Recursivite, Complexite spatiale, Terminaison et Recursivite Terminale corrige

October 23, 2020

1 Complexité spatiale

La complexité spatiale est analogue à la complexité temporelle, sauf qu'on évalue l'espace mémoire nécessaire à l'exécution de la fonction en fonction de la taille des données.

Prenons l'exemple de la fonction itérative de calcul de la puissance n-ième de x

[1]:
```
def puissance_iterative(x,n) :
    resultat = 1
    for i in range(n):
        resultat = x*resultat
    return resultat
```

Ce programme exécute une boucle de 0 à n-1 et affecte la nouvelle valeur de resultat. L'espace mémoire utilisé est juste 1 nombre stocké dans la variable resultat. On obtient donc une complexité spatiale C(n) = O(1), elle est indépendante de n.

Prenons maintenant l'exemple de la fonction récursive

[2]:
```
def puissance_recursive(x,n) :
    if n==0 :
        resultat = 1
        return resultat
    else :
        resultat = x*puissance_recursive(x,n-1)
        return resultat
```

Cette fois-ci un appel de la fonction puissance_recursive consiste à stocker une valeur de résultat. Pour un appel de la fonction, on a donc une complexité spatiale de O(1). Mais n appels récursifs sont stockés dans la pile d'appels lors de l'exécution de la fonction, en effet puissance(x,n) demande d'appeler puissance(x,n-1) puis puissance(x,n-2) puis ... jusqu'au critère d'arrêt puissance(x,0). Donc la complexité spatiale totale est de $C(n) = n \times O(1) = O(n)$, elle est linéaire.

Enfin pour la fonction récursive accélérée

[3]:
```
def puissance_rapide(x, n):
    if n == 0:
        return 1
    else:
        r = puissance_rapide(x, n // 2)
        if n % 2 == 0:
```

```
            return r * r
        else:
            return x * r * r
```

Encore une fois on stocke un nombre r pour un appel donc la complexité d'un appel est $O(1)$. Pour calculer le nombre d'appels on introduit k tel que $n = 2^k$ et on fait réalise les appels suivant puissance_rapide(x, $n = 2^k$), puis puissance_rapide(x, $n/2 = 2^{k-1}$), puis puissance_rapide(x, $n/4 = 2^{k-2}$), ... Donc on effectue $k = log_2(n)$ appels. La complexité spatiale de la fonction est donc $C(n) = log_2(n) \times O(1) = O(\ln n)$, elle est logarithmique.

1.0.1 exercice:

Calculer la complexité spatiale du programme récursif suivant:

[4]:
```
def u(n):
    if n == 0:
        return 2.
    else:
        return 0.5 * (u(n-1) + 3. / u(n-1))
```

A chaque appel n on doit stocker en mémoire les résultats des appels $n-1$ et $n-1$ d'où une complexité $C(n) = 2C(n-1)$ c'est à nouveau exponentielle $C(n) = 2^n$

1.0.2 exercice

Soit l une liste triée par ordre croissant et x un élément à chercher dans cette liste.

Reprendre le programme récursif qui prend en argument la liste triée l et l'élément à rechercher x et qui renvoie True si x est dans l, et False si x n'est pas dans l. On utilisera une méthode de recherche par dichotomie.

Puis calculer sa complexité spatiale.

[ ]:
```
def dichotomie(x,l):
    n = len(l)
    if n==0:
        return False
    elif x < l[n//2]:
        return dichotomie(x,l[0:n//2])
    elif x > l[n//2]:
        return dichotomie(x,l[n//2+1:n])
    else :
        return True
```

A chaque appel récursif n on a en mémoire la liste l de taille n et on fait appel à la fonction dichotomie avec en mémoire une liste de taille n/2, et cela jusqu' à n=1, on stocke donc en mémoire des listes de tailles :

$ C(n) = n + n/2 + n/4 + n/8 + \ldots + 1 = \sum{0}^{i\{max\}} = \log 2(n)\} 2^i = 2^{i\{max\}+1\} - 1 = 2n - 1 = O(n)$

2 Comparaison entre itératif et récursif

À travers les différents exemples rencontrés, on remarque que les programmes récursifs présentent l'avantage d'avoir une écriture plus concise que les programmes itératifs, car on n'a pas besoin d'utiliser de boucle.

On remarque également que dans un contexte où une relation de récurrence existe pour résoudre le problème, le programme récursif est plus simple à écrire que le programme itératif.

Par exemple écrire un programme itératif qui calcule les termes de la suite (u_n) telle que :

$u_0 = 2$

et

$u_n = \frac{1}{2}\left(u_{n-1} + \frac{3}{u_{n-1}}\right)$

est plus compliqué que d'utiliser un programme récursif.

Mais lors de l'écriture d'un programme récursif, il faut faire attention à la complexité.

La complexité temporelle du programme si on ne fait pas attention peut être élevée, par exemple elle peut être exponentielle selon l'écriture du programme qui calcule la suite (u_n). Certains cas particulier comme puissance_rapide où on divise la taille de l'argument en 2 à chaque appel peuvent présenter une complexité récursive inférieure à l'itérative correspondante.

La complexité spatiale est minimale pour un programme itératif, un programme récursif utilisera toujours plus d'espace mémoire que le programme itératif correspondant.

3 Terminaison

Lors de l'écriture d'un programme récursif, il faut faire attention à ce que le programme s'arrête un jour.

