Leçon d'informatique : Terminaison, correction et complexité des algorithmes

S. Benlhajlahsen - PCSI$_1$

Sommaire

Extrait du programme :

Thèmes	Exemples d'activité, au choix du professeur et non exigibles des étudiants. Commentaires
Terminaison. Correction partielle. Correction totale. Variant. Invariant.	La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête, la correction est totale si elle est partielle et si l'algorithme termine. On montre sur plusieurs exemples que la terminaison peut se démontrer à l'aide d'un variant de boucle. Sur plusieurs exemples, on explicite, sans insister sur aucun formalisme, des invariants de boucles en vue de montrer la correction des algorithmes.
Complexité.	On aborde la notion de complexité temporelle dans le pire cas en ordre de grandeur. On peut, sur des exemples, aborder la notion de complexité en espace

Introduction : Dans cette leçon, on tente pour un algorithme de répondre à trois questions :

- Est-que l'algorithme calcule bien ce qu'on veut ? C'est la **correction**.
- Est-que l'algorithme fait un nombre fini d'opération ? C'est la **terminaison**.
- Comment évolue le temps de calcul si on modifie les variables ? C'est la **complexité temporelle**.

I Terminaison

I.A Un exemple simple

Exemple 1 : On considère la suite d'instructions de la figure 1. Notons n_k la valeur prise par la variable `n` à la fin de la k-ième itération et on pose $n_0 = $ `n`.
À chaque itération, on enlève 2 à `n` donc au bout de k itérations, $n_k = n - 2k$. L'algorithme s'arrête dès que $n_k \leqslant 0$, c'est-à-dire pour $2k \geqslant n$, soit encore $k \geqslant \frac{n}{2}$. Le nombre d'itérations est alors $p = \lceil \frac{n}{2} \rceil$.
On vient de prouver que le nombre de tour de boucle est finie, ce qui veut dire que l'algorithme **se termine**.

```
1 n = 256
2 while n>0:
3     n = n-2
```

FIGURE 1

```
1 n = 256
2 while n>0:
3     n = n//2 # quotient de n par 2
```

FIGURE 2

I.B Définitions

Terminaison et variant de boucle : Un programme itératif est constitué d'une boucle `for` ou `while`. On cherche à se démontrer que cette boucle se termine.

On considère alors un **variant de boucle**, par exemple un entier naturel qui décroît à chaque itération de la boucle et qui atteindra 0 à un moment ce qui permet de démontrer que la boucle se termine.

Remarque : On rappelle qu'une suite décroissante et minorée converge vers sa borne inférieure. En particulier, une suite décroissante et positive converge.

Exemple 1 : Dans l'exemple 1, la suite (n_k) est un variant de boucle. On a montré que cette suite décroît et s'arrête lorqu'elle atteint 0.

Exemple 2 : À chaque itération, `n` est au moins divisé par 2, donc au bout de k itérations, $n_k \leqslant \frac{n}{2^k}$.
L'algorithme s'arrête lorsque $n_k < 1$ car, dans ce cas, $n_k = 0$ puisque c'est un entier naturel. Cette condition est réalisée dès que $\frac{n}{2^k} < 1$, c'est-à-dire pour $2^k > n$ et donc $k > \log_2(n)$.
Le nombre p d'itérations est donc inférieur à $\lfloor \log_2(n) \rfloor + 1$.
On a donc montré que le programme s'arrête après un nombre fini d'opérations : on a prouvé la terminaison de l'algorithme.

I.C Cas d'un algorithme récursif

Exemple de la factorielle récursive : On donne en figure 3 l'algorithme de calcul de $n!$.
Pour prouver la terminaison de l'algorithme, il faut prouver que le nombre d'appels récursifs est fini. On définit encore une fois une suite positive décroissante appelée variant de boucle. Ici, le variant de boucle est n qui décroît d'une unité à chaque appels et finit par atteindre 0 qui est la condition d'arrêt. Ainsi, si $n \geqslant 0$, le programme se termine.

```
1 def fact(n):
2     if n==0:
3         return 1
4     else:
5         return n*fact(n-1)
```

FIGURE 3

II Correction

II.A Exemple de l'exponentiation naïve

Exemple de la l'exponentiation naïve : On considère l'algorithme de la figure 4. On souhaite prouver par récurrence la proposition :

$$\mathcal{P}_n \text{ : le programme renvoie } x^n \text{ pour } n \geqslant 0$$

- **Initialisation** : pour $n = 0$, la boucle `for` est ignorée et le programme renvoie $1 = x^0$.
- **Hérédité** : Supposons la proposition $\mathcal{P}_n$ est vraie donc le programme renvoie x^n. On notera d'ailleurs que l'on fait n tours de boucles donc `p=1` est multiplié n fois par x ce qui correspond bien au calcul de x^n.
  Pour $n+1$, le programme calcule `p*x` ce qui correspond à $x^n \cdot x = x^{n+1}$. La proposition $\mathcal{P}_{n+1}$ est vraie.
- **Conclusion :** on a montré par récurrence que le programme renvoie bien x^n.

```
1 def expo_naive(x, n):
2     p = 1 # p comme produit
3     for i in range(1,n+1):
4         p = p*x
5     return p
```

FIGURE 4

Remarque : La propriété $\mathcal{P}_n$ est appelée dans la suite **invariant de boucle**.

