Coût et complexité

Un algorithme doit déjà être correct, c'est-à-dire faire qu'il est supposé faire ! Mais il doit aussi être efficace, c'est-à-dire donner "rapidement" la réponse: c'est l'objet de ce chapitre.

I) Coût et complexité d'un algorithme

1) Comment évaluer l'efficacité d'un algorithme ?

L'exécution d'un algorithme consomme de la mémoire (la RAM) et du calcul processeur. Tant que la RAM utilisée ne dépasse pas la quantité maximale disponible, la consommation de mémoire ne ralentit pas l'exécution, et le temps de réponse est à peu près proportionnel au nombre de calculs élémentaires effectués par le processeur.

2) Coût d'un algorithme, opérations élémentaires

D Le coût d'un algorithme est le nombre d'opérations élémentaires effectuées par l'algorithme. Les opérations élémentaires sont:

- faire une opération mathématique de base $(+, *, \times, /, \sqrt{\ })$
- lire une valeur contenue dans une variable, affecter une valeur à une variable
- effectuer un test (comparaison, égalité), faire une opération booléenne (and, or, not)

Par exemple:

```
1. def somme_entiers(n):
2.     total =  0                  # Affectation
3.     while n > 0:                   # Lecture puis comparaison
4.           total = total + n     # Lecture×2 puis somme puis affectation
5.           n  = n - 1            # Lecture puis différence puis affectation
6.     return total                   # Lecture
```

Cette fonction (qui calcule la somme des entiers de 1 à n) fait:

- 2n opérations mathématiques (n×L4 + n×L5)
- 4n + 1 lectures (n×L3 + 2n×L4 + n×L5 + L6)
- 2n + 1 affectations (L2 + n×L4 + n×L5)
- n comparaisons (n×L3)

Le coût est de 9n + 2 opérations élémentaires

En pratique, on ne compte pas toutes les opérations élémentaires, mais seulement les plus significatives de l'algorithme. Ici, par exemple, on pourrait ne compter que le nombre d'additions de L4 (il y en a n), qui sont le cœur de cet algorithme.

3) La taille des données

D La taille des données d'un algorithme est (en général) un entier *n* qui "mesure" les données à traiter.

Exemples typiques:

- donnée = entier n ⟶ taille = n
- donnée = liste ⟶ taille = longueur de la liste

4) Estimation du coût d'un algorithme: la complexité

Calculer le coût exact d'un algorithme devient vite fastidieux et est souvent impossible. De plus, qu'il y ait n^2 ou $n^2 + 2n$ opérations élémentaires, c'est pareil lorsque *n* est grand. D'où la notion de "complexité".

Le rôle de la complexité est d'estimer, dans le cas le plus défavorable, un ordre de grandeur du coût d'un algorithme en fonction

de la taille n des données lorsque n tend vers l'infini.

D La complexité d'un algorithme est en $f(n)$ lorsque le coût $C(n)$ de l'algorithme au pire, c'est à dire dans le cas le plus défavorable, vérifie $C(n) = O(f(n))$. Il faut essayer de trouver la meilleure (la plus petite) fonction f possible.

On rappelle que:

- $f(n) = O(g(n))$ lorsque $\frac{f}{g}$ est bornée en $+\infty \Leftrightarrow \exists\, M \in \mathbb{R}^+,\ \exists\, n_0 \in \mathbb{N} \,/\, \forall\ n \geqslant n_0,\ |f(n)| \leqslant M\,|g(n)|$
- En particulier: si $f(n) \sim k\,g(n)$ avec $k \in \mathbb{R}^{*+}$ alors $f(n) = O(g(n))$.

L'algorithme donné en exemple dans le 2) est de complexité $O(n)$.

5) Complexités usuelles

Plus la complexité $C(n)$ est "élevée", moins l'algorithme est efficace. Le plus souvent, on sera dans l'un des cas suivants:

D
- $C(n) = O(1)$: complexité constante (parfait)
- $C(n) = O(\ln(n))$: complexité logarithmique (excellent)
- $C(n) = O(n)$: complexité linéaire (très bon)
- $C(n) = O(n \times \ln(n))$: complexité quasi linéaire (très bon)
- $C(n) = O(n^2)$: complexité quadratique (moyen)
- $C(n) = O(n^p)$ $(p > 2)$: complexité polynomiale (très mauvais)
- $C(n) = C(a^n)$ $(a > 1)$: complexité exponentielle (catastrophique)
- $C(n) = O(n!)$: complexité factorielle (exécrable)

6) Ordre de grandeur des temps d'exécution suivant la complexité

On suppose que l'ordinateur effectue 10^7 opérations élémentaires par seconde. (C'est à peu près le cas avec Python, mais C++ serait cent fois plus véloce). On donne le temps d'exécution T(n) en fonction de n selon la complexité C(n) de l'algorithme.

