

Algorithmique et Programmation 2

*Transparents réalisés avec **Quarto** et **Revealjs**.*

Anthony Perez

Partie I – Récursivité

Rappel

```
1 Fonction f(n):
2     Si (n==0) Alors
3         retourner 0
4     Sinon si (n==1) Alors
5         retourner 1
6     Sinon
7         retourner n + f(n-1)
8     FinSi
9
10 afficher(f(10))
11 afficher(f(100))
12 afficher(f(1000))
```

55
5050
500500

... frustrant ?

```
1 print(fonction(10000))  
-----  
RecursionError Traceback (most recent call last)  
Cell In[2], line 1  
----> 1 print(fonction(10000))  
  
Cell In[1], line 7, in fonction(n)  
    5     return 1  
    6 else:  
----> 7     return n + fonction(n-1)  
  
Cell In[1], line 7, in fonction(n)  
    5     return 1  
    6 else:  
----> 7     return n + fonction(n-1)  
  
[... skipping similar frames: fonction at line 7 (2968 times)]  
  
Cell In[1], line 7, in fonction(n)  
    5     return 1  
    6 else:  
----> 7     return n + fonction(n-1)  
  
Cell In[1], line 2, in fonction(n)  
    1 def fonction(n):  
----> 2     if(n==0):  
    3         return 0  
    4     if(n==1):
```

Une fausse limitation...

```
1 import sys  
2 print(sys.getrecursionlimit())
```

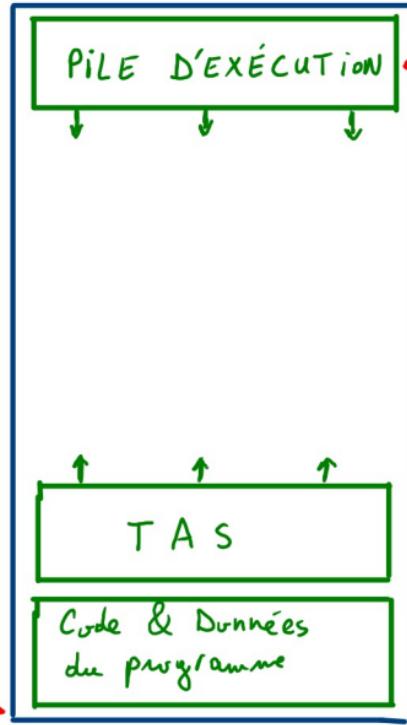
3000

```
1 sys.setrecursionlimit(10005)
```

 ... mais une limitation raisonnable !

Détour par la mémoire

Mécanisme de **mémoire virtuelle** (dédiée à un unique programme).



Note

- code et données du programme de l'utilisateur
- **tas** : variation dynamique de l'occupation mémoire, peut contenir des variables du programme
- **pile d'exécution** : (essentiellement) appels de fonction pendant l'exécution du programme

Fonction en appelant une autre

```
1 Fonction P():
2 ...
3   Q()
4 ...
5
6 Fonction Q():
7 ...
8   Retourner 2025
```



Note

- Les fonctions **P** et **Q** ont besoin d'allouer de la mémoire pour leurs variables locales ou d'initier l'appel à une autre fonction.
- Dès l'appel de **Q**, la fonction **P** est suspendue temporairement.
- Lorsque **Q** retourne sa valeur, la mémoire est libérée et **P** reprend son exécution.

