:
        return pair(n-1)
```

Au stylo, déterminer le résultat de l'appel `pair(3)`, et `pair(2)`.

Solution

Pour le premier appel :

- comme $3 \neq 0$, on calcule `impair(2)`,
- comme $2 \neq 0$, on calcule `pair(1)`,
- comme $1 \neq 0$, on calcule `impair(0)`,
- C'est un cas terminal, la fonction renverra `False`.

Solution

Pour le premier appel :

- comme $3 \neq 0$, on calcule `impair(2)`,
- comme $2 \neq 0$, on calcule `pair(1)`,
- comme $1 \neq 0$, on calcule `impair(0)`,
- C'est un cas terminal, la fonction renverra **`False`**.

Pour le second :

- comme $2 \neq 0$, on calcule `impair(1)`,
- comme $1 \neq 0$, on calcule `pair(0)`.
- C'est un cas terminal, la fonction renverra **`True`**.

Exercice

On considère les suites (u_n) et (v_n) définies par :

$$u_0 = a, \quad v_0 = b, \quad \begin{cases} u_{n+1} = u_n - v_n \\ v_{n+1} = u_n + v_n. \end{cases}$$

On suppose deux variables `a, b` fixées en début de programme.

1. Écrire deux fonctions récursives croisées `u(n:int)->int` et `v(n:int)->int` qui permettent de calculer respectivement u_n et v_n.
2. Écrire une fonction `uv(n:int)->(int, int)`, impérative cette fois, renvoyant la valeur de (u_n, v_n).

Solution

- Versions récursives

```
def u(n:int)->int:
    if n == 0:
        return a
    else :
        return u(n-1)-v(n-1)
```

```
def v(n:int)->int:
    if n == 0:
        return b
    else:
        return u(n-1)+v(n-1)
```

Solution

- On utilise ici des boucles classiques.

```
def uv(n:int)->(int, int):
    u, v = a, b
    for _ in range(1, n+1):
        u, v = u-v, u+v
    return u, v
```

Solution

- Les résultats sont cohérents :

```
>>> a, b = 2, 3
>>> u(5), v(5)
(4, -20)
>>> uv(5)
(4, -20)
```