n \ C(n)	ln (n)	n	n ln (n)	n^2	n^3
$n = 10^2$	500 ns	10 μs	46 μs	1 ms	100 ms
$n = 10^4$	900 ns	1 ms	9 ms	10 s	28 h
$n = 10^6$	1 μs	100 ms	1 s	28 h	X
$n = 10^8$	2 μs	10 s	200 s	31 a	X

n \ C (n)	2^n	n !
n = 10	100 μs	300 ms
n = 20	100 ms	7700 a
n = 100	$4 * 10^{15}$ a	X

Un algorithme qui résout théoriquement un problème mais qui ne donne jamais la réponse est une curiosité guère utile...

II) Exemples

Indiquez le paramètre mesurant la taille des données, les opérations élémentaires caractéristiques effectivement comptées et la complexité. Expliquez ce que calculent les algorithmes, donnez des exemples. Donnez aussi s'il y a lieu des exemples ou l'algorithme s'exécute le plus vite (meilleur des cas) ou le moins vite (pire des cas) possible.

1) Algorithmes sans boucles "for" ou "while" : complexité $O(1)$

```
def f1(n):  # n est un entier
    return ((n % 400) == 0) or (((n % 4) == 0) and ((n % 100) != 0))
```

2) Algorithmes avec uniquement des boucles "for" : complexité O(longueur de la boucle)

```
def f2(n):  # n est un entier
    s = 0
    for k in range(1, n + 1):
        s = s + k**3
    return s
```

```
def f3(n):  # n est un entier
    a, b = 1, 1
    for i in range(n - 1):
        a, b = b, a + b
    return b
```

```
from math import *
def f4(n):  # n est un entier
    for d in range(2, floor(sqrt(n)) + 1):
        if (n % d) == 0:
            return False
    return True
```

```
def f5(n):  # n est un entier
    s = 0
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            s = s + i / j
    return s
```

3) Algorithmes avec une boucle "while" : complexité plus difficile à calculer

```
def f6(n):  # n est un entier
    res = ""
    while n > 0:
        n, c = n // 2, n % 2
        res = str(c) + res
    return res
```

III) Deux ou trois petits problèmes

1) Evaluation expérimentale de la complexité de l'algorithme d'Euclide

```
def pgcd(a, b):  # a et b sont des entiers  avec a ⩾ b
  while b > 0:
     a, b = b, a % b
  return a
```

Le coût $C(a, b)$ est ici le nombre d'exécutions de la boucle while. On cherche à l'évaluer en fonction de $b = \min(a, b)$ et pour ce faire on cherche la plus "petite" fonction $f(b)$ telle que $\frac{C(a,b)}{f(b)}$ soit bornée lorsque $b \longrightarrow +\infty$. On va faire une série aléatoire de calculs de pgcd (a, b) avec b_min $\leqslant b \leqslant$ b_max et $a \leqslant b < 2\,b$ (ainsi la première division euclidienne de a par b pourra générer n'importe quel reste). Puis on tracera le graphe des valeurs de $\frac{C(a,b)}{f(b)}$ obtenues pour chacun des (par exemple 1000) calculs.

En faisant varier la plage [b_min, b_max] par exemple de $[10^n, 2\times 10^n]$ avec $n = 3, 6, 9$ pour simuler $b \longrightarrow +\infty$, on verra si la fonction f est la bonne. On peut commencer par exemple par $f(b) = b$ puis ajuster en fonction de ce qui est observé.

On propose les fonctions suivantes:

```
import numpy as np, matplotlib.pyplot as plt, random as rd
from math import *

def pgcd_cout(a,b): # a et b sont des entiers avec a >= b
    c = 0
    while b > 0:
        a, b, c = b, a % b, c + 1
    return c
```

```
def eval_cout_pgcd(b_min, b_max, nb_calculs):
    res = []
    for k in range(nb_calculs):
        b = rd.randint(b_min, b_max)
        a = rd.randint(b, 2 * b - 1)
        res.append(pgcd_cout(a, b) / f(b))
    plt.grid(True)
    plt.plot(res)
    plt.show()
```

a) Expliquer ce que font ces deux fonctions. Faire les tests conseillés avec $f(b) = b$ pour commencer puis ajuster f.

b) Que peut-on finalement conjecturer expérimentalement sur la complexité de la fonction pgcd ?

2) Etude d'une suite (Ou encore: mieux vaut bien s'y prendre si l'on veut avoir une réponse dans un délai raisonnable)

On pose $x_0 = 1$ et $\forall\, n \in \mathbb{N}$, $x_{n+1} = x_n + \frac{1}{x_n}$. On démontre (maths) que $x_n - \sqrt{2\,n} \longrightarrow 0$.