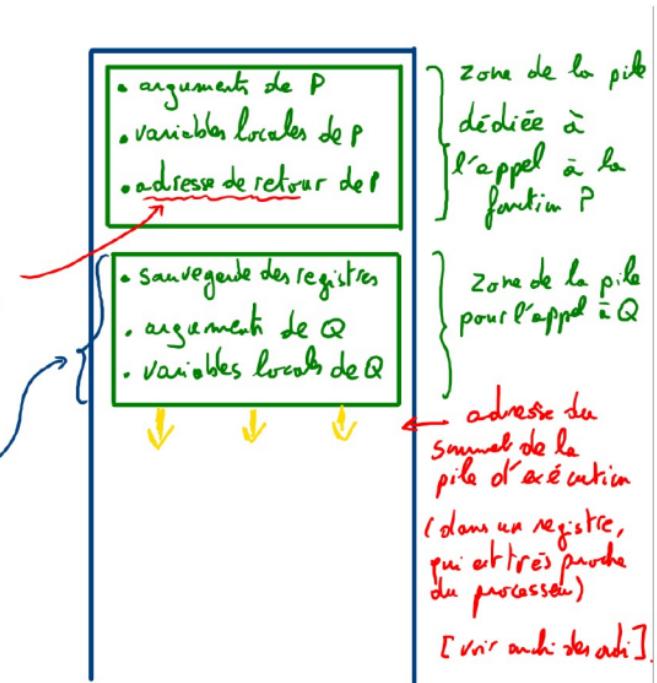
Gestion par la **pile d'exécution**

Zoom sur la pile d'exécution

Zoom sur le fonctionnement de la pile d'exécution.

utile pour reprendre l'exécution de P une fois que Q aura terminé.

Lorsque Q termine,
toute cette zone est
libérée de la pile



Utilité



Tip

- Simplicité du code (parfois)
- Facilite les preuves de terminaison/correction ([CM6](#))
- Programmation fonctionnelle : paradigme (en partie) basé sur la récursivité



Les appels récursifs ont un coût (la pile d'exécution) !

Récursivité terminale

Définition

- Une fonction est **récursive terminale** si l'appel récursif est la dernière instruction évaluée lors de son exécution.

Placer l'appel récursif en dernière ligne de la fonction ne suffit pas !

- La fonction n'effectue aucune autre opération après l'appel récursif...
- ... ni même aucun calcul avec le résultat obtenu de l'appel récursif.

recursive_terminale

```
1 Fonction P(x) {  
2     ...  
3     ...  
4     ...  
5     retourner P(x-1)  
6 }
```

recursive_non_terminale

```
1 Fonction Q(x) {  
2     ...  
3     ...  
4     ...  
5     retourner x + Q(x-1)  
6 }
```

Somme des n premiers entiers

```
somme_recursive_non_terminale
```

```
1 Fonction f(n) {  
2     Si (n==0) Alors  
3         retourner 0  
4     Sinon si (n==1) Alors  
5         retourner 1  
6     Sinon  
7         retourner n + f(n-1)  
8     FinSi  
9 }
```



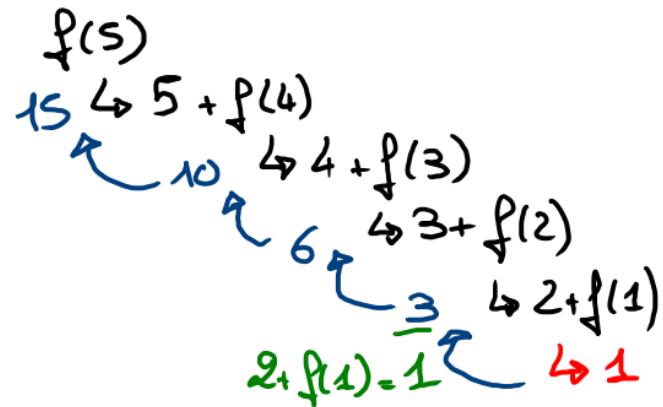
Utilisation d'un accumulateur comme paramètre supplémentaire

```
somme_recursive_terminale
```

