I_{02} Informatique théorique : terminaison de boucle, invariant de boucle et complexité.

PCSI 2023 – 2024

Introduction : Lorsque l'on écrit un algorithme, on souhaite plusieurs choses :

1. Que cet algorithme se termine (pas de boucle infini)
2. Qu'il produise le résultat attendu (algorithme correct)
3. Qu'il soit le plus rapide possible et nécessite le moins de mémoire possible.

Ces problèmes sont directement lié à l'algorithme et pas à son implémentation dans un langage particulier. En général, on procède en effectuant des tests sur l'algorithme, mais lorsque le coût humain ou financier d'une erreur est trop important, on aimerait pouvoir démontrer que l'algorithme est correct.

I Terminaison d'un algorithme : variant de boucle

On s'intéresse ici à la terminaison des boucles. Il n'y a pas de problème avec les boucles `for` dont le nombre d'itération est connu à l'avance. Le problème apparait avec les boucles while pour lesquelles on ne sait pas a priori si la condition de terminaison sera vérifiée.

Exemple :

```
def f(n): #n un entier
    assert (type(n) == int and n >= 0)
    compteur = 2
    produit = 1
    while n-compteur >= 0:
        produit = produit*compteur
        compteur += 1
    return produit
```

1. Le programme précédent se termine-t-il toujours, quelque soit l'entrée ?
2. Quel est son but ?

Oui, si on n'a pas un entier positif, on retourne none, si l'entier est inférieur à 1 on ne rentre pas dans la boucle while et s'il est plus grand que 2, on considère $i = n - compteur$ c'est un entier qui décroit strictement à chaque itération.
Son but est de calculer $n!$

Terminaison de boucle : Un moyen usuel de démontrer la terminaison d'une boucle est de trouver une quantité, fonction des paramètres de la boucle, qui est un entier positif et qui décroit strictement à chaque itération. Cette quantité est appelée variant de boucle.

Exercice : Écrire un algorithme de recherche dichotomique dans un tableau trié et proposé un variant de boucle pour démontrer que votre algorithme se termine.
indice fin- indice deb, cf programme dicho_variant_de_boucle.py
J'appelle L_n la taille du tableau à la n^e itération. Je veux montrer que $L_{n+1} < L_n$.
$L_n = fin_n - deb_n$

— soit je trouve le bon nombre, je suis heureux et l'algo se termine, pas de soucis
— soit $fin = m - 1$ (et deb inchangé)
— soit $deb = m + 1$ (et fin inchangé)

On va avoir besoin d'un encadrement de $m = E\left(\frac{deb+fin}{2}\right)$ (avec E la fonction partie entière), et on a les 2 derniers cas à gérer
$x - 1 < E(x) \leqslant x$

• Soit $deb_{n+1} = m + 1$ et $fin_{n+1} = fin_n$
$L_{n+1} = fin_{n+1} - deb_{n+1} = fin_n - m - 1$
Or $x - 1 < E(x)$ d'où $-E(x) < 1 - x$ donc
$L_{n+1} < fin_n + (1 - \frac{deb_n+fin_n}{2}) - 1 = \frac{fin_n-deb_n}{2} = \frac{L_n}{2}$
Or $L_n \geqslant 0$ (sinon on arrête l'algorithme donc $\frac{L_n}{2} \leqslant L_n$)
$L_{n+1} < \frac{L_n}{2} \leqslant L_n$ On en déduit $\boxed{L_{n+1} < L_n}$

• Soit $fin_{n+1} = m - 1$ et $deb_{n+1} = deb_n$
$L_{n+1} = fin_{n+1} - deb_{n+1} = m - 1 - deb_n$ Or $E(x) \leqslant x$ d'où $m - 1 - deb_n \leqslant \frac{deb_n+fin_n}{2} - 1 - deb_n = \frac{fin_n-deb_n}{2} - 1 = \frac{L_n}{2} - 1 \leqslant L_n - 1$ car $L_n \geqslant 0$ comme plus haut
On a donc $\boxed{L_{n+1} \leqslant L_n - 1 < L_n}$

• L_n est donc un entier (différence entre 2 entiers) qui décroit strictement à chaque itération. La boucle s'arrête si $L_n < 0$, ce qui arrivera forcément un jour.