II.B Définitions

À retenir : Prouver la correction d'un algorithme revient à prouver que cette algorithme effectue bien la tâche souhaitée. La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête, la correction est totale si elle est partielle et si l'algorithme termine.

Méthode pour un algorithme itératif : Pour un algorithme itératif, on peut montrer par récurrence la correction. La propriété $\mathcal{P}_n$ qui sera vraie à chaque tour de boucle sera appelée **invariant de boucle**.

II.C Autres exemples

Recherche du maximum d'une liste : On considère l'algorithme de la figure 5 qui renvoie l'indice du maximum d'une liste. Pour montrer la correction, on considère l'invariant de boucle :

$$\mathcal{P}_k \ : \ \texttt{p} \text{ est l'indice du maximum de } \texttt{L[0:k+1]}$$

- **Initialisation** : Pour $k = 0$, on ne rentre pas dans la boucle `for`. La liste `L[0:0+1]` est composée uniquement du premier élément et la fonction renvoie $p = k = 0$ donc $\mathcal{P}_0$ est vraie.
- **Hérédité** : Supposons $\mathcal{P}_k$ vraie. Au tour de boucle suivant, on compare `L[k+1]` à `L[p]` et on conserve le maximum. De ce fait, `p` sera l'indice du maximum de `L[:k+2]`. La propriété $\mathcal{P}_{k+1}$ est donc vraie.
- **Conclusion** : on a montré par récurrence la correction du programme.

Remarque : cette correction est partielle et totale puisque l'algorithme se termine.

```python
def maximum(L):
    n = len(L)
    p = 0
    m = L[0]
    for k in range(1,n):
        if L[k]>m:
            p = k
    return p
```

FIGURE 5

```python
def expo_rapide(x, n):
    if n==0:
        return 1
    else:
        res = expo_rapide(x,n//2)
        if n%2==0: #n pair
            return res**2
        else: #n impair
            return x*res**2
```

FIGURE 6

Exponentiation rapide - version récursive : On considère le programme de la figure 2. On souhaite prouver par **récurrence forte** la proposition :

$$\mathcal{P}_n \ : \text{le programme renvoie } x^n \text{ pour } n \geqslant 0$$

- **Initialisation :** $\mathcal{P}_0$ est vraie puisque c'est la condition d'arrêt de la boucle.
- **Hérédité :** Supposons $\mathcal{P}_0$, $\mathcal{P}_1$,..., $\mathcal{P}_n$ vraies. Pour $n+1$, deux cas peuvent se présenter :
  - $n+1$ est pair. Dans ce cas, le programme renvoie `expo_rapide(x, (n+1)//2)**2` qui correspond à $\left(x^{\left\lfloor \frac{n+1}{2} \right\rfloor}\right)^2 = x^{n+1}$ car $n+1$ est pair et la propriété $\mathcal{P}_{\left\lfloor \frac{n+1}{2} \right\rfloor}$ est supposée vraie ;
  - $n+1$ est impair. Dans ce cas, le programme renvoie `x*expo_rapide(x, (n+1)//2)**2` qui correspond à $x \cdot \left(x^{\left\lfloor \frac{n+1}{2} \right\rfloor}\right)^2 = x \cdot \left(x^{n/2}\right)^2 = x^{n+1}$ car $n+1$ est impair et la propriété $\mathcal{P}_{\left\lfloor \frac{n+1}{2} \right\rfloor}$ est supposée vraie.
- **Conclusion :** On a montré par récurrence forte la correction de l'algorithme.

Remarque : la correction est totale car l'algorithme se termine.

III Complexité temporelle

À retenir : On va estimer la performance d'un algorithme via le nombre d'opérations qu'il nécessite.

III.A Factorielle

Exemple de la factorielle Considérons l'exemple de la figure 7. On va énumérer le nombre d'opérations effectuées lors d'un appel de la fonction.

- une affectation ligne 2,
- la génération de la liste des valeurs de la variable de boucle ligne 3,
- à chaque tour de boucle, une affectation à `i` et une multiplication suivie d'une affectation ligne 4,
- le renvoi vers l'extérieur ligne 5.

