Analyse d'algorithmes

Nous allons nous intéresser à l'analyse de programmes en posant (et tenter de répondre à) 3 questions concernant un problème que nous aide à résoudre un programme :

1. Le programme fournit-il un résultat ? Autrement dit, est-ce que le programme termine [1] et renvoie un résultat ?
2. Le programme est-il correct ? Autrement dit, le résultat renvoyé par le programme est-il le bon ?
3. Le programme fournit-il le bon résultat dans un temps raisonnable ?

Si, pour des programmes simples, la réponse à ces question va (normalement) de soi, il est nécessaire de développer des outils plus généraux et puissants pour pouvoir traîter des programmes plus complexes.

I. Complexité

1. Rappels

L'analyse de la complexité d'un algorithme est l'étude de la quantité de ressources (principalement en **temps** et en **espace**) nécessaire à son exécution.

Nous allons plus particulièrement nous intéresser à la **complexité temporelle dans le pire des cas**, que l'on cherche à déterminer l'ordre de grandeur en fonction de la taille des données d'entrée, ce qui peut être :

- ☞ la valeur d'une variable entière,
- ☞ la taille d'un tableau,
- ☞ la longueur d'un mot,
- ☞ le degré d'un polynôme,
- ☞ etc.

En notant n la taille des données d'entrée, on parle de complexité…

- ☞ constante quand elle est en $O(1)$,
- ☞ logarithmique quand elle est en $O(\ln n)$,
- ☞ linéaire quand elle est en $O(n)$,
- ☞ linéarithmique quand elle est en $O(n \ln n)$,

1. En informatique, on utilise le verbe terminer de façon intransitive, et on parle de terminaison.

- ☞ quadratique quand elle est en $O(n^2)$,
- ☞ polynomiale quand elle est en $O(n^d)$ pour un certain entier d,
- ☞ exponentielle quand elle est en $O(a^n)$ pour un certain réel $a > 1$.

2. Exemples

2.1. Parcours d'un tableau

De nombreux algorithmes simples sur les tableaux ou les listes ont une forme similaire, comme par exemple le calcul de la somme des éléments d'une liste ou la recherche de son minimum.

```
def minimum(l):
    mini = l[0]
    for i in range(1, len(l)):
        if mini > l[i]:
            mini = l[i]
    return mini
```

En notant n la longueur de la liste passée en argument, on a une boucle `for` répétée $n-1$ fois, le corps de la boucle étant en temps constant, comme l'initialisation initiale de `mini`. La fonction est donc de complexité linéaire, en $O(n)$.

2.2. Recherche dans une liste

Dans cette variante de parcours de liste, où l'on recherche si un élément précis y est présent, il n'est pas forcément nécessaire de parcourir la liste en entier. [2]

```
def recherche(l, e):
    trouvé = False
    i = 0
    while not trouvé and i < len(l):
        if l[i] == e:
            trouvé = True
        else:
            i = i + 1
    return trouvé
```

À nouveau, le corps de la boucle s'effectue en temps constant, et comme on considère la complexité **dans le pire des cas**, la fonction est encore en $O(n)$.

2. Pour simplifier l'analyse, on évite d'avoir un « `return` » ou un « `break` » à l'intérieur de la boucle.

II. Terminaison

Le seul cas (pour l'instant [3]) où la terminaison d'un programme peut être problématique est lorsque l'on rencontre une boucle *non bornée* de type « `while` ».

Un moyen pratique pour prouver qu'une boucle ne va pas être répétée indéfiniment est d'identifier un *variant de boucle*.

Définition (Variant de boucle) Un **variant de boucle** est une quantité entière dépendant des variables du programme qui :

☞ est à valeurs positives lorsque la condition de la boucle est vérifiée ;

☞ décroit *strictement* d'une itération à l'autre.

Proposition - Terminaison de boucle avec variant

Si une boucle admet un variant, alors elle termine nécessairement.

Preuve
Si la boucle se répètait indéfiniment, les valeurs successives du variant formeraient une suite strictement décroissante d'entiers naturels, ce qui est absurde.

Exemple Considérons la fonction suivante :

```
def appartient(val, tableau):
    p = 0
    while p < n and tableau[p] != val:
        p = p + 1
    return p < n
```

Un *variant de boucle* est ici clairement la quantité « `n - p` », puisque :

☞ Si la condition de boucle « `p < n and tableau[p] != val` » est vérifiée, alors bien sûr $n - p \geq 0$.

☞ D'une itération à l'autre, ayant effectué `p = p + 1`, le variant « `n - p` » diminue strictement.

L'existence d'un variant nous assure que la boucle va terminer.

Exercice 1 Prouver que la fonction suivante termine.

```
def compte(n):
    k = n
    cnt = 0
    while k > 0:
        if k % 2 == 1:
```

3. La récursivité, que l'on verra plus tard, peut aussi amener ce genre de problèmes.

```
            cnt = cnt + 1
        k = k // 2
    return cnt
```

Remarque La notion de *variant de boucle* est un outil assez simple pour déterminer si une boucle termine. Cependant, il n'est pas toujours possible d'en trouver. Un exemple classique est la fonction suivante :

```
def siracuse(n):
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
```

Personne ne sait à l'heure actuelle si cette fonction termine. D'après le mathématicien hongrois Paul Erdős, « les mathématiques ne sont peut-être pas encore prêtes pour de tels problèmes. »

> **Question** Concrètement, que signifie le fait que l'on ne sait pas si cette fonction termine ?

III. Correction

Dernière question importante, comment être sûr (au sens mathématique) que le résultat renvoyé par une fonction est le bon ?

