Bases de la récursivité

C. Thomas

24 février 2022

Résumé

Ce document présente les bases de la récursivité de façon (je l'espère) simple et compréhensible pour tout le monde le langage utilisé pour les fonction sera le python, version 3.8

Table des matières

Table des figures

1 Qu'est ce que la récursivité ?

Définition 1.1 La *récursivité* est une technique d'écriture des **fonctions** et de la manière dont elles sont interprétées, elles sont assez proches des boucles en cela qu'elles éxecutent un programme et ont des conditions de sorties (proche donc des boucles *while*).

Par exemple prenons l'application $(!) \in \mathbb{N}^{\mathbb{N}}$ qui $\forall n \in \mathbb{N}^*, n \mapsto !n = \prod_{k=1}^{n} k$ et $0! = 1$ voici une manière classique de l'implémenter

```
def fact(n):
  result = 1
  for i in range(1,n):
    result = result*i
  return result
```

FIGURE 1 – Implémentation de l'application factorielle de manière "classique"

la boucle *for* est une boucle inconditionnelle, on remarque assez vite que l'algorithme ci présent donne bien la factorielle de n en $\mathcal{O}(n)$ en temps et $\mathcal{O}(1)$ en espace. Les conditions de sorties qu'on peut en tirer sont ce qu'on appelle les *cas de base* et ici on peut voir avec la ligne $result = 1$ qu'on a le cas de base $0! = 1$. On s'arrête une fois que i a atteint la valeur $n - 1$ (comprise) on peut donc naïvement la convertir en une fonction *récursive* en prenant les cas de bases :

```
def fact(n):
  if n==0:
    return 1
  else:
    return n*(fact(n-1))
```

FIGURE 2 – Implémentation naïve de l'application factorielle de manière récursive

On remarque que l'écriture récursive est **plus compacte** et **plus explicite** que l'écriture dites "classique" de la fonction : en effet l'écriture récursive est souvent **proche de l'écriture mathématique** ce qui la rends plus compréhensible mais, à quel prix ? On se rend vite compte que la complexité temporelle est toujours en $\mathcal{O}(n)$ cependant l'ordinateur doit sauvegarder n calculs intermédiaires (n.b ils sont sauvegardé dans ce qu'on appelle la *stack* (ou pile en français)) on passe donc à $\mathcal{O}(n)$ en espace ! On en déduit la remarque-proposition suivante

Proposition 1.1 On préférera une écriture *classique* à une écriture *récursive*, le gain en lisibilité perds *en général* sur la perte de performance en espace.

Cependant cette perte de performance est résolvable à l'aide d'un autre paradigme étudié dans le chapitre suivant, *la récursivité terminale*

2 Récursivité Terminale

La proposition 1.1 est très générale, en effet il existe des situations où une réponse résursive vous viendra tout de suite à l'esprit et où vous ne trouverez pas de réponse "classique" lisible, dans la plupart des cas vous pouvez mettre directement cette fonction en réponse, les critère de complexité sont en général sur la complexité temporelle et non spatiale, cependant, si jamais vous devez vous trouver dans une situation où vous **devez** faire une réponse spatialement optimale alors le concept de *récursivité terminale* est là pour vous !

Remarque 2.1 Il est très important d'appuyer que dans la plupart des cas vous n'avez **pas** besoin de trouver une réponse en récursivité terminale elle est utile quand la complexité spatiale est exponentielle ou factorielle et que vous devez faire tourner le code. Rappelez vous de la première proposition aussi.

Le principe de la récursivité terminale est simple ce qui fais augmenter la complexité spatiale dans le cas de la récursivité c'est les calculs intermédiaires qui doivent être calculé et stocké à chaque fois, on a donc

Définition 2.1 Un programme est récursif terminal si et seulement si il ne fait pas appel à la pile d'appel, donc si il ne fait que faire appel à lui même et sans faire de calcul lors de l'éxécution

pour exemple on reprend le code de la figure 2 et on se propose de le rendre récursif terminal, la ligne qui cause les appels est $returni * fact(n-1)$ il faut donc faire ce calcul ailleurs, on fait régulièrement appel à deux concept une **fonction auxilliaire** et **un accumulateur** une fonction auxilliaire est une fonction définie localement dans une autre fonction et l'accumulateur est un *argument* de cette fonction. Il est ici important de noter que la fonction récursive en elle même est la fonction **auxillaire**.

```
def fact(n):
  def aux(accumulateur,i):
    if i==0:
      return accumulateur
    else:
      return aux(i*accumulateur,i-1)
  return aux(1,n)
```

FIGURE 3 – Implémentation récursive terminale de l'application factorielle

Il est aussi ici aisé de prouver la correction de l'algorithme, et le calcul à la fin est stocké dans *accumulateur* ce qui permet d'éviter d'appeler la pile, la fonction $fact$ est donc bien récursive terminale. On remarque ici bien la similarité avec la boucle while de la figure 1. Une analyse de complexité nous valide bien que la fonction est en $\mathcal{O}(n)$ en temps et en $\mathcal{O}(1)$ en espace.

Remarque 2.2 Il est à rappeler qu'une implémentation "classique" est à préférer même devant une implémentation récursive terminale même si les avantages ne sont plus qu'esthétique les candidats hors de l'option informatique sont en général bien plus à l'aise avec des boucles classique.

3 Exercices et entrainement

3.1 Exercice 1

Dans cet exercice on se propose d'étudier un classique, la suite de fibonacci.

1. Écrivez une fonction "classique" permettant d'obtenir le n-ième nombre de fibonacci en complexité spatiale linéaire
2. Écrivez une fonction "récursive" naïve faisant le même travail.
3. Enfin ecrivez une fonction permettant d'obtenir le n-ième nombre de fibonacci en complexité temporelle constante, quelle technique avez vous utilisée ?

3.2 Exercice 2

On se propose ici d'observer plusieurs codes et de déterminer s'il s'agit ou non d'une fonction résursive, résursive terminale, ou "classique" et d'en déterminer les complexitées

```
def multiplication(a,b):
  for i in range(b):
    a = a+a
  return a
```

FIGURE 4 – Code A

```
def fast_power(x,n):
  if n==1:
    return x
  elif n%2==0:
    return fast_power(x*x, n//2)
  else:
    return x*fast_power(x*x, (n-1)//2)
```

FIGURE 5 – Code B

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

FIGURE 6 – Code C