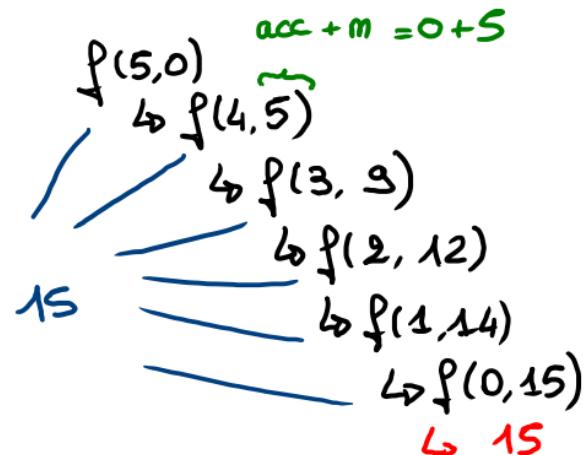
```
1 Fonction f(n, acc=0) {  
2     Si (n>0) Alors  
3         retourner f(n-1, acc+n)  
4     Sinon  
5         retourner acc
```

Traces des exécutions

```
somme_recursive_non_terminale
1 Fonction f(n) {
2     Si (n==0) Alors
3         retourner 0
4     Sinon si (n==1) Alors
5         retourner 1
6     Sinon
7         retourner n + f(n-1)
8     FinSi
9 }
```



```
somme_recursive_terminale
1 Fonction f(n, acc=0) {
2     Si (n>0) Alors
3         retourner f(n-1, acc+n)
4     Sinon
5         retourner acc
```



Cas d'étude – La suite de Fibonacci

(i) Définition

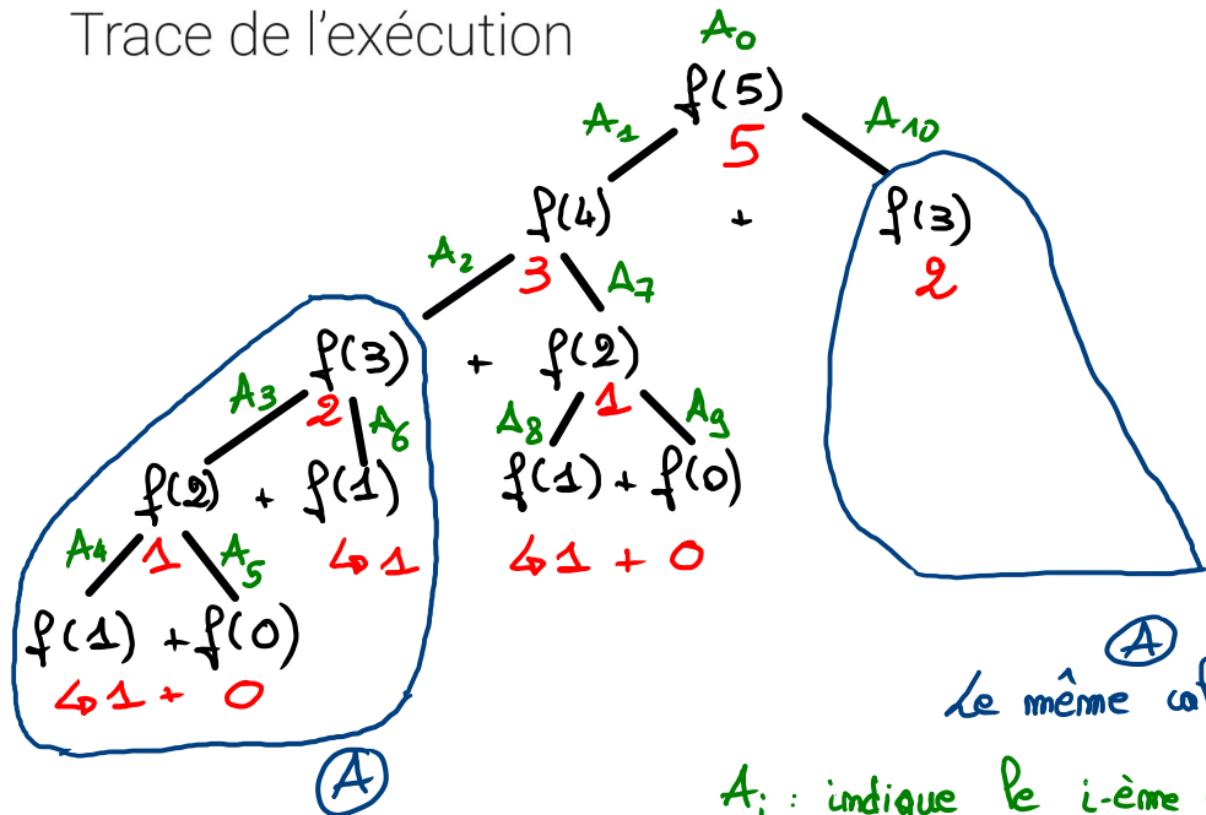
$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \text{ si } n > 1$$

```
1 Fonction fibonacci(n)
2     Si (n==0) Alors
3         retourner 0
4     Si (n==1) Alors
5         retourner 1
6     retourner fibonacci(n-1) + fibonacci(n-2)
7
8 Afficher(fibonacci(10))
```

