INFO-H-100 - Informatique

Séance d'exercices 17
Introduction à Python
Complexité

Université Libre de Bruxelles
Faculté des Sciences Appliquées

20 novembre 2013

Notation grand O

Définition de O :

On dit d'une fonction $f(n)$ qu'elle "a une complexité en grand o de g", noté $f(n) \in \mathcal{O}(g(n))$, s'il existe un entier $N > 0$ et une constante $c > 0$ tels que pour tout entier $n > N$ nous avons $f(n) \leq c.g(n)$. Autrement dit, f est dominée par g, ou encore f ne croît pas plus vite que g (après un certain point).

Cette définition permet d'effectuer de nombreuses simplifications dans les calculs. En effet, de cette définition, il résulte que :

- Les facteurs constants ne sont pas importants. En effet, $T(n) = n$ ou si $T(n) = 100n$, $T(n)$ est toujours en $\mathcal{O}(n)$.
- Les termes d'ordres inférieurs sont négligeables. Ainsi, si $T(n) = n^3 + 4n^2 + 20n + 100$, on peut voir que pour $n > N = 5$
  - $n^3 > 4n^2$
  - $n^3 > 20n$
  - $n^3 > 100$

  De ce fait, $T(n)$ est en $\mathcal{O}(n^3)$

Notation grand O

$\mathcal{O}(1)$	Constant
$\mathcal{O}(n)$	Linéaire
$\mathcal{O}(log(n))$	Logarithmique
$\mathcal{O}(n.log(n))$	“ennloguenne“
$\mathcal{O}(n^2)$	Quadratique
$\mathcal{O}(n^3)$	Cubique

Complexité des opérations

Opération	**Types**	**Complexité**
+, -, *, /, //	nombres	$\mathcal{O}(1)$
a ** b	nombres	$\mathcal{O}(\log(b))$
a + b	string	$\mathcal{O}(len(a+b))$
return <expr>	-	$\mathcal{O}(expr)$
appel de fonction	-	$\mathcal{O}(1)$*
a = <expr>	-	$\mathcal{O}(expr)$

* : Pour l'appel lui-même ; il faut compter en plus la complexité de l'exécution de la fonction.

Complexité des opérations sur les listes

Opération	**Complexité maximale**
Copy	$\mathcal{O}(n)$
Append	$\mathcal{O}(n)$
Insert	$\mathcal{O}(n)$
Get item	$\mathcal{O}(1)$
Set item	$\mathcal{O}(n)$
Delete item	$\mathcal{O}(n)$
Iteration	$\mathcal{O}(n)$
Get slice	$\mathcal{O}(k)$
Del slice	$\mathcal{O}(n)$
Set slice	$\mathcal{O}(k+n)$
Extend	$\mathcal{O}(n+k)$
Sort	$\mathcal{O}(n \log(n))$
Multiply	$\mathcal{O}(nk)$
x in s	$\mathcal{O}(n)$
min(s), max(s)	$\mathcal{O}(n)$
len(s)	$\mathcal{O}(1)$

n est la longueur de la première liste, **k** celle de la seconde.

Complexité des opérations sur les dictionnaires

Opération	**Complexité maximale**
Copy	$\mathcal{O}(n)$
Get item	$\mathcal{O}(n)$
Set item	$\mathcal{O}(n)$
Delete item	$\mathcal{O}(n)$
Iteration	$\mathcal{O}(n)$

Règles de calcul en pratique

1. **Unité** : Une assignation, un `print`, un `return`, l'évaluation d'une expression ou d'une condition simple (ne donnant pas lieu à l'exécution d'une fonction, prends un temps fixé et est donc en $O(1)$.)

2. **Séquence** : Si un traitement 1 prend un temps $T_1(n)$ en $O(f_1(n))$ et un traitement 2 prends un temps $T_2(n)$ en $O(f_2(n))$ alors l'enchaînement de ces deux opérations prend $T_1(n) + T_2(n)$ et est en $O(f_1(n) + f_2(n)) = O(\max(f_1(n), f_2(n))$

Règles de calcul en pratique

3. `if` : Si `instruction_1` est en $O(f_1(n))$, `instruction_2` est en $O(f_2(n))$ et l'évaluation de `condition` est en $O(g(n))$ alors, suivant le test le `if` sera en $O(\max(f_1(n), g(n)))$ ou en $O(\max(f_2(n), g(n)))$ et peut être borné par $O(\max(f_1(n), f_2(n), g(n)))$

```
if condition:
    instruction_1
else:
    instruction_2
```

Règles de calcul en pratique

4. `while` : Sachant que le corps de la boucle est en $O(f_1(n))$ et que l'évaluation de la condition est en $O(f_2(n))$, si on a une fonction $O(g(n))$ qui donne une borne supérieure du nombre de fois que le corps sera exécuté, alors le `while` est en $O(f(n) * g(n))$ avec $f(n) = \max(f_1(n), f_2(n))$.

5. `for` : Une boucle `for` est à traiter de manière identique au `while` équivalent.

6. **Fonction :** L'appel à une fonction est en $\mathcal{O}(f(n))$ correspondant à la complexité du traitement de cette fonction pour les paramètres donnés.