Prenons l'exemple de calcul de puissance :

[5]:
```
def puissance_rapide(x, n):
    if n == 0:
        return 1
    else:
        r = puissance_rapide(x, n // 2)
        if n % 2 == 0:
            return r * r
        else:
            return x * r * r
```

Identifiez la touche qui vous permet d'interrompre l'exécution de votre programme manuellement et testez ce programme pour des puissances non entières ou des puissances négatives.

Que ce passe-t-il, et pourquoi ?

On remarque que le programme ne s'arrête jamais ou qu'il affiche l'erreur "maximum recursion depth exceeded" lors des différents appels récursifs on atteint jamais le cas n=0, le programme continue à s'appeler lui même avec des valeurs de n négatives qui tendent vers $-\infty$

Il faut s'assurer lors de l'écriture d'un programme récursif qu'il possède un cas d'arrêt et que ce cas d'arrêt est atteint.

De manière plus formelle, la terminaison est assurée lorsque l'on peut identifier dans le programme un entier positif qui décroît à chaque appel récursif et qui prend une valeur définissant un cas d'arrêt.

4 Récursivité terminale

Il s'agit d'un type de programme récursif particulier qui ont pour objectif d'être plus rapide lors de l'exécution de la pile d'appels récursifs.

Les programmes récursifs écrits utilisent en général l'appel récursif dans une opération ou comme argument d'une fonction. Par exemple dans le programme suivant l'appel récursif intervient dans une multiplication

[6]:
```
def puissance_recursive(x,n) :
    if n==0 :
        resultat = 1
        return resultat
    else :
        resultat = x*puissance_recursive(x,n-1)
        return resultat
```

Une version récursive terminale de ce programme ne fait pas intervenir l'appel récursif dans aucune opération ni comme argument de fonction, ce qui donnerait :

[7]:
```
def puissance_recursive(x,n) :
    def puissance_recursive_terminale(x,k,intermediaire) :
        if k == 0 :
            return intermediaire
        else :
            return puissance_recursive_terminale(x,k-1,intermediaire*x)
    return puissance_recursive_terminale(x,n,1)
```

On remarque dans ce programme que puissance_recursive_terminale n'est appelé que dans les return. Les valeurs renvoyées par l'appel récursif ne sont pas utilisées dans des opérations ou en argument d'autre fonction.

L'opération de récurrence est cette fois-ci effectuée en argument de la fonction récursive et non l'inverse.

Si on détaille l'ordre des appels récursifs de la version terminale on a :

- puissance_récursive appelle puissance_recursive_terminale(x,n,1) donc demande : "combien vaut $x^n \times 1$?"
- puissance_recursive_terminale(x,n,1) ne répond pas mais appelle puissance_recursive_terminale(x,n-1,1*x), donc demande : "combien vaut $x^{n-1} \times (1 \times x)$?"
- puissance_recursive_terminale(x,n-1,x) ne répond pas mais appelle puissance_recursive_terminale(x,n-2,x*x), donc demande : "combien vaut $x^{n-2} \times (x \times x)$?"
- puissance_recursive_terminale(x,n-2,x^2) ne répond pas mais appelle puissance_recursive_terminale(x,n-3,x^2*x), donc demande : "combien vaut $x^{n-3} \times (x^2 \times x)$?"
- etc
- jusqu'à - puissance_recursive_terminale(x,n-(n-1),x^{n-1}) ne répond pas, mais appelle puissance_recursive_terminale(x,0,x^{n-1}*x), donc demande : "combien vaut $x^0 \times (x^{n-1} \times x)$?"

- qui répond directement la valeur de x^n sans avoir à dépiler la pile d'appels, car la variable intermédiare contient déjà le résultat voulu.

4.0.1 exercice:

- Ecrire un programme itératif qui prend en entrée une fonction f, un entier n et un élément x et qui calcule $f^n(x) = fofofo...of(x) = f(f(f(...f(x)...)))$ soit la composé n-ième de f évaluée en x.
- Ecrire une version récursive de ce programme.
- Ecrire une version récursive terminale de ce programme.

[1]:
```
def composee_iterative(f,n,x):
    resultat = x
    for i in range(n):
        resultat = f(resultat)
    return resultat
```

[8]:
```
import numpy as np

for i in range(10):
    print(composee_iterative(np.sin,i,3.141592/2))
```

```
1.570796
0.9999999999999466
0.8414709848078676
0.7456241416655387
0.6784304773607261
0.6275718320491482
0.5871809965734222
0.5540163907556223
0.5261070755028354
0.5021706762685499
```

[9]:
```
def composee_recursive(f,n,x):
    if n == 0:
        resultat = x
        return resultat
    else :
        resultat = f(composee_recursive(f,n-1,x))
        return resultat
```

[10]:
```
import numpy as np

for i in range(10):
    print(composee_recursive(np.sin,i,3.141592/2))
```

```
1.570796
0.9999999999999466
0.8414709848078676
0.7456241416655387
0.6784304773607261
0.6275718320491482
0.5871809965734222
0.5540163907556223
0.5261070755028354
0.5021706762685499
```

[11]:
```python
def composee_recursive_terminale(f,n,x):
    def composee_terminale(f,k,x,intermediaire) :
        if k == 0 :
            return intermediaire
        else :
            return composee_terminale(f,k-1,x,f(intermediaire))
    return composee_terminale(f,n,x,x)
```

[12]:
```python
import numpy as np

for i in range(10):
    print(composee_recursive_terminale(np.sin,i,3.141592/2))
```

```
1.570796
0.9999999999999466
0.8414709848078676
0.7456241416655387
0.6784304773607261
0.6275718320491482
0.5871809965734222
0.5540163907556223
0.5261070755028354
0.5021706762685499
```