Trace de l'exécution



\textcircled{A}
Le même calcul est fait.

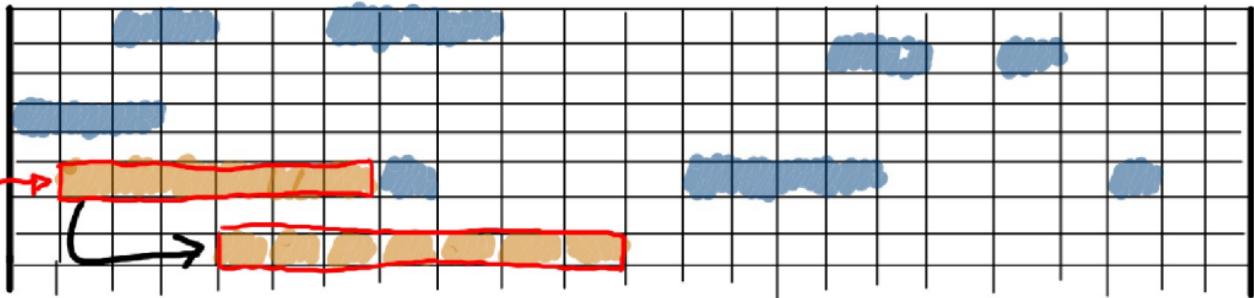
A_i : indique le i -ème appel récursif.

Une fatalité ?

```
1 Fonction fibonacci_iterative(n):
2   Si (n<=1) Alors
3     retourner 1
4   premier_terme = 0
5   second_terme = 1
6   resultat = premier_terme + second_terme
7   Pour i allant de 3 à n+1 Faire
8     premier_terme = second_terme
9     second_terme = resultat
10    resultat = premier_terme + second_terme
11  retourner resultat
12
13 Afficher(fibonacci_iterative(10))
```

Partie II – Listes chaînées

Retour sur les tableaux



un tableau est stocké dans des cellules contiguës de la mémoire

Dessin © Mathieu Liedloff

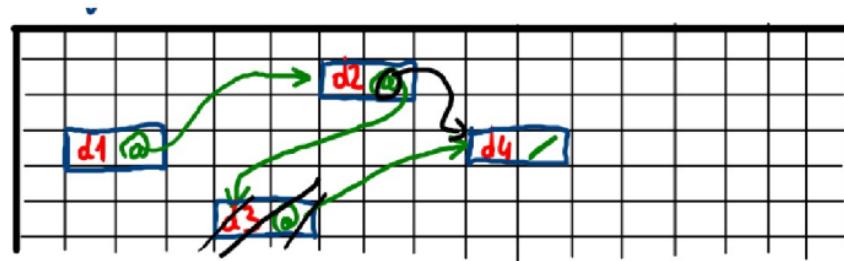
i Note

- L'ajout d'un élément peut nécessiter de déplacer tout le tableau
- La répétition d'opérations de type ajout/suppression peut devenir coûteuse

Description

Liste chaînée

- Les objets ont une **référence** vers l'objet suivant
- Les objets sont dispersés dans la mémoire
- L'ajout d'une donnée ne nécessite pas de tout déplacer
- La suppression d'une donnée peut être rapide