Règles de calcul du grand O

Règle de sommation :
Soit f_1, f_2 deux fonctions telles que $f_1(n) \in O(g(n))$ et $f_2(n) \in O(h(n))$. Nous avons alors
$f_1(n) + f_2(n) \in \mathcal{O}(\max(g(n), h(n))$.

En code, cette situation se produit lorsque l'on a une séquence d'opérations.

Ce résultat signifie que la complexité d'une séquence d'opérations sera dominée par la plus longue opération de la séquence.

Exemple :

```
def regle_sommation(x):      # O(1)
    x = x + 1                # O(1)
    x = x // 2               # O(1)
    return x                 # O(1)
```

Règles de calcul du grand O

Règle des facteurs constants :
Si $f(n) \in O(g(n))$ alors $k.f(n) \in O(g(n))$ pour toute *constante* $k > 0$.

Cela signifie qu'une instruction se répétant un nombre constant de fois ne change pas sa complexité.

Exemple :

```
def regle_facteurs_constants(x):      # O(1)
    i = 0                             # O(1)
    while i < 3:                      # O(1) + O(1) + O(1) = O(1)
                                      ## Note : i ne depend pas de x
        x += 1                        # O(1)
        i += 1                        # O(1)
    return x                          # O(1)
```

Règles de calcul du grand O

Règle du produit :
Si $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$ alors
$f_1(n).f_2(n) \in O(g_1(n).g_2(n))$.

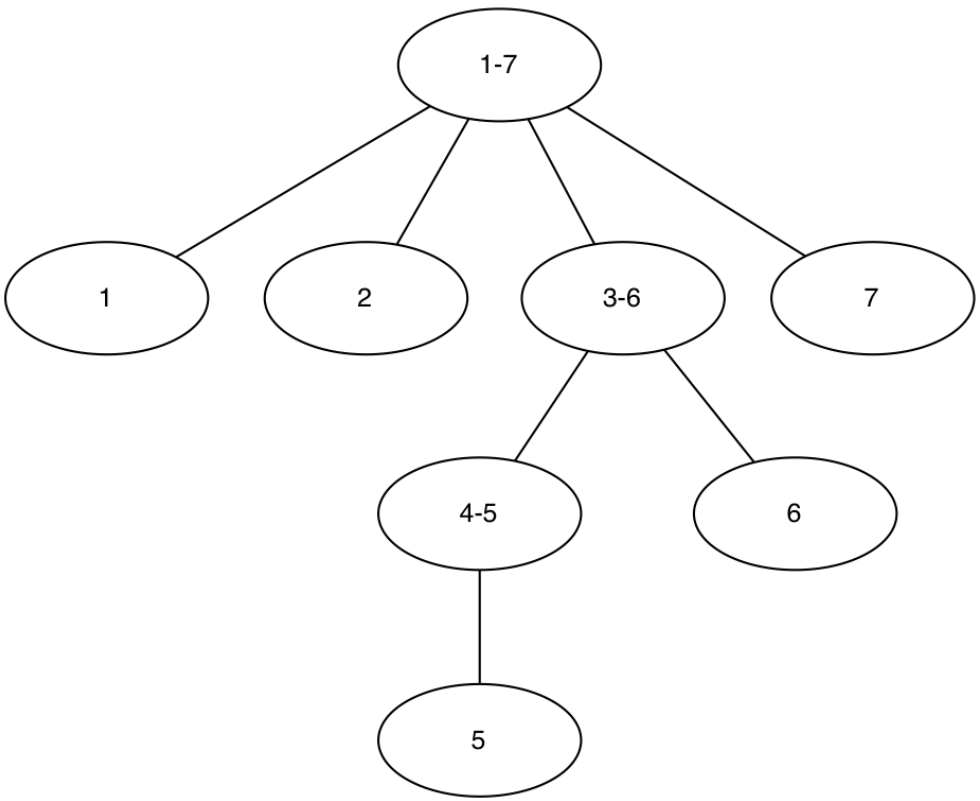
Dans du code, cette situation se produit par exemple dans le cadre des boucles imbriquées. Ce résultat signifie que l'imbrication (nesting) va augmenter la complexité du code.

Exemple :

```
def regle_produit(liste1,liste2):                  # O(n^2)
    result = 0                                     # O(1)
    for i in range(len(liste1)):                   # O(n.n) = O(n^2)
        for j in range(len(liste2)):               # O(n)
            result += liste1[i] * liste2[j]        # O(1)
    return result                                  # O(1)
```

Exemple

```
def minimum(liste,longueur_liste):      #0
    res = 0                             #1
    i = 1                               #2
    while i < longueur_liste:           #3
        if liste[i] < liste[res]:       #4
            res = i                     #5
        i = i + 1                       #6
    return res                          #7
```



Exemple

- (Règle 1) Les noeuds sans fils (les feuilles) de l'arbre, soit les noeuds 1,2,5,6 et 7 sont en $\mathcal{O}(1)$.
- (Règle 3) Le noeud 4-5 correspondant au `if` est en $\mathcal{O}(1)$.
- (Règle 2) La complexité de la séquence 4-6 est en $\mathcal{O}(1)$.
- (Règle 4) La complexité du `while` est en $\mathcal{O}(n)$.

Par la règle 2, la complexité de la totalité du traitement est en $\mathcal{O}(1 + 1 + n + 1)$ soit un $\mathcal{O}(n)$.