Informatique en CPGE (2018-2019)
Algorithmes : validité et complexité

1 Validité d'un algorithme itératif

Un algorithme itératif est construit avec des boucles, par opposition à récursif qui remplace les boucles par des appels à lui-même.

1.1 Invariants de boucle

1.1.1 Le principe

La méthode des "invariants de boucle" aide à prouver la validité d'un algorithme itératif.

Définition

Un invariant d'une boucle est une propriété qui est vérifiée à chaque exécution du corps de cette boucle. On utilise un raisonnement par récurrence pour prouver qu'une propriété est un invariant d'une boucle :

- initialisation : on montre que la propriété est vérifiée avant le premier passage dans le corps de la boucle,
- hérédité : on montre que si l'invariant est vérifié avant une exécution quelconque du corps de la boucle, il est encore vérifié après son exécution, donc avant l'exécution suivante.
- conclusion : l'invariant est alors vérifié à la fin de la boucle, ce qui traduit le fait que la boucle réalise bien la tâche souhaitée.

1.1.2 Exemple

On effectue la division euclidienne de a par b où a et b sont deux entiers strictement positifs. Il s'agit donc de déterminer deux entiers q et r tels que $a = bq+r$ avec $0 \leq r < b$. Voici un algorithme déterminant q et r :

```
q = 0
r = a
tant que r ≥ b
    q = q + 1
    r = r − b
fin du tant que
```

On choisit comme invariant de boucle la propriété $a = bq + r$.

— Initialisation : q est initialisé à 0 et r à a, donc la propriété $a = bq + r = b.0 + a$ est vérifiée avant le premier passage dans la boucle.

— Hérédité : avant une itération arbitraire, supposons que l'on ait $a = bq + r$ et montrons que cette propriété est encore vraie après cette itération. Soient q' la valeur de q à la fin de l'itération et r' la valeur de r à la fin de l'itération. Nous devons montrer que $a = bq' + r'$. On a $q' = q + 1$ et $r' = r - b$, alors $bq' + r' = b(q + 1) + (r - b) = bq + r = a$. La propriété est bien conservée.

1.2 Terminaison

1.2.1 Le principe

L'algorithme étudié ci-dessus semble valide, mais rien ne prouve pour l'instant que le programme se termine, et que lorsqu'il s'arrête, on aura bien $0 \leq r < b$.

Pour prouver qu'un programme s'arrête, on choisit une variable et on vérifie que la suite formée par les valeurs de cette variable au cours des itérations converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt. Cette variable s'appelle un variant de boucle.

1.2.2 Exemple

Nous reprenons l'exemple précédent.

— Commençons par montrer que le programme s'arrête : la suite formée par les valeurs de r au cours des itérations est une suite d'entiers strictement décroissante : r étant initialisé à a, si $a \geq b$ alors la valeur de r sera strictement inférieure à celle de b en un maximum de $a - b$ étapes.

— Ensuite, si le programme s'arrête, c'est que la condition du "tant que" n'est plus satisfaite, donc que $r < b$. Il reste à montrer que $r \geq 0$. Comme r est diminué de b à chaque itération, si $r < 0$, alors à l'itération précédente la valeur de r était $r' = r + b$; or $r' < b$ puisque $r < 0$. Et donc la boucle se serait arrêtée à l'itération précédente, ce qui est absurde ; on on déduit que $r \geq 0$.

En conclusion, le programme se termine avec $0 \leq r < b$ et la propriété $a = bq + r$ est vérifiée à chaque itération ; ceci prouve que l'algorithme effectue bien la division euclidienne de a par b.

1.3 Un autre exemple

L'objectif est de calculer le produit de deux nombres entiers positifs a et b sans utiliser de multiplication :

$p = 0$
$m = 0$
tant que $m < a$
    $m = m + 1$
    $p = p + b$
fin du tant que

Comme dans l'exemple précédent, le programme se termine car la suite des valeurs de m est une suite d'entiers consécutifs strictement croissante, et atteint la valeur a en a étapes.

Un invariant de boucle est ici : $p = m.b$

— Initialisation : avant le premier passage dans la boucle, $p = 0$ et $m = 0$, donc $p = m.b$.

— Hérédité : supposons que $p = m.b$ avant une itération ; les valeurs de p et m après l'itération sont $p' = p + b$ et $m' = m + 1$. Or $p' = (p + b) = m.b + b = (m + 1)b = m'b$. Donc la propriété reste vraie.