On stocke les données d_1, d_2, d_3, d_4

Les classes

```
1 class Maillon:
2     def __init__(self, valeur=None):
3         self.valeur = valeur
4         self.suivant = None
5
6     def __str__(self):
7         return str(self.valeur)
8
9 class ListeChaine:
10    def __init__(self):
11        self.tete = None
12
13    def ajouter_debut(self, valeur):
14        maillon = Maillon(valeur)
15        ...
16
17    def supprimer_fin(self):
18        ...
```



Remarque

- Les classes existent dans de nombreux langages de programmation mais par souci de simplicité, on ne les écrira pas en pseudo-code.

Gain par rapport aux tableaux

```
1 def test(F,n=1000):
2     import random
3     for i in range(n):
4         F.ajouter_debut(random.randint(1,100))
5     for i in range(n):
6         F.supprimer_fin()
```



Remarque

- `F` est un objet supportant les méthodes `ajouter_debut` et `supprimer_fin`

```
1 class Tableau:
2     def __init__(self):
3         self.T = list()
4
5     def ajouter_debut(self, valeur):
6         self.T.insert(0,valeur)
7
8     def supprimer_fin(self):
9         self.T.pop()
```



Temps d'exécution tableau : 0.22906902898103

Gain par rapport aux tableaux

```
1 class Maillon:
2     def __init__(self, valeur=None):
3         self.valeur = valeur
4         self.suivant = None
5
6     def __str__(self):
7         return str(self.valeur)
8
9 class ListeChaine:
10    def __init__(self):
11        self.tete = None
12
13    ...
```

Temps d'exécution tableau : 0.22906902898103

Temps d'exécution liste chaînée : 6.4146325619658455

! Le gain en temps n'est pas flagrant...

- Une idée de la raison ?
- **La liste est entièrement parcourue à chaque suppression !**

Enrichir la structure

Une solution

- Ajouter une référence sur le dernier objet de la liste
- Conserver une référence vers le maillon précédent

```
1 class Maillon:  
2     def __init__(self):  
3         self.precedent = None  
4  
5 class ListeChainee:  
6     def __init__(self):  
7         self.tete = None  
8         self.queue = None  
9     ...
```



Sur $n = 10000$ éléments

Temps d'exécution tableau : 1.9569160589999228
Temps d'exécution liste chaînée : 1.2926261479997265

Manipuler une liste chaînée

Comment l'afficher ?

```
1 # Affichage depuis n'importe quel maillon
2 Fonction afficher(maillon) {
3     s = ""
4     Si maillon != None Alors
5         maillon_courant = maillon
6         Tant que maillon_courant.suivant != None Faire
7             # On suppose que afficher_maillon(...) retourne une chaîne de caractères
8             s = s + afficher_maillon(maillon_courant) + " -> "
9             maillon_courant = maillon_courant.suivant
10            s = s + afficher_maillon(maillon_courant)
11        retourner s
12 }
```

```
1 import random
2 L = ListeChaine()
3 for _ in range(10):
4     L.ajouter_fin(random.randrange(1, 20))
5 print(L)
```

```
1 -> 14 -> 17 -> 3 -> 1 -> 15 -> 12 -> 6 -> 8 -> 1
```

Une solution plus élégante

```
1 Fonction afficher(maillon) {
2     Si (maillon.suivant != None) Alors
3         retourner afficher_maillon(maillon) + " -> " + afficher(maillon.suivant)
4     retourner afficher_maillon(maillon)
5 }
6
7 class ListeChaine:
8     ...
9     def __str__(self):
10        if self.tete is not None:
11            return afficher(self.tete)
12        return ""
13    ...
14
15 -> 11 -> 12 -> 11 -> 11 -> 19 -> 7 -> 17 -> 19 -> 8
```

La suite en TD

Ajout, suppression, recherche, ...

- En version itérative...
- ... et récursive !

That's all folks!