Méthodologie de programmation

Session 5

Alex Singh

Les listes sont récursives !

- Jusqu’à présent, nous avons manipulé des listes en les parcourant de manière itérative.

Les listes sont récursives !

- Jusqu'à présent, nous avons manipulé des listes en les parcourant de manière itérative.
- Mais il existe une autre façon de les traiter, basée sur leur définition suivante:

> Une list est soit :
> - Vide.
> - Un **premier** élément suivi du **reste** de la liste.

Les listes sont récursives !

- Jusqu'à présent, nous avons manipulé des listes en les parcourant de manière itérative.
- Mais il existe une autre façon de les traiter, basée sur leur définition suivante:

Une **liste** est soit :
- Vide.
- Un élément suivi d'une **liste**.

D'une simplicité frustrante !

Raisonnement impératif et fonctionnel

Problème

Calculer la somme des éléments d'une liste donnée.

Raisonnement impératif et fonctionnel

Problème

Calculer la somme des éléments d'une liste donnée.

Algorithme itératif

Idée : Parcourir la liste à l'aide d'une boucle, calculer et conserver les sommes partielles sur une variable que nous mettons à jour à chaque étape de la boucle, retourner cette variable.

Raisonnement impératif et fonctionnel

Problème

Calculer la somme des éléments d'une liste donnée.

Algorithme itératif

Idée : Parcourir la liste à l'aide d'une boucle, calculer et conserver les sommes partielles sur une variable que nous mettons à jour à chaque étape de la boucle, retourner cette variable.

Algorithme récursif

Idée :

- **Cas de base**: Pour les listes vides, la somme est de 0.
- **Récursion**: La somme des éléments d'une liste non-vide est le premier élément plus la somme des éléments du reste de la liste.

Raisonnement impératif et fonctionnel

Problème

Calculez la somme des éléments d'une liste donnée.

Raisonnement impératif et fonctionnel

Problème

Calculez la somme des éléments d'une liste donnée.

Pseudocode

```
def somme(l):
  sommePartielle = 0
  for i from 0 to length(l):
    sommePartielle += l[i]
  return sommePartielle
```

Raisonnement impératif et fonctionnel

Problème

Calculez la somme des éléments d'une liste donnée.

Pseudocode

```
def somme(l):
  sommePartielle = 0
  for i from 0 to length(l):
    sommePartielle += l[i]
  return sommePartielle
```

Pseudocode

```
def somme(l):
  if is-empty(l):
    0
  else:
    head(l)+somme(tail(l))
```

Les nombres naturels sont récursifs eux aussi !

- Comment peut-on définir les nombres naturels ?

Les nombres naturels sont récursifs eux aussi !

- Comment peut-on définir les nombres naturels ?

Un nombre naturel est soit :

- Zero.
- Le **successeur** d'un autre nombre naturel, c'est-à-dire $n + 1$ pour un nombre naturel n.

Les nombres naturels sont récursifs eux aussi !

- Comment peut-on définir les nombres naturels ?

Un nombre naturel est soit :
- Zero.
- Le **successeur** d'un autre nombre naturel, c'est-à-dire $n + 1$ pour un nombre naturel n.

Un **nombre naturel** est soit :
- Zéro.
- Un **nombre naturel** "plus un".

Les nombres naturels sont récursifs eux aussi !

- Comment peut-on définir les nombres naturels ?

Un **nombre naturel** est soit :

- Zéro, dont le **prédécesseur** est zéro.
- Le **prédécesseur** $n - 1$ d'un nombre naturel n.

Les nombres naturels sont récursifs eux aussi !

- Comment peut-on définir les nombres naturels ?

Un **nombre naturel** est soit :

- Zéro, dont le **prédécesseur** est zéro.
- Le **prédécesseur** $n - 1$ d'un nombre naturel n.

Un **nombre naturel** est soit :

- Zéro, dont le **prédécesseur** est zéro.
- Un **nombre naturel** (non-null) “moins un”.

Raisonnement impératif et fonctionnel, vol. 2.

Problème

Calculer $n!$, c'est-à-dire le produit:

$$\prod_{i=1}^{n} = 1 \times 2 \times 3 \times 4 \times \ldots \times n.$$

Raisonnement impératif et fonctionnel, vol. 2.

Problème

Calculer $n!$, c'est-à-dire le produit:

$$\prod_{i=1}^{n} = 1 \times 2 \times 3 \times 4 \times \ldots \times n.$$

Algorithme itératif
Mise en place d'une boucle, calcul des produits partiels.

Raisonnement impératif et fonctionnel, vol. 2.

Problème

Calculer $n!$, c'est-à-dire le produit:

$$\prod_{i=1}^{n} = 1 \times 2 \times 3 \times 4 \times \ldots \times n.$$

Algorithme itératif
Mise en place d'une boucle, calcul des produits partiels.

Algorithme récursif
Utiliser $0! = 1, n! = n \times (n-1)!$.

Plus sur la récursivité

- Une méthode puissante qui peut remplacer complètement l'itération.

Plus sur la récursivité

- Une méthode puissante qui peut remplacer complètement l'itération.
- Plus facile ou plus difficile à raisonner, selon le contexte: certains problèmes sont naturellement traités de manière récursive, d'autres de manière itérative.

Plus sur la récursivité

- Une méthode puissante qui peut remplacer complètement l'itération.
- Plus facile ou plus difficile à raisonner, selon le contexte: certains problèmes sont naturellement traités de manière récursive, d'autres de manière itérative.
- Doit être mis en place avec soin pour être efficace (récursion terminale, mémoïsation).

Plus sur la récursivité

- Une méthode puissante qui peut remplacer complètement l'itération.
- Plus facile ou plus difficile à raisonner, selon le contexte: certains problèmes sont naturellement traités de manière récursive, d'autres de manière itérative.
- Doit être mis en place avec soin pour être efficace (récursion terminale, mémoïsation).
- Si votre structure de données ou votre problème a une définition récursive (une décomposition en instances **plus petites** de elle/lui-même), pensez à la récursivité !

Plus sur la récursivité

- Une méthode puissante qui peut remplacer complètement l'itération.
- Plus facile ou plus difficile à raisonner, selon le contexte: certains problèmes sont naturellement traités de manière récursive, d'autres de manière itérative.
- Doit être mis en place avec soin pour être efficace (récursion terminale, mémoïsation).
- Si votre structure de données ou votre problème a une définition récursive (une décomposition en instances **plus petites** de elle/lui-même), pensez à la récursivité !