

Performances d'un programme

Savoir si un algorithme est « satisfaisant » est une des préoccupations principales des développeurs. Derrière ce mot se cachent trois questions fondamentales en algorithmique :

- l'algorithme s'arrête-t-il après un nombre fini d'étapes ? C'est la **terminaison** ;
- renvoie-t-il le résultat attendu ? C'est la **correction** ;
- reste-t-il utilisable lorsque la taille des données augmente ? Autrement dit, est-il efficace ? C'est la question de la **complexité**.

1 Terminaison

Définition

Terminaison

Un algorithme *termine* pour une entrée donnée si son exécution s'arrête après un nombre fini d'étapes. Dire qu'il termine sans autre précision signifie généralement qu'il termine pour toute entrée qui respecte ses préconditions.

Sans boucle ni appel récursif [1], la terminaison est en général immédiate. Avec une boucle, il faut justifier qu'elle ne peut pas être répétée à l'infini, pour cela on regarde un variant de boucle.

1.1 Variant de boucle

Définition

Variant

Un **variant de boucle** est une quantité

- entière,
- positive ou nulle tant que la boucle peut recommencer,
- qui décroît strictement à chaque itération.

Une suite d'entiers naturels strictement décroissante ne peut pas être infinie : la boucle termine donc, à condition que le corps de chaque itération termine lui aussi.

La preuve comporte deux vérifications : la quantité choisie reste dans $\mathbb{N}$ aux instants considérés, puis sa valeur diminue **strictement** à chaque passage. Une quantité seulement décroissante, qui pourrait rester constante, ne suffit pas.
Un variant n'est pas nécessairement une seule variable. Si plusieurs variables évoluent, il faut parfois trouver une expression qui combine leurs valeurs.

1.2 Boucle `for`

Lorsque `range(a, b)` a des bornes entières fixées, la boucle parcourt $a, a+1, \ldots, b-1$ si $a < b$, donc exactement $b-a$ valeurs. Un variant au début de l'itération d'indice i est $b-i$: il est strictement positif et diminue de 1 à chaque tour. Si $a \geq b$, aucune itération n'a lieu.

```
1 def somme_entiers(n):       # precondition : n >= 0
2     s = 0
3     for i in range(1, n + 1):
4         s = s + i           # ajoute l'entier courant
5     return s
```

1. sujet d'un prochain chapitre

Ici, `range(1, n + 1)` fournit n valeurs si $n \geq 0$: la boucle termine. Plus généralement, une boucle `for` qui parcourt une liste finie termine si son corps termine.

Exemple

```
def compter_positifs(L):
    compteur = 0
    n = len(L)
    for i in range(n):
        if L[i] > 0:
            compteur = compteur + 1
    return compteur
```

Question 1 : Déterminer en justifiant le variant de boucle pour la boucle `for` de `compter_positifs(L)`.

Réponse 1 : On peut choisir $n - i$ au début de l'itération d'indice i. Tant que la boucle s'exécute, $0 \leq i < n$, donc $n - i$ est un entier strictement positif. Il diminue de 1 au passage à l'indice suivant.

1.3 Boucle `while`

Le nombre d'itérations n'est pas inscrit dans la syntaxe d'une boucle `while` ; le choix d'un variant de boucle n'est pas aussi immédiat. Il faut étudier la condition et les variables modifiées dans son corps.

```
def division_euclidienne(a, b):  # preconditions : a >= 0, b > 0
    q, r = 0, a
    while r >= b:
        q = q + 1
        r = r - b
    return q, r
```

Question 2 : Compléter le tableau d'évolution des variables de `division_euclidienne(10, 3)`.

État	q	r
Avant la boucle	0	10
Après la première itération	1	7
Après la deuxième itération	2	4
Après la troisième itération	3	1

Au début de chaque itération, $r \geq b > 0$. Après la soustraction, r reste positif ou nul et diminue strictement de b. La quantité r est donc un variant. À la sortie, $r < b$.

Remarque : Une précondition est indispensable ici : si $b = 0$, r ne diminue pas et la boucle peut ne jamais s'arrêter. Il faut annoncer les entrées pour lesquelles une preuve est valable.

Deux variantes d'une multiplication. Dans les deux fonctions ci-dessous, on suppose $b \in \mathbb{N}$. Elles calculent $a \times b$ par additions successives, mais leurs variants ne sont pas écrits de la même façon.

```
def multiplication_1(a, b):
    m = 0
    while b > 0:
        m = m + a
        b = b - 1
    return m
```

Question 3 : Pour `multiplication_1`, proposer un variant de boucle et vérifier qu'il est positif et strictement décroissant tant que la boucle s'exécute.
Réponse 3 : La variable b est un variant : elle est strictement positive tant que la condition de boucle est vraie et diminue de 1 à chaque itération. Comme b est initialement un entier naturel, la boucle termine.

```
def multiplication_2(a, b):
    i, m = 0, 0
    while i < b:
        m = m + a
        i = i + 1
    return m
```

Question 4 : Pour `multiplication_2`, proposer un variant de boucle. Pourquoi i ne convient-il pas ?
Réponse 4 : La quantité $b - i$ est un variant : elle est strictement positive au début d'une itération puisque $i < b$, et elle diminue de 1 lorsque i augmente de 1. En revanche, i augmente : ce n'est pas un variant.

2 Correction

Une boucle qui s'arrête peut tout de même renvoyer un résultat faux. On distingue donc l'arrêt du programme de la validité de son résultat.

Définition

Correction partielle et totale

Un algorithme est **partiellement correct** si, pour toute entrée respectant les préconditions, *lorsqu'il termine*, le résultat satisfait les spécifications. Il est **totalement correct** s'il est partