3 et 5 oct. 2018 | # Récursivité | I2 — Promo 122

A. Découverte d'une fonction récursive au travers d'un exemple

1. Premier exemple

On considère la suite (U_n) suivante, qui calcule une approximation de $\sqrt{3}$

$$U_0 = 2 \quad \text{et} \quad \forall n \in \mathbb{N}, U_{n+1} = \frac{1}{2}\left(U_n + \frac{3}{U_n}\right)$$

Pour calculer le n-ème terme de cette suite, on peut définir une fonction qui s'appelle elle même de la façon suivante :

```
1  def u(n):
2      if n==0:
3          return 2
4      else:
5          x=u(n-1)
6          return 0.5*(x+3/x)
7
8  #test de la fonction:
9  n=int(input("entrez la valeur de n"))
10 print("u(n) pour n=",n,"vaut:",u(n))
```

Exercice 1 Pour n=0,1,2,3 puis 4, compter le nombre d'appels total à la fonction u.

Exercice 2 Programmer la fonction u et vérifier les résultats de l'exercice 1.

2. Concevoir une fonction récursive

Exercice 3 Il vous est demandé de créer les fonctions ci-dessous sous python en utilisant le concept de fonction récursive (donc sans utiliser les fonctions déjà disponibles dans le module math par exemple) :

1. La fonction factorielle qui prend en argument un entier positif et qui renvoie $n!$.
2. La fonction puissance qui prend en argument un nombre x et un entier n et qui renvoie x^n.

3. Limites des fonctions récursives

On appelle suite de Syracuse une suite d'entiers naturels définie de la manière suivante : on part d'un nombre entier x_0 plus grand que zéro ; s'il est pair, on le divise par 2 ; s'il est impair, on le multiplie par 3 et on ajoute 1.

On remarque que, quelque soit x_0, cette suite arrive toujours à la valeur 1 au bout d'un certain temps et, qu'ensuite, la séquence 1, 4, 2 se répète indéfiniment. Cette conjecture n'a jamais été démontrée, mais aucun contre exemple n'a encore été trouvé.

Exercice 4 Ecrire la fonction convergence_syracuse ci-dessous et tester la avec $x_0 = 15$.

```
1  def convergence_syracuse(x0):
2      if x0==1 :
3      →   return 1
4      elif (x0%2)==0 :
5      →   return convergence_syracus(x0/2)
6      else :
7          return convergence_syracus(3*x0+1)
```

Exercice 5 Tester maintenant votre fonction avec

$$x_0 = 12345678912345678912345678912345678912345678.$$

Que se passe-t-il ? A quoi est due cette erreur ?

Cette limitation de la récursivité sous python (mais aussi sur les autres langages de programmation) peut être aussi mise en évidence sur factorielle(1000). Vous pouvez faire le test si vous le souhaitez.

Une manière de créer la fonction en contournant la récursivité est la suivante :

```
def convergence_syracuse2(x):
    i=0
    while(x !=1):
        i=i+1
        if x%2==0:
            x=x/2
        else :
            x=3*x+1
    return x
```

B. Complexité d'une fonction récursive

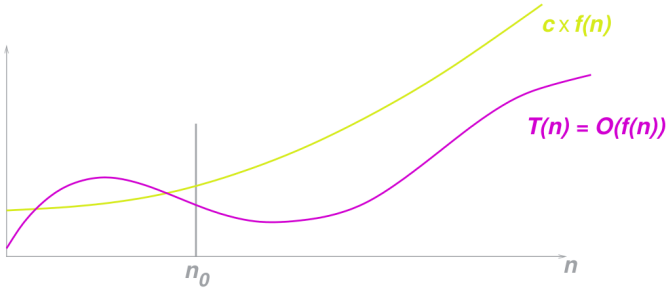
1. Rappels sur la complexité

a. Complexité

La complexité d'un algorithme est le nombre d'opérations élémentaires (échanges, comparaisons, produits, etc.) nécessaires à son exécution. Ce nombre s'exprime en fonction de la taille n des données traitées. On le note ici $T(n)$.

b. Notation

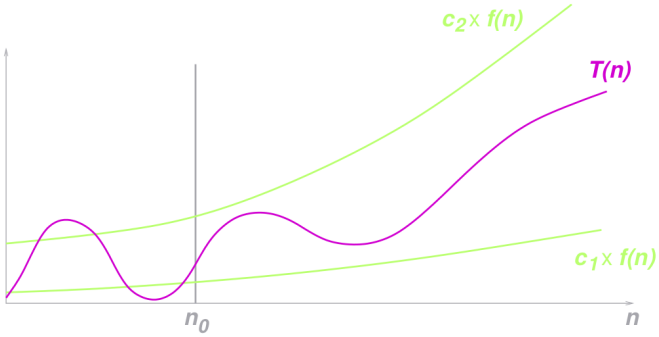
Lorsque l'on exprime la complexité d'un algorithme la valeur précise de $T(n)$ est inutile. La notation mathématique $T(n) = \mathcal{O}(f(n))$ est déjà plus adaptée. Elle signifie que $T(n)$ est majorée, à partir d'un certain rang n_0, par $c \times f(n)$.



