CI 2 – Algorithmique et Programmation

Chapitre 3 – Performance des algorithmes

Savoir

Savoirs :
- justifier qu'une itération (ou boucle) produit l'effet attendu au moyen d'un invariant ;
- démontrer qu'une boucle se termine effectivement ;
- s'interroger sur l'efficacité algorithmique temporelle.

Ce document évolue. Merci de signaler toutes erreurs ou coquilles.

1 Invariance de boucle

1.1 Problématique

Lors de la réalisation d'un algorithme, il est nécessaire
- de vérifier que celui-ci permet bien de répondre au problème initial ;
- de s'assurer que celui-ci se termine sans quoi le résultat du problème ne sera jamais délivré à l'utilisateur.

On s'intéresse ici uniquement aux structures itératives. (Les structures récursives seront traitées en seconde année.)

Définition

Invariant de boucle [?]

Un invariant de boucle est une propriété ou une formule logique,
- qui est vérifiée après la phase d'initialisation ;
- qui reste vraie après l'exécution d'une itération ;
- et qui, conjointement à la condition d'arrêt, permet de montrer que le résultat attendu et bien le résultat calculé.

Méthode

[?]

1. Définir les préconditions (état des variables avant d'entrer dans la boucle)
2. Définir un invariant de boucle
3. Prouver l'invariant (correspond à $\mathscr{P}(n) \Longleftarrow \mathscr{P}(n+1)$).

Méthode

4. Montrer la terminaison du programme.
5. Condition de sortie de boucle + invariant de boucle $\Longleftarrow$ postcondition.

1.2 Exemple – L'algorithme d'Euclide.

Cet algorithme permet de calculer le PGCD de deux nombres entiers. Il se base sur le fait que si a et b sont deux entiers naturels non nuls, $pgcd(a,b) = pgcd(b, a \bmod b)$.

Pseudo Code

Data : $a, b \in \mathbb{N}^*$, $a < b$
$x \leftarrow a$
$y \leftarrow b$
tant que $y \neq 0$ **faire**
  $r \leftarrow$ reste de la division euclidienne de x par y
  $x \leftarrow y$
  $y \leftarrow r$
fin

Exemple

Calculons le PGCD de 240 et 64 :

$$\begin{aligned} 240 &= 3 \cdot 64 + 48 \\ 64 &= 1 \cdot 48 + 16 \\ 48 &= 3 \cdot 16 + 0 \end{aligned}$$

$pgcd(240, 64)$ est le dernier reste non nul à savoir 16.

Démonstration

Soient a et b, tels que $(a;b) \in \mathbb{N}^*$ tels que $a > b$. Soient $(q;r) \in \mathbb{N}$ tels que $a = b \cdot q + r$. Avec r le reste de la division euclidienne de a par b et q le quotient.

Montrons que $PGCD(a,b) = PGCD(b,r)$.

Soit $d \in \mathbb{N}$ un diviseur commun de a et b. Dans ce cas, d divise $a - b \cdot q$; donc d divise r.

En conclusion, si d divise a et b alors $PGCD(a,b) = PGCD(b,r)$.

Réciproquement, soit $d \in \mathbb{N}$ un diviseur commun de b et r. Dans ce cas, d divise $b \cdot q + r$; donc d divise a.

Démonstration

En conclusion, si d divise b et r alors $PGCD(a,b) = PGCD(b,r)$.

On a donc $PGCD(a,b) = PGCD(b,r)$.

Démonstration

Preuve par invariant de boucle

Montrons que la propriété $x = y \cdot q + r$ est un invariant de boucle :
- au premier incrément, r étant le reste de la division euclidienne de x par y, on a donc bien $x = y \cdot q + r$;
- à l'incrément suivant, $x' \leftarrow y$ et $y' \leftarrow r$. On peut donc trouver r' tel que $x' = y' \cdot q' + r'$. La propriété reste donc vraie ;
- par définition de la division euclidienne, $r < y$; donc à chaque incrément la quantité y diminue. La boucle pourra donc s'arrêter.

2 Complexité des algorithmes

2.1 Présentation

Il existe souvent plusieurs façons de programmer un algorithme. Si le nombre d'opérations à effectuer est peu important et les données d'entrée de l'algorithme sont de faibles tailles, le choix de la solution importe peu. En revanche, lorsque le nombre d'opérations et la taille des données d'entrée deviennent importants, deux paramètres deviennent déterminants : le temps d'exécution et l'occupation mémoire.

Définition

Complexité en temps

La complexité en temps donne le nombre d'opérations effectuées lors de l'exécution d'un programme. On appelle C_o le coût en temps d'une opération o.