Sachant qu'il y a n tours de boucles, en supposant que chaque opération a la même durée[a], cela fait $3n+3$.
Si n est petit, le temps d'exécution est tellement faible que ceci n'a pas d'importance. C'est donc seulement ce qui se passe pour n grand qui nous intéresse. Alors :

- On peut approximer $3n+3$ à $3n$, ce qui revient à négliger les quelques opérations ayant lieu en dehors de la boucle.
- Si on veut juste un ordre de grandeur pour avoir une idée du temps de calcul, le coefficient devant n n'est pas très utile non plus.

a. Ce qui est une approximation grossière ! Une simple affectation et l'exécution de `range` n'ont rien à voir !

Cela revient à dire que le nombre d'opération "est un polynôme du premier degré en n" ou qu'il est "de l'ordre de grandeur de n". Cela revient aussi à compter simplement les tours de boucle ! Donc c'est la seule ligne 3 qui contient l'information utile pour évaluer la complexité.

```
1 def factorielle(n) :
2     facto = 1
3     for i in range(1, n+1) :
4         facto *= i
5     return facto
```

FIGURE 7 – Factorielle

À retenir : Pour beaucoup d'algorithmes qu'on vous demandera de coder, les programmes seront courts, avec peu d'instructions par tour de boucle. Donc le point clef pour évaluer la complexité sera simplement le nombre de tours de boucles. Rappelons qu'il s'agit d'une approximation souvent optimiste !

Remarque Quand le nombre d'opérations est de l'ordre de n, on dit que "la complexité de l'algorithme est en $O(n)$" ou encore que "la complexité de l'algorithme est linéaire".

III.B Test de primalité

Une manière très directe de déterminer si un entier est premier est donnée en figure[1].

Décortiquons son fonctionnement

1. La boucle parcourt les diviseurs potentiels. Sachant qu'un diviseur de n ne peut pas être supérieur à $\sqrt{n}$ (sauf n lui-même, qui ne nous intéresse pas), la boucle peut se limiter à $k \leqslant \sqrt{n}$ (ligne 6).
2. On calcule le reste de la division euclidienne de n par k. S'il est nul (ligne 7), alors n n'est pas premier. On interrompt alors la boucle en sortant de la fonction avec `return`
3. Si la boucle a pu arriver à son terme, alors on n'a trouvé aucun diviseur, donc n est premier et on peut renvoyer `True`.

Conclusion : Déterminons la complexité de cet algorithme, toujours en supposant n grand. La boucle fait jusqu'à $\sqrt{n}-1$ tours, disons $\sqrt{n}$. Cette valeur est atteinte si n est premier, sinon la boucle sera interrompue plus tôt. Cet algorithme est donc "au pire" en $O\left(\sqrt{n}\right)$.

Remarque Notez que cet algorithme, bien que "brutal", est déjà un peu malin :

- Grâce au return ligne 8, $O\left(\sqrt{n}\right)$ est le "pire" scénario ;

1. Rappelons que, suivant la définition moderne d'un nombre premier, 1 n'est pas premier.

```python
import math as m
# Renvoie True si n>3 est premier , False sinon.
def test_primalite(n) :
    racine = round(m.sqrt(n))
    for k in range(2, racine+1) :
        if n%k == 0 :
            return False
    return True
n = 42 # valeur exemple
est_premier = test_primalite(n)
```

FIGURE 8 – Test de primalité.

- Si on n'avait pas utilisé l'astuce que les diviseurs non triviaux sont forcément inférieurs à $\sqrt{n}$, la complexité aurait été en $O(n)$.

Remarque Est-ce que $O(n)$ est vraiment si mauvaise com parée à $O\left(\sqrt{n}\right)$? Fin 2018, il a été prouvé que $2^{82\,589\,933} - 1$ était premier[a]. La différence entre ce nombre et sa racine est gigantesque !

a. Il appartient à la famille des nombres de Mersenne. Inutile de préciser qu'on a utilisé un autre algorithme…

Remarque Voilà un cas où, en pratique, supposer que les opérations usuelles se font en une seule instruction est rarement une bonne approximation. En cryptographie, les entiers manipulés sont énormes, nécessitant des traitements spéciaux même pour des opérations aussi simples qu'une addition ou une comparaison. Ces opérations peuvent alors nécessiter beaucoup d'instructions chacune.