Remarques :
— Il existe des moyens plus complexes de démontrer qu'une boucle se termine en faisant intervenir plusieurs expressions, nous ne les verrons pas cette année.
— Il est parfois impossible de démontrer qu'une boucle se termine même si on le vérifie toujours expérimentalement (exemple : suite de syracuse)

II Correction d'un algorithme : invariant de boucle

Une fois que l'on a démontré qu'un algorithme se termine effectivement, il serait agréable de savoir si celui-ci produit le résultat attendu. Par exemple une fonction qui se termine tout le temps pourrait être

```
1 def factorielle(n):
2     return 1 #retourne n! , mais ne marche que pour 0 et 1
```

Cette fonction se termine tout le temps mais ... n'est pas très intéressante.

Les instructions simples ne posent pas de problème. Les problèmes arrivent en général lorsque l'on a des boucles. Pour ce paragraphe, on n'écrira que des boucles while (on peut toujours ré-écrire une boucle for en boucle while) avec un seul point d'entrée (pas d'instruction du type « go to » qui n'existe de toutes façons pas en python) et un seul point de sortie (pas de break ou de return au milieu de la boucle).

Exercice : Montrer par récurrence que le terme général de la suite définie par $u_0 = 2$ et $u_{n+1} = \frac{1}{3}u_n$ peut s'écrire $u_n = \frac{2}{3^n}$
u_0 s'écrit bien comme on veut. Si c'est vrai au rang n alors c'est vrai au rang $n + 1$ donc c'est vrai pour tout n

En informatique, on procède en général d'une façon similaire :

Invariant de boucle : Pour démontrer qu'un algorithme produit le résultat attendu, on utilise un invariant de boucle , c'est-à-dire une propriété qui :

- **est vérifiée avant d'entrer dans la boucle**
- **si cette propriété est vérifiée avant une itération, elle est vérifiée après celle-ci**
- **lorsqu'elle est vérifiée en sortie de boucle, elle permet (combinée avec la condition de sortie de boucle) d'en déduire que le programme produit le résultat attendu.**

```
def puissance(k,n):# calcule k^n en supposant n > 0
    c=n
    p=1
# invariant de boucle : p = k^(n-c) et c >= 0  ; condition d'entrée dans la boucle c > 0
    while c>0: #  p = k^(n-c) et c >= 0 et c > 0
        p=k*p  #  p = k^(n-c+1) = k^(n-(c-1)) et c > 0
        c=c-1  #  p = k^(n-c) et c >= 0
    # (p = k^(n-c) et c >= 0) et non(c > 0)
    return p # (p = k^(n-c) et c >= 0) et c <= 0 ⇒ p = k^n
```

Remarques :

- Raisonnement analogue aux raisonnement par récurrence :
  1. On vérifie que l'invariant de boucle est vrai avant d'entrer dans la boucle ($\mathcal{P}_0$ est vrai)
  2. On montre que s'il est vrai au début d'une itération, alors il est vrai à la fin ($\mathcal{P}_k \Rightarrow \mathcal{P}_{k+1}$)
  3. À la fin de la boucle $\mathcal{P}_{k_{max}}$ est vrai et la condition d'entré dans la boucle est fausse (sinon on ne serait pas sorti)
- Toute la difficulté est de trouver le bon invariant de boucle.

Autre exemple :

```
def puissance_rapide(k,n):
    c=n
    p=1
    i=k
    # p × i^c = k^n et c >= 0
    while c>0:       # p × i^c = k^n et c >= 0 et c > 0
        if c%2==0: # p × (i^2)^(c/2) = k^n et c > 0
            i=i**2 # p × (i)^(c/2) = k^n et c > 0
            c=c//2 # p × (i)^c = k^n et c >= 0
        else:      # p × i * i^(c-1) = k^n et c > 0
            p=p*i  # p × i^(c-1) = k^n et c > 0
            c=c-1  # p × i^c = k^n et c > 0
    # p × i^c = k^n et c >= 0 et non(c > 0) ⇒ p × i^0 = p = k^n
    return p
```

Exercice : Démontrez à l'aide d'un invariant de boucle que la fonction suivante retourne $n!$ pour tout $n \geqslant 2$

```
def factorielle(n):
    p=1
    k=1
    # p = k! et n - k >= 0
```

```
    while n-k>0:# p = k! et n − k − 1 ⩾ 0
        k=k+1# p = (k − 1)! et n − k ⩾ 0
        p=p*k# p = k! et n − k ⩾ 0
    # p = k! et n − k >= 0 et n − k <= 0 donc p = n!
    return p
```

Remarque : Lorsque l'on fait une démonstration à l'aide d'un invariant, on montre que si le programme se termine, alors le résultat est correct. On parle alors de correction partielle. Si on démontre de plus que le programme termine (variant de boucle), alors on parle de correction totale.