Complexité en mémoire (ou en espace)

La complexité en mémoire donne le nombre d'emplacements mémoires occupés lors de l'exécution d'un programme.

Remarque

On distingue la complexité dans le pire des cas, la complexité dans le meilleure des cas, ou la complexité en moyenne. En effet, pour un même algorithme, suivant les données à manipuler, le résultat sera déterminé plus ou moins rapidement.

Généralement, on s'intéresse au cas le plus défavorable à savoir, la complexité dans le pire des cas.

Résultat

Soit une suite de deux instructions ayant un coût respectif de C_1 et C_2. Le coût total de la suite d'opération est donc $C = C_1 + C_2$.

Plus généralement, lorsqu'on réalise une itération, le coût total correspond à la somme des différents coûts. En notant C_i le coût de la $i^{\text{ème}}$ suite d'instruction, le coût total de l'itération est donc :

$$C = \sum_{i=1}^{n} C_i$$

Exemple

Calcul de factorielle

Pseudo Code

```
Début Fonction
    factorielle(n) :
    si n=0 alors
        retourner 1
    sinon
        i ← 1
        res ← 1
        tant que i ≤ n faire
            res ← res · i
            i ← i + 1
        fin
        retourner res
    fin
Fin
```

Complexité en mémoire : lors de l'exécution du porgramme, il sera nécessaire de stocker les variables suivantes :

- n ;
- res ;
- i.

Complexité en temps : On réalise une première comparaison dont on note le coût C_c. lorsque $n > 0$, on réalise à chaque incrément :

- une comparaison de coût C_c ;
- une multiplication de coût C_m ;
- une addition (itération) de coût C_i ;
- deux affectations de coût C_a ;

Exemple

Pour calculer $n!$ la boucle est réitérée n fois. En conséquence, la complexité en temps s'élève à :

$$C = n \cdot (C_c + C_m + C_a + C_i) + C_c$$

Il est fréquent que la complexité en temps soit améliorée au prix d'une augmentation de la complexité en espace, et vice-versa. La complexité dépend notamment :

- de la puissance de la machine sur laquelle l'algorithme est exécuté ;
- du langage et compilateur / interpréteur utilisé pour coder l'algorithme ;
- du style du programmeur.

Remarque

Le coût de la mémoire étant aujourd'hui relativement faible, on cherche en général à améliorer la complexité en temps plutôt que le complexité en mémoire.

2.2 D'autres exemples

2.2.1 Recherche d'un maximum

Soit une liste de nombre entiers désordonnés. Comment déterminer le plus grand nombre de la liste ?

Intuitivement, une solution est de parcourir la liste d'élément et de déterminer le plus grand élément par comparaisons successives.

Pseudo Code

```
Data : tab tableau de taille n
max ← −∞
for i = 1 to n do
    if tab[i] > max then
        max ← tab[i]
    end
end
```

Exemple

Ainsi, quelle que soit la taille n du tableau, il faudra n itération pour connaître le plus grand élément de la liste.

2.2.2 Tri d'une liste

Algorithme naïf

Soit une liste de nombre entiers désordonnés. Comment les trier par ordre croissant ?

Une méthode dite naïve pourrait être la suivante :
- trouver le plus petit élément du tableau. Notons min son indice ;
- on permute alors le min^{e} élément avec le premier élément ;
- ...
- on trouve le plus petit élément du tableau compris entre l'indice i et N ;
- on permute alors le min^{e} élément avec le i^{e} élément.

Pseudo Code

Data : tab tableau d'entiers désordonnés de taille n
Result : tab tableau d'entiers ordonnés
for $i = 1$ **to** $n-1$ **do**
  $min \leftarrow i$
  for $j = i+1$ **to** n **do**
    if $tab[j] < tab[min]$ **then**
      $min \leftarrow j$
    end
  end
  $tmp \leftarrow tab[i]$
  $tab[i] \leftarrow tab[min]$
  $tab[min] \leftarrow tmp$
end

Exemple

Dans ce cas, pour un tableau de taille n, la première boucle sera réalisée $n-1$ fois. Lors de l'itération i, la comparaison située dans la seconde boucle sera exécutée $N-(i+1)$ fois.

En considérant que seule la comparaison à un coût C, le coût total s'obtient en comptant le nombre de comparaisons :

$$\begin{aligned}(N-(1+1)+1) &+ (N-(2+1)+1)+\ldots+(N-(i+1)+1)+\ldots+(N-(N)+1)+(N-(N+1)+1)\\ &= (N-1)+(N-2)+\ldots+1\\ &= \frac{N(N-1)}{2}\end{aligned}$$

Remarque

Il existe d'autres algorithmes de tris plus performant que l'algorithme présenté ci-dessus :
- tri de shell (shell sort) ;
- tri fusion (merge sort) ;
- tri rapide (quick sort)...

