DIU « ENSEIGNER L'INFORMATIQUE AU LYCEE »

NOTIONS DE COMPLEXITE – EXERCICES SUPPLEMENTAIRES

1. Trouver le plus petit $n_0 \in \mathbb{N}$ de telle façon que pour tout $n \geq n_0$:
   a. $8n\log(n)$ est plus petit que $2n^2$
   b. 2^n est plus grand que n^4

a. $8n\log(n) < 2n^2 \Rightarrow 4\log(n) < n$. En essayant pour des n petits puissances de 2 :
$n = 4 : 8 > 4$
$n = 8 : 12 > 8$
$n = 16 : 16 \geq 16$
C'est vrai pour tout $n \geq n_0$ si $n_0 = 17$.

b. $2^n > n^4 \Rightarrow \log(2^n) > \log(n^4) \Rightarrow n > 4\log n$. Ce qui revient à a.

2. Donner une constante $x > 0$ et un entier $n \geq 1$ tels que pour tout $n \geq n_0$:
   a. $16\, n\log(n^2) \leq c \times n^2$
   b. $\frac{1}{10}2^n \geq c \times n^4$

a. $c = 32, n_0 = 1$
b. $c = 16, n_0 = 1/10$

3. Montrer que
   a. $2n^3 + 9n^2$ est $O(n^3)$
   b. $(n\log n)/8$ est $\Omega(n\log n)$
   c. $2^{n+2} - \mathrm{n}$ est $\Theta(2^n)$

a. En choisissant $c = 11$ et $n_0 = 1$, pour tout $n \geq n_0$, on a : $2n^3 + 9n^2 < 11n^3 = c \times n^3$
b. En choisissant $c = 1/9$ et $n_0 = 2$, pour tout $n \geq n_0$, on a : $\frac{1}{8}n\log n \geq \frac{1}{9}n\log n = c \times n\log n$
c.
Grand-O : en choisissant $c = 4$ et $n_0 = 1$, pour tout $n \geq n_0$, on a : $2^{n+2} - n = 4 \times 2^n - n < 4 \times 2^n = c \times 2^n$
Grand-Omega : en choisissant $c = 1$ et $n_0 = 1$, pour tout $n \geq n_0$, on a : $2^{n+2} - n = 4 \times 2^n - n > 2^n$ (cette inéqualité est vraie si $3 \times 2^n > n$ ce qui est vrai pour tout $n \geq 1$).
On a donc que $2^{n+2} - n$ est $\mathrm{O}(2^n)$ et $\Omega(2^n)$ donc est aussi $\Theta(2^n)$.

4. Prouver ou réfuter que « si $d(n)$ est $O(f(n))$, alors $a \times d(n)$ est $O(f(n))$ pour toute constante $a > 0$ ».

Supposons qu'il y a $c > 0$ et un $n_0 > 1$ tels que pour tout $n \geq n_0$: $d(n) \leq c \times f(n)$.
Nous devons prouver qu'il existe $c' > 0$ et un $n_0{}' > 1$ tels que pour tout $n \geq n_0{}'$: $a \times d(n) \leq c' \times f(n)$.
En choisissant $c' = a \times c$, alors nous avons pour tout $n \geq n_0$: $a \times d(n) \leq a \times c \times f(n) = c' \times f(n)$

5. Prouver ou réfuter que « si d(n) est O(f(n)) et e(n) est O(g(n)), alors d(n)-e(n) est O(f(n)-g(n))».

C'est faux, et nous pouvons donner un contre-exemple. Prenons $d(n) = 5n$, $e(n) = 2n$, $f(n) = n + 1$ et $g(n) = n$. Alors $d(n) - e(n) = 3n$ mais ça n'est pas $O(n + 1 - n) = O(1)$.