III Temps d'exécution : complexité en temps

Nous avons maintenant un algorithme qui produit le résultat demandé et qui se termine ... un jour. Mais si l'algorithme se termine dans 10000 ans, il est correct, mais peu intéressant. On va maintenant s'intéresser à la durée d'exécution du programme ou plus précisément à « comment la durée d'exécution augmente lorsque la taille des données augmente ».

Plusieurs problèmes se posent en fait, le temps exact d'exécution dépend de beaucoup de choses :

- langage dans lequel l'algorithme est implémenté (et en particulier si le langage est compilé ou interprété)
- processeur et en particulier fréquence du processeur et propriétés de sa mémoire cache
- si d'autres programmes s'exécutent en même temps
- des données exactes que l'on utilise pour tester l'algorithme

On va donc essayer de caractériser l'efficacité d'un algorithme indépendamment de son implémentation en comptant le nombre d'opérations élémentaires nécessaires à son exécution. (On va donc éviter d'utiliser les fonctions « avancées » de python comme `min` ou `in` qui peuvent cacher un nombre important d'opérations)

Les types de données de bases pour nous seront les entiers, les nombres flottants et les caractères de taille limitée (par exemple 64 bits). Les opérations de bases pour nous seront :

- addition, soustraction, multiplication...
- affectation, accès à un élément d'un tableau...
- test simple ($<$; $>$; $==$...)
- renvoi d'une valeur pour une fonction

Ces opérations de bases ont normalement un faible temps d'exécution (quelques cycles d'horloge du processeur), nous allons faire comme si elles ont le même temps et ce sera notre unité de base. On s'intéresse ensuite au comportement asymptotique lorsque la taille des données traitée devient grande. Si l'on note la taille des données n et que le nombre d'opérations de base exécutées par l'algorithme est $3n^2 + 2n + 20$ alors :

- seul le terme de plus grand ordre $3n^2$ est réellement significatif et traduit le temps que mettra l'algorithme à s'exécuter lorsque les données sont de taille un peu importante ;
- le préfacteur 3 devant le n^2 n'est pas très intéressant , en effet, toutes les opérations de bases n'ont pas réellement le même temps d'exécution et il dépend en particulier de la machine exacte sur laquelle on fait tourner l'algorithme ;
- on se limite donc à dire que l'algorithme a un temps d'exécution qui croit de la même façon que n^2 et on parle de complexité en temps.

Définition : On dit qu'un algorithme a une complexité en $O(f(n))$ si son cout pour une donnée de taille n divisé par $f(n)$ est borné lorsque n tend vers l'infini.

Complexité	Nom	Ordre de grandeur de temps pour $n = 10^6$	Remarques
$O(1)$	temps constant	1 ns	Le temps ne dépend pas de la taille des données. Très rare.
$O(\log n)$	logarithmique	10 ns	Exécution quasi instantanée, algorithme extrêmement rapide.
$O(n)$	linéaire	1 ms	Exécution, très rapide. Problème de mémoire avant que le temps d'exécution soit un problème.
$O(n^2)$	quadratique	15 min	Commence à devenir problématique lorsque la taille des données est grande.
$O(n^k)$	polynomiale	30 ans ($k = 3$)	
$O(2^n)$	exponentielle	$> 10^{300000}$ milliards d'années	inutilisable sauf pour de très très petites données.

Règles simples :

- Affectation, accès à élément d'un tableau, opération arithmétique (opérations de bases) : $O(1)$
- Instruction If-Then-Elif-Else : le cout est égal au plus grand des couts dans les différentes branches.

- Séquence d'opérations : l'opération la plus couteuse domine
- Boucle simple exécuté m fois : m fois le cout du corps de la boucle

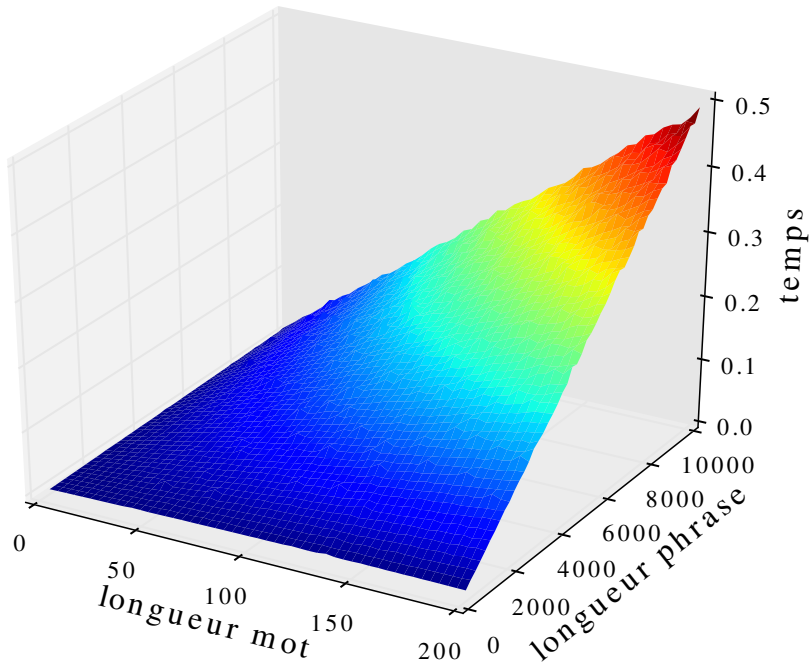
Exemples :