— Conclusion : à la sortie de la boucle $p = m.b$.

Puisqu'à la sortie de la boucle $m = a$, on a bien $p = a.b$.

2 Complexité

Pour traiter un même problème, il existe souvent plusieurs algorithmes et un algorithme n'aura pas le même temps d'exécution s'il est programmé dans deux langages différents ou s'il est exécuté sur deux machines différentes. Pour pouvoir comparer deux algorithmes il faut donc trouver un moyen de mesurer le temps d'exécution indépendamment du langage et de la machine.

En général, le temps d'exécution d'un algorithme sera évalué en fonction de la taille des données. Par exemple, le temps d'exécution d'une recherche dans un tableau dépend de la taille de ce tableau. Cette évaluation peut se révéler parfois très difficile, mais dans de nombreux cas, elle peut se faire en appliquant quelques règles simples.

2.1 Mesure du temps d'exécution

La quantité de données prise en entrée sera notée n ; par exemple la taille d'un tableau, le nombre de caractères d'une chaîne, etc.

La procédure est la suivante :
— nous comptons le nombre d'opérations u_n exécutées par l'algorithme sans nous intéresser au temps mis par une machine particulière à exécuter une opération.
— nous étudions la croissance du nombre d'opérations u_n effectuées par l'algorithme en fonction du nombre n de données.

Les règles pour évaluer le temps d'exécution sont alors les suivantes :

- le temps d'exécution, relativement petit, d'une affectation, d'une comparaison ou d'une opération simple constitue l'unité de base ;
- le temps pris pour exécuter une séquence `[p q]` est la somme des temps pris pour exécuter les instructions `p` puis `q` ;
- le temps pris pour exécuter un test `if (b): p else: q` est inférieur ou égal au maximum des temps pris pour exécuter les instructions `p` et `q`, plus une unité pour évaluer l'expression `b` ;
- le temps pris pour exécuter une boucle `for i variant de 1 à m: p` est `m` fois le temps pris pour exécuter l'instruction `p`. Si le nombre `m` ne dépend pas des entrées de l'algorithme, alors le temps pris pour exécuter cette boucle est une constante ;
- le cas des boucles `while` est plus complexe à traiter puisqu'en général le nombre de répétitions n'est pas connu a priori.

Pour une même quantité de données, le temps d'exécution peut être très variable : lorsqu'on recherche un élément dans un tableau, l'élément peut se trouver au début du tableau dans le cas le plus favorable ou à la fin du tableau dans le pire des cas. Le plus souvent nous considérerons le pire des cas.

2.2 Borne asymptotique supérieure

Une suite u est de borne asymptotique supérieure v si pour n assez grand, u est majorée par v à une constante près. On écrira $u_n \in \mathcal{O}(v_n)$.

Définition : $u_n \in \mathcal{O}(v_n)$ si $\exists N \in \mathbb{N}, \exists k \in \mathbb{R}, \forall n > N, u_n \leq kv_n$. (u est majorée par $k.v$)

En pratique, on utilise la propriété suivante :
Propriété : $u_n \in \mathcal{O}(v_n)$ si $\lim\limits_{n\to+\infty} \dfrac{u_n}{v_n} = \ell$ (autrement dit, si $\frac{u_n}{v_n}$ converge vers une constante).

2.3 Niveaux de complexité

2.3.1 Complexité constante

Un algorithme est en temps constant, $\mathcal{O}(1)$, si son temps d'exécution est indépendant du nombre de données ; par exemple, retourner le premier élément d'une liste ne dépend pas de la longueur de la liste.

2.3.2 Complexité logarithmique

Un algorithme de complexité logarithmique, $\mathcal{O}(\ln n)$, croît de façon linéaire en fonction de 2^n ; cela signifie que pour doubler le temps d'exécution, il faut mettre au carré le nombre de données. La recherche par dichotomie dans un tableau trié est de complexité $\mathcal{O}(\ln n)$.

Pour la plupart, ces algorithmes sont dans la pratique très performants.

2.3.3 Complexité linéaire

Un algorithme est de complexité linéaire, $\mathcal{O}(n)$, si le temps de calcul croît de façon linéaire en fonction du nombre de données. La recherche séquentielle dans un tableau non trié est de complexité linéaire : dans le pire des cas, l'élément que l'on recherche est celui qui est examiné en dernier et il est donc nécessaire d'effectuer n itérations (qui s'effectuent en temps constant) pour examiner chaque élément du tableau. (Autres exemples : le calcul d'une somme ou d'une moyenne.)