Etant donné $\varepsilon > 0$, on cherche le plus petit entier n (qui existe car $x_n - \sqrt{2\,n} \longrightarrow 0$) pour lequel $x_n - \sqrt{2\,n} \leqslant \varepsilon$.

a) Expliquer ce que font les trois fonctions suivantes, en précisant la différence entre recherche_1 et recherche_2:

```
def x(n):
    s = 1
    for i in range(n):
        s = s + 1/s
    return s

def recherche_1(eps):
    n = 0
    while x(n) - sqrt(2*n) > eps:
        n = n + 1
    return n

def recherche_2(eps):
    x, n = 1, 0
    while x - sqrt(2*n) > eps:
        x, n = x + 1/x, n + 1
    return n
```

On note $C_1(\varepsilon)$ et $C_2(\varepsilon)$ le coût des fonctions recherche_1(eps) et recherche_2(eps), n le retour de ces fonctions et T1(ε) et T2(ε) les durées d'exécution. La fonction $x(n)$ a un coût en $\alpha\, n$ et est appelée à chaque itération de la boucle while dans recherche_1, donc C1(ε) = $\sum_{k=1}^{n} (\alpha\, k + O(1)) \approx$ k1 n^2 mais pas dans recherche_2 donc C2(ε) = $\sum_{k=1}^{n} O(1) \approx$ k2 n. Les temps de calculs étant proportionnels aux coûts, on en déduit l'existence de constantes k_1 et k_2 telles que T1(ε) $\approx k_1\, n^2$ et T2(ε) $\approx k_2\, n$ et donc l'existence d'une constante k telle que T1(ε) $\approx k * n *$ T2(ε) . On va vérifier expérimentalement tout cela. La fonction ci dessous donne avec précision le temps d'exécution de fonction(eps) et présente correment le résultat.

```
def duree_calcul(fonction, eps):
    t = time.perf_counter()          # time.clock() avec Python 3.2
    n = fonction(eps)
    t = time.perf_counter() - t      # time.clock() avec Python 3.2
    print("{0}({1}) = {2} ; temps de calcul = {3:5.4} s".format(fonction.__name__,
eps, n, t))
```

b) Exécuter duree_calcul(f, eps) pour f = recherche_1 puis recherche_2 et eps $\in \{0.08 , 0.04 , 0.02\}$. Calculer la constante k tel que T1(ε) $\approx k * n *$ T2(ε) .

c) Exécuter duree_calcul(recherche_2, 0.01), estimer (avec une calculette) le temps de calcul de duree_calcul(recherche_1, 0.01) puis le vérifier.

d) Exécuter duree_calcul(recherche_2, 0.001). Combien de jours prendrait environ l'exécution de recherche_1(0.001) ?

Et s'il reste du temps, voir un troisième problème sur maths-algo.fr

3) Calcul de la somme maximum de termes consécutifs d'une liste de nombres (Voir la suite sur maths-algo.fr)

Etant donnée une liste s de nombres positifs ou négatifs, les 3 fonctions suivantes calculent

$M(s) = \max\left\{\sum_{k=i}^{j} s[k] \ / \ 0 \leqslant i \leqslant j < n = \text{len}(s)\right\}$, c'est à dire la plus grande somme de termes consécutifs de la liste s.

```
def somme_max_1(s):
    n, maxi = len(s), s[0]
    for i in range(n):
        for j in range(i, n):
            somme = 0
            for k in range(i, j + 1):
                somme = somme + s[k]
            if somme > maxi:
                maxi = somme
    return maxi
```

```
def somme_max_2(s):
    n, maxi = len(s), s[0]
    for i in range(n):
        somme = 0
        for j in range(i, n):
            somme = somme + s[j]
            if somme > maxi:
                maxi = somme
    return maxi
```

```
def somme_max_3(s):
    n, maxi, somme = len(s), 0, 0
    for i in range(n):
        if somme > 0:
            somme = somme + s[i]
        else:
            somme = s[i]
        if somme > maxi:
            maxi = somme
    return maxi
```

a) Expliquer leur fonctionnement et et pourquoi elles calculent toutes les trois $M(s)$.

Tester par exemple avec s = s = [-1, 4, -1, 4 , -3, -2, 5, -1, 2].

b) Calculer la complexité de ces trois fonctions.

c) On va tester ces trois fonctions sur des listes de n entiers de ⟦-100, 100⟧ générées par la fonction suivante

```
def liste_test(n):
    return [round(100*sin(i*i)) for i in range(n)]
```

Chercher par tâtonnement des ordres de grandeur des plus grands entiers $N_i = N_1, N_2, N_3$ pour lequel somme_max _i (s)) s'exécute en moins de trois secondes environ pour $s = s(N_i)$. Les résultats sont-ils cohérents avec les calculs théoriques de complexité du b) ?