1. Écrire un algorithme de recherche de minimum et évaluer sa complexité en fonction de la taille du tableau.
2. Écrire un algorithme naïf qui prend en entrée un entier positif n et qui renvoie la somme des entiers positifs inférieurs à n. Quelle est sa complexité en n ?
3. À l'aide de vos connaissances en mathématiques, écrivez un programme qui fait la même chose que le précédent mais en temps constant. (D'où l'utilité de réfléchir avant de programmer).
4. On considère le programme suivant :

```
def f(n): #n un entier
    for i in range(1,n):
        for j in range(1,n):
            print(i*j,end='\t')
        print('\n')
```

   Que fait ce programme ? Quel est sa complexité ?
5. écrire un algorithme naïf de recherche d'un mot de taille m dans une phrase de taille p et estimer sa complexité en fonction de p et de m.

   Est-ce cohérent avec le graphique suivant qui présente le temps de calcul mesuré en fonction de la taille des mots et des phrases pour des données aléatoires ?



6. Écrire un algorithme de recherche dichotomique dans un tableau. Quelle est sa complexité en fonction de la taille du tableau ?

la dichotomie se démontre par récurrence : la taille du sous tableau considérée à l'étape k est $fin - deb \leqslant \frac{n}{2^k}$
À l'étape 0, on a bien un tableau de taille n
On a ensuite deux cas :

1. soit $deb = m + 1$
2. soit $fin = m - 1$

Dans le cas 1. on a à la fin $fin' - deb' = fin - ((fin + deb)//2) - 1$
or

$$(fin + deb)//2 + 1 >= (fin + deb)/2$$
$$-(fin + deb)//2 - 1 <= -(fin + deb)/2$$
$$fin - (fin + deb)//2 - 1 <= fin - (fin + deb)/2 = (fin - deb)/2 \leqslant \frac{1}{2}\frac{n}{2^k}$$
$$fin' - deb' \leqslant \frac{n}{2^{k+1}}$$

Dans le cas 2. on a à la fin $fin' - deb' = ((fin + deb)//2 - deb$ or

$$(fin + deb)//2 < (fin + deb)/2$$
$$(fin + deb)//2 - deb < (fin + deb)/2 - deb = (fin - deb)/2 \leqslant \frac{1}{2}\frac{n}{2^k}$$
$$fin' - deb' \leqslant \frac{n}{2^{k+1}}$$

On a donc à chaque itération $fin - deb \leqslant \frac{n}{2^k}$.
De plus $1 \leqslant fin - deb \leqslant \frac{n}{2^k}$ d'où $2^k \leqslant n$ d'où $k \leqslant \log_2(n)$: on a une complexité logarithmique.

Remarques :

- Dans des cas comme celui de la dichotomie (boucle while en particulier), si l'on considère des données de taille fixée, il peut arriver que le temps de recherche dépende très fortement du jeu de donnée choisi (par exemple si on recherche l'élément qui est pile au milieu du tableau). On différencie alors la
  - la complexité dans le meilleur des cas (donne une borne inférieure, en général peu représentative)
  - la complexité dans le pire des cas (donne une borne supérieure)
  - Si l'on connait la fréquence d'apparition des données, on peut aussi calculer une complexité dans le cas moyen (beaucoup plus difficile)

  Pour nous cette année, on se limitera au pire des cas.
- On peut définir la complexité en espace de la même façon qui concerne la mémoire que nécessite le programme. Tous les raisonnement sont similaires. La complexité en mémoire est bornée par la complexité en temps car il faut au moins le temps de faire les affectations !
- Dans le cas des problèmes de petite taille, les pré-facteurs ont de l'importance.

Mesure du temps dans python : Pour mesurer le temps en python, on peut faire appel au module `time` et en particulier à la fonction `time.clock()` qui renvoie de façon précise le temps. En faisant des différences, on peut en déduire la durée d'exécution du programme ou d'une partie du programme.

Table des matières