Par exemple, considérons le programme suivant :

```
def somme(l):
    s = 0
    for i in range(len(l)):
        s = s + l[i]
    return s
```

que nous réécrivons avec une boucle **while** :

```
def somme(l):
    s = 0
    i = 0
    while i < len(l):
        s = s + l[i]
        i = i + 1
    return s
```

Il semblerait qu'elle renvoie la somme des éléments d'un tableau. Mais comment

le prouver ?

Définition (Invariant de boucle) Un **invariant de boucle** est une propriété mathématique, dont la véracité dépend des variables du programme, qui :

- ☞ si elle est vérifiée au début de l'exécution du corps de la boucle, elle reste est encore vérifiée à la fin de son exécution ;
- ☞ est vérifiée lors de l'entrée dans la boucle.

À nouveau, pour la première condition, on suppose que l'on ne sort pas de la boucle à l'aide d'un « `break` » ou un « `return` ».

Proposition - Invariant et sortie de boucle

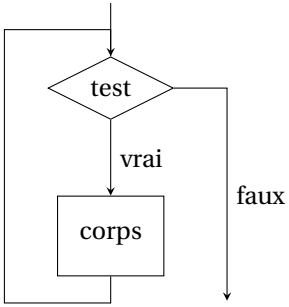
Un invariant de boucle est encore vérifié en sortie de boucle.

Preuve

Comme le test ne modifie pas les valeurs des variables et donc la véracité éventuelle de l'invariant, alors

- ☞ si l'invariant est vérifié en entrant dans la boucle,
- ☞ et si le corps de la boucle est tel que si l'invariant est vérifié au début de l'exécution du corps, il demeure vérifié à la fin,

alors l'invariant est vérifié à chaque fois que le test est exécuté, ainsi que juste après.



Exemple Montrons que la fonction `somme` précédente renvoie bien la somme des éléments d'un tableau.

Exercice 2 Montrer que la fonction suivante renvoie bien la factorielle de l'entier `n` passé en argument, en considérant un invariant de boucle approprié.

```python
def factorielle(n):
    f = 1
    i = 1
    while i < n:
        i += 1
        f *= i
    return f
```

Exercice 3 Prouver la correction de l'algorithme d'Euclide que l'on a programmé ainsi :

```python
def pgcd(a, b):
    x = a
    y = b
    while y > 0:
        z = x % y
        x = y
        y = z
    return a
```

Exercice 4 On considère la fonction suivante :

```python
def mystere(a, b):
    r = 1
    e = b
    p = a
    while e > 0:
        if e % 2 == 1:
            r = r * p
        p = p * p
        e = e // 2
    return r
```

1. Déterminer ce que fait cette fonction.
2. Le prouver.

IV. Étude de la recherche dichotomique

Nous allons, pour finir, étudier un algorithme **fondamental** (et plus délicat qu'il n'y paraît) : celui de la recherche dichotomique dans un tableau **trié**.

1. Une première version

On considère la fonction suivante :

```python
def dichoto(t, e):
    a = 0
    b = len(t) - 1
    while a <= b:
        c = (a + b) // 2
        if t[c] == e:
            return True
        if t[c] < e:
            a = c + 1
        else:
            b = c - 1
    return False
```

Question 1 Montrer que la fonction `dichoto` termine.

Question 2 Montrer que l'appel de fonction `dichoto(t, e)` renvoie `True` si et seulement si la valeur `e` apparaît dans le tableau **trié** `t`. On pourra utiliser comme invariant de boucle

$$\texttt{e} \in \texttt{t} \implies \exists k \in [\![\texttt{a}, \texttt{b}]\!] : \texttt{e} = \texttt{t}[k]$$

Question 3 Déterminer la complexité temporelle dans le pire des cas de la fonction.

Nous allons maintenant étudier plus précisement le comportement de cette fonction (en distinguant les succés et les échecs), en utilisant comme mesure de complexité temporelle le nombre de tests, et en déterminant la complexité *moyenne*.

On suppose dans un premier temps que le tableau `t` a comporte $2^n - 1$ éléments.

Question 4 Montrer que le nombre moyen de comparaisons effectuées en cas d'échec est $\frac{7}{2}n + 1$. Comment modifier simplement le programme pour n'en faire que $\frac{5}{2}n + 1$?

Question 5 En tenant compte de la modification précédente, montrer que le nombre moyen de comparaisons en cas de succés est égal à

$$\frac{\sum_{k=0}^{n-1} 2^k \times \frac{5k+6}{2}}{\sum_{k=0}^{n-1} 2^k} = \frac{5 \times 2^{n-1} n}{2^n - 1} - 2.$$

On considère maintenant la fonction suivante :

```python
def dichoto2(t, e):
    if len(t) == 0:
```

```
        return False
    a = 0
    b = len(t)
    while a + 1 < b:
        c = (a + b) // 2
        if t[c] <= e:
            a = c
        else:
            b = c
    # b == a + 1
    return t[a] == e
```

Question 6 Montrer que l'appel `dichoto2(t, e)` renvoie de même `True` si et seulement si la valeur `e` apparaît dans le tableau `t`, en utilisant un invariant adapté.

Question 7 Déterminer le nombre moyen de comparaisons effectuées lors d'une recherche infructueuse dans un tableau, en supposant cette fois qu'il est de taille 2^n.

Question 8 Faire de même pour une recherche fructueuse, et comparer avec la fonction `dichoto`.

Question 9 Comment étendre l'étude au cas où la longueur du tableau n'est pas de la forme 2^n (ou $2^n - 1$)?