III.C Boucles imbriquées indépendantes

Exemple Deux (ou plus) boucles `for` imbriquées est une situation courante. Maintenant qu'il est clair que le détail des instructions dans les boucles n'est pas important pour l'estimation de la complexité, regardons le programme minimal de la figure 9 (la ligne 3 ne joue aucun rôle ici).

```python
for i in range(n) :
    for j in range(m) :
        a = i*j
```

FIGURE 9 – Boucles imbriquées indépendantes.

```python
for i in range(n) :
    for j in range(i+1) :
        a = i*j
```

FIGURE 10 – Boucles imbriquées non indépendantes.

Analyse de la complexité Ces deux boucles sont dites *indépendantes* car le nombre de tours de la boucle intérieure est le même à chaque tour de la boucle extérieure. La complexité est donc immédiatement en $O\left(n \cdot m\right)$, ce qu'on appelle une complexité **quadratique**. Ce terme est plus facile à comprendre en prenant le cas particulier $n = m$ car la fonction $n \mapsto n^2$ est dite **quadratique**.

III.D Boucles imbriquées non indépendantes

Autre exemple On considère à présent le code de la figure 10. Pour la boucle extérieure, `i` parcourt la liste $[\![0,n-1]\!]$. À chaque tour de cette boucle, la boucle intérieure tourne `i+1` fois[a]. D'où le nombre total N de tours de boucles :

$$N = \sum_{i=0}^{n-1} \sum_{j=0}^{i} 1 = \sum_{i=0}^{n-1} (i+1)\,\frac{n \cdot (n+1)}{2} + n$$

On peut arrêter là le calcul : ceci est un polynôme du second degré en n, donc la complexité est encore **quadratique**. La conclusion est que, par rapport au cas précédent, on a fait moins de tours de boucles, mais la différence devient

négligeable quand n est grand.

a. Le `+1` n'est pas vraiment important pour discuter la complexité, mais il permet d'éviter d'avoir range(0) au premier tour de la boucle extérieure. Ce n'est pas faux en soi, mais donne zéro tour de boucle.

IV Complexités usuelles

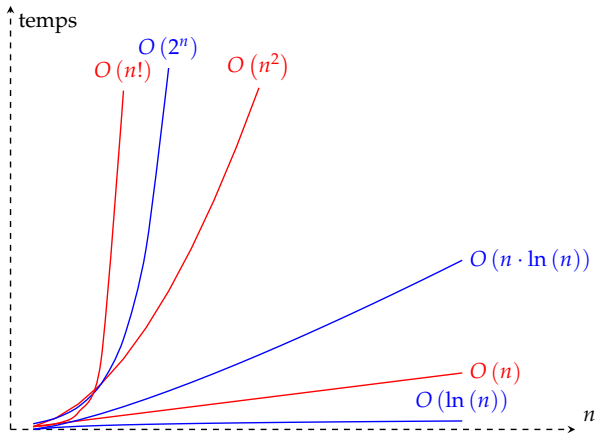

FIGURE 11 – Comparaison de quelques complexités à grand n.

Les complexités typiquement considérées comme "faibles" sont :

- en temps constant : $O(1)$;
- en temps logarithmique : $O(\ln(n))$: algorithme de recherche dichotomique d'un élément dans une liste ;
- en temps sous-linéaire : $O(n \cdot \ln(n))$;
- en temps linéaire : $O(n)$: boucle `for`, algorithme itératif, recherche linéaire.

Complexités intermédiaires : Viennent ensuite les complexités intermédiaires, dites polynômiales, de la forme $O(n^p)$ avec p un entier (on a vu la complexité quadratique pour $p = 2$).

Complexités élevées Arrivent ensuite les complexités considérées comme plus élevées, que l'on essaie d'éviter (quand on peut !) :

- en temps exponentiel : $O(2^n)$;
- en temps factoriel : $O(n!)$.

Allure La figure 11 montre les allures comparées de quelques complexités usuelles. Comme on l'a déjà expliqué, ces allures supposent que le coefficient en facteur est de l'ordre de 1.

Remarque : Si on prend l'algorithme de recherche linéaire en figure 12, on constate en réalité trois complexités différentes :

- l'élément `a` peut être présent au début de la liste `L`, on sort alors de la boucle `for` après un nombre d'opérations indépendant de la taille de la liste. C'est la complexité **dans le meilleur des cas** qui est, ici, en $O(1)$ (coût constant) ;
- l'élément `a` peut être présent en fin de liste `L`, on va alors la parcourir entièrement. On aura une complexité **dans le pire des cas** qui est, ici, en $O(n)$;
- On pourrait tester cette algorithme pour une multitude de liste de même taille n. On évaluerait alors une complexité **moyenne** qui est ici aussi en $O(n)$.

Sauf mention contraire, le programme demande de se concentrer sur la **complexité dans le pire des cas**.

```
1 def recherche(L, a):
2     for x in L:
3         if x==a:
4             return True
5     return False
```

FIGURE 12