6. On suppose l'algorithme suivant qui calcule et retourne la multiplication de deux entiers positifs m et n.

```
  def multiplier(m,n) :
1    x = 0
2    while n > 0 :
3       if n%2 == 1:
4          x = x + m
5       m = 2 x m
6       n = n // 2
7    return x
```

   a. Compter le nombre d'opérations primitives exécutées à chaque ligne dans le pire des cas. En conclure sur la complexité en notation O.
   b. Quel est la borne supérieure de la complexité dans le pire des cas en notation Ω ? En conclure sur la complexité en notation Θ.

a.
Ligne 1 : 1 opération (affectation) faite 1 fois
Ligne 2 : 1 opération (comparaison) faite $\lfloor \log_2 n \rfloor + 2$ fois
Ligne 3 : 2 opérations (modulo et comparaison) faites $\lfloor \log_2 n \rfloor + 1$ fois
Ligne 4 : 2 opérations (addition et affectation) faites $\lfloor \log_2 n \rfloor + 1$ fois
Ligne 5 : 2 opérations (multiplication et affectation) faites $\lfloor \log_2 n \rfloor + 1$ fois
Ligne 6 : 2 opérations (division et affectation) faites $\lfloor \log_2 n \rfloor + 1$ fois
Ligne 7 : 1 opération (return) faite 1 fois

Le nombre de mises à jour à appliquer à n pour atteindre 0 est $\lfloor \log_2 n \rfloor + 1$. La condition du tant-que a besoin d'être testée une fois de plus, d'où le $\lfloor \log_2 n \rfloor + 2$.
Dans le pire des cas, la condition de la ligne 3 est toujours vraie (ex. si $n = 2^k - 1$ avec k entier positif), alors l'instruction est exécutée $\lfloor \log_2 n \rfloor + 1$ fois.

Le nombre total d'opérations éxécutées dans le pire des cas est :

$$1 + 1 \times (\lfloor \log_2 n \rfloor + 2) + 8 \times (\lfloor \log_2 n \rfloor + 1) + 1 = 9 \times \lfloor \log_2 n \rfloor + 12$$

L'algorithme est donc en $O(\log_2 n)$.

b. Dans le pire des cas ($n = 2^k - 1$ avec k entier positif), l'algorithme a aussi besoin d'exécuter au moins le corps du tant-que $\lfloor \log_2 n \rfloor + 1$ fois. Donc l'algorithme est en $\Omega(\log n)$. Comme il est à la fois $O(\log n)$ et $\Omega(\log n)$, alors il est en $\Theta(\log n)$.

7. Ecrire une fonction de test de primalité d'un nombre entier plus grand que 1 passé en paramètre où le nombre de tests de multiplicité est n-2. Donner au moins deux variantes de votre fonction produisant un nombre différent d'opérations (plus petit).

Version avec n-2 opérations :

```
def nombrePremier(n) :
```

```
   for i in range(2,n) :
      if n % i == 0:
         return False
   return True
```

On fait bien n-2 tests, pour i allant de 2 à n-1.

Version avec n//2-2 opérations :

```
def nombrePremier(n) :
   for i in range(2,n//2) :
      if n % i == 0:
         return False
   return True
```

On peut s'arrêter à n//2-1, il n'y aura pas de diviseur entre n//2 et n-1.
On fait ici n//2-2 tests, pour i allant de 2 à n//2-1.

Version avec sqrt(n)-1 opérations :

```
def nombrePremier(n) :
   for i in range(2,int(math.sqrt(n))+1) :
      if n % i == 0:
         return False
   return True
```

On peut en fait s'arrêter à sqrt(n) car les diviseurs entre sqrt(n) et n-1 sont forcément des multiples des diviseurs entre 2 et sqrt(n), qui ont donc déjà été testé.
On fait ici sqrt(n)-1 tests, pour i allant de 2 à sqrt(n).

Version avec sqrt(n)-2//2 opérations :

```
def nombrePremier(n) :
   if n == 2 : return True
   for i in range(3,int(math.sqrt(n))+1,2) :
      if n % i == 0:
         return False
   return True
```

Comme tous les entiers pairs plus grands que 2 ne sont pas premiers (puisque multiples de 2), on peut les éliminer (incrément de 2 dans la boucle). Il ne faut pas oublier de traiter 2 à part.
On fait ici sqrt(n)-2//2 tests, pour i allant de 3 à sqrt(n) par pas de 2.