2.3.4 Complexité log-linéaire

Certains algorithmes de tri sont de complexité $\mathcal{O}(n.\ln n)$. C'est le meilleur résultat qu'il est possible d'obtenir avec des tris par comparaisons.

2.3.5 Complexité quadratique

D'autres algorithmes de tri, construits avec deux boucles imbriquées, effectuent un nombre d'itérations de l'ordre de n^2. Souvent, les problèmes pour lesquels il existe des algorithmes quadratiques, $O(n^2)$, admettent aussi des algorithmes de meilleure complexité.

2.3.6 Complexité polynomiale

Ce sont les algorithmes, en $\mathcal{O}(n^k)$, k fixé, dont le temps d'exécution peut être majoré par un polynôme. On utilise le terme polynomial par opposition à exponentiel. En algorithmique, il est très important de faire la différence entre les algorithmes polynomiaux, utilisables en pratique, et les algorithmes exponentiels, inutilisables en pratique.

2.3.7 Complexité exponentielle

Ces algorithmes en $\mathcal{O}(k^n), k > 1$, sont tellement longs à l'exécution, qu'on ne les utilise presque jamais. Malheureusement, il existe des problèmes pour lesquels les seuls algorithmes de résolution exacte connus à l'heure actuelle sont de complexité exponentielle.

3 Exemples

3.1 Recherche séquentielle dans un tableau non trié

Le programme suivant effectue la recherche d'un élément dans un tableau t de taille n par balayage (en anglais, linear search = recherche linéaire). Le programme s'arrête dès que l'élément est trouvé ou lorsque le compteur i atteint la valeur n. Dans le pire des cas, on parcourt donc tout le tableau et la complexité est en $\mathcal{O}(n)$.

Pour le tableau t, on peut utiliser les types `list`, `tuple`, ou même le type `str`.

```
def recherche(x,t):
    n=len(t)
    i=0
    while(i<n) and (x!=t[i]):
        i=i+1
    if i<n:
        return i
    else:
        return False
```

En utilisant une boucle "for", on peut écrire un programme plus condensé (ci-dessous) mais la complexité est toujours en $\mathcal{O}(n)$.

```
def search(x,t):
    for i in range(len(t)):
        if t[i]==x:
            return i
    return False
```

3.2 Recherche par dichotomie dans un tableau trié

Si le tableau est trié on peut utiliser le principe de dichotomie (binary search en anglais). A chaque étape, on coupe le tableau en deux et on effectue un test pour savoir dans quelle partie peut se trouver l'élément cherché. C'est le principe *diviser pour régner* (en anglais *divide-and-conquer*). On utilise ici le type `list` pour pouvoir préalablement trier le tableau avec la méthode `sort()`.

```
def rechercheDichotomie(x,liste):
    g=0
    d=len(liste)
    while g<d-1:
        k=(g+d)//2
        if x<liste[k]:
            d=k
        else:
            g=k
    if x==liste[g]:
        return g
    else:
        return False
```

Après k itérations, si n est la taille du tableau, $d-g \leq \dfrac{n}{2^k}$ (démonstration par récurrence).

Or $\dfrac{n}{2^k} \leq 1$ dès que $k \geq \dfrac{\ln n}{\ln 2}$ et dans ce cas le programme s'arrête. La complexité de la recherche dichotomique est donc $\mathcal{O}(\ln n)$.

3.3 Recherche d'un mot dans une chaîne de caractères

Le principe est le même que pour la recherche d'un caractère dans une chaîne mais ici il nécessaire d'ajouter une boucle "while" pour tester tous les caractères du mot.

```
def searchWord(mot,texte):
    if len(mot)>len(texte):
        return False
    for i in range(1+len(texte)-len(mot)):
        j=0
        while j<len(mot) and mot[j]==texte[i+j]:
            j+=1
        if j==len(mot):
            return i
    return None
```

Dans le pire des cas le programme va chercher le mot à toutes les places possibles et va tester tous les caractères du mot. La complexité est donc de l'ordre de $m(1+n-m)$ où m est la longueur du mot et n la longueur du texte. En particulier le maximum est atteint pour $m=\dfrac{n}{2}$ et on obtient alors une complexité de l'ordre de $\dfrac{n^2+2n}{4}$; dans ce cas la complexité de l'algorithme est donc en $\mathcal{O}(n^2)$.