L'inconvénient de la notation mathématique est qu'elle ne minore pas $T(n)$. En d'autres termes, une complexité en $\mathcal{O}(n)$ est aussi $\mathcal{O}(n^2)$ ou encore $\mathcal{O}(2^n)$.

En informatique, on se veut donc plus précis : $T(n) = \mathcal{O}(f(n))$ signifie qu'il existe des constantes c_1 et c_2 telles qu'à partir d'un certain rang n_0,

$$c_1 f(n) \leq T(n) \leq c_2 f(n)$$



Dans de nombreux cas, pour un même n fixé, le nombre $T(n)$ peut varier en fonction des données (comme dans le cas du tri basique ou le nombre de comparaisons est plus faible lorsque les données sont déjà "un peu triées"). Dans ce cas, on peut, par exemple, s'intéresser aux cas où les calculs sont les plus défavorables.

c. Classes de complexité

Voici les principales classes de complexité :

Type de complexité	Complexité en temps ...	Exemple :
$\mathcal{O}(1)$	... constant	permuter 2 éléments
$\mathcal{O}(\ln n)$	... logarithmique	recherche dichotomique
$\mathcal{O}(n)$	... linéaire	calcul de moyenne
$\mathcal{O}(n \ln n)$	... pseudo-linéaire	méthode de tri efficace (tri fusion)
$\mathcal{O}(n^k)$	.... polynomial	méthode de tri naïf (tri basique, à bulle)
$\mathcal{O}(2^n)$	... exponentiel	fonction récursive qui s'appelle plusieurs fois elle même

d. Quelques exercices

Exercice 6 Quelles sont les classes de complexités dans les exemples suivants ?

1. $T(n) = 2n \ln n + 8n + 4$
2. $T(n) = 10^2 - 7n + 2^n$

Exercice 7 Ecrire une fonction qui permute le premier et le dernier élément d'une liste. Calculer le nombre d'affectations. Quelle est la classe de complexité de la fonction ?

Exercice 8 Écrire une fonction qui recherche un élément A dans une liste L. Quelle est la complexité de cet algorithme ?

Exercice 9 Compter le nombre de comparaisons lorsque l'on trie un tableau de n éléments par la méthode du tri à bulles. Quelle est la classe de complexité ?

Exercice 10 Calcul approché de la solution de l'équation $x^3 - 3x^2 + 1 = 0$ comprise entre 0 et 2. On cherche une solution avec une précision de $1/2^k$ donnée.

1. Ecrire un algorithme itératif qui renvoie la solution en utilisant une boucle.
2. Ecrire un algorithme récursif qui renvoie la solution.
3. Quelle est la complexité de cet algorithme en fonction de $n = 2^k$?

2. Cas des fonctions récursives

Revenons sur l'exemple de la première partie :

```
1  def u(n):
2      if n==0:
3          return 2
4      else:
5          x=u(n-1)
6          return 0.5*(x+3/x)
7
8  #test de la fonction:
9  n=int(input("entrez la valeur de n"))
10 print("u(n) pour n=",n,"vaut:",u(n))
```

Exercice 11 Combien y-a-t-il d'appels dans le cas général pour le calcul de `u(n)` ?

Exercice 12 Evaluer le nombre d'opérations (addition, multiplication et division confondues) qu'effectue u(n) en fonction du paramètre n. En déduire le type de complexité de cette fonction.

On propose maintenant une programmation plus naïve de cette même fonction :

```
def u2(n):
    if n==0:
        return 2
    else:
        return 0.5*(u2(n-1)+3/(u2(n-1)))

#test de la fonction:
n=int(input("entrez la valeur de n"))
print("u(n) pour n=",n,"vaut:",u2(n))
```

Exercice 13 Pour n=0,1,2,3 puis 4, compter le nombre d'appel à la fonction u2.

Exercice 14 Combien y-a-t-il d'appels dans le cas général pour le calcul de u2(n) ?

Exercice 15 Programmer la fonction u2 et vérifier les résultats de l'exercice 13.

Exercice 16 Evaluer le nombre d'opérations (addition, multiplication et division confondues) qu'effectue u2(n) en fonction du paramètre n. En déduire le type de complexité de cette fonction. Comparer avec la complexité trouvée pour l'algorithme précédent. Conclure quant à l'importance du bon choix de l'algorithme.