2.3 Complexité algorithmique

Définition

[**?**] Soient f et g deux fonctions $f, g : \mathbb{N} \to \mathbb{R}_*^+$. On note $f(n) = \mathcal{O}(g(n))$ lorsqu'il existe des entiers c et n_0 tels que pour tout $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

Intuitivement, cela signifie que f est inférieur à g à une constante multiplicative près pour les données suffisamment grandes.

Exemple

Ainsi, l'algorithme de recherche du maximum dans une liste non trié (présenté précédemment) est de complexité $\mathcal{O}(n)$ où n est le nombre d'éléments de la liste. Cet algorithme est proportionnel au nombre d'éléments.

L'algorithme de tri naïf est de complexité $\mathcal{O}(n^2)$. On parle d'algorithme quadratique. Le temps d'exécution devient très grand lorsque le nombre de données et très important.

Par ordre de complexité croissante on a donc :
- $\mathcal{O}(1)$: algorithme s'exécutant en temps constant, quelle que soit la taille des données ;
- $\mathcal{O}(log(n))$: algorithme rapide (complexité logarithmique) (Exemple : recherche par dichotomie dans un tableau trié) ;
- $\mathcal{O}(n)$: algorithme linéaire ;
- $\mathcal{O}(n \cdot log(n))$: complexité $n \log n$;
- $\mathcal{O}(n^2)$: complexité quadratique ;
- $\mathcal{O}(2^n)$: complexité exponentielle.

Résultat

Pour une opération ayant un temps d'exécution de $10^{-9}s$, on peut calculer le temps d'exécution en fonction du nombre de données et de la complexité de l'algorithme :

Données	$\mathcal{O}(log(n))$	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(2^n)$
100	$2 \cdot 10^{-9}\ s$	$0,1 \cdot 10^{-6}\ s$	$0,2 \cdot 10^{-6}\ s$	$10 \cdot 10^{-6}\ s$	$1,26765 \cdot 10^{21}\ s$
1 000	$3 \cdot 10^{-9}\ s$	$1 \cdot 10^{-6}\ s$	$3 \cdot 10^{-6}\ s$	$0,001\ s$	$1,0715 \cdot 10^{292}\ s$
10 000	$4 \cdot 10^{-9}\ s$	$10 \cdot 10^{-6}\ s$	$40 \cdot 10^{-6}\ s$	$0,1\ s$	$+\infty$

3 Profiling des algorithmes

Afin d'évaluer la performance des algorithmes, il existe des fonctionnalités permettant de compter le temps consacré à chacune des fonctions ou à chacune des instructions utilisées dans un programme http://docs.python.org/2/library/profile.html.

Exemple

Voici un exemple du crible d'Eratosthène.

```python
def crible(n):
    tab=[]
    for i in range(2,n):
        tab.append(i)
    for i in range(0,len(tab)):
        for j in range(len(tab)-1,i,-1):
            if (tab[j]%tab[i]==0):
                tab.remove(tab[j])
    return tab

import cProfile
cProfile.run('crible(10000)')
```

cProfile renvoie alors le message suivant :

```
28770 function calls in 1.957 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.957    1.957 <string>:1(<module>)
        1    0.420    0.420    1.957    1.957 eratosthene.py:4(crible)
        1    0.000    0.000    1.957    1.957 {built-in method exec}
     9999    0.015    0.000    0.015    0.000 {built-in method len}
     9998    0.016    0.000    0.016    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     8769    1.505    0.000    1.505    0.000 {method 'remove' of 'list' objects}
```

On alors le bilan du temps passé à effectuer chacune des opérations. Ainsi pour améliorer notablement l'algorithme, le plus intéressant serait d'optimiser la méthode remove.

Références

[1] François Denis http://pageperso.lif.univ-mrs.fr/~francois.denis/algoL2/chap1.pdf

[2] Alain Soyeur http://asoyeur.free.fr/

[3] François Morain, Cours de l'Ecole Polytechnique, http://www.enseignement.polytechnique.fr/profs/informatique/Francois.Morain/TC/X2004/Poly/www-poly009.html.

[4] Renaud Kerivent et Pascal Monasse, La programmation pour ... , Cours de l'École des Ponts ParisTech - 2012/2013 http://imagine.enpc.fr/~monasse/Info.

[5] Olivier Bournez, Cours INFO 561 de l'Ecole Polytechnique, Algorithmes et programmation, http://www.enseignement.polytechnique.fr/informatique/INF561/uploads/Main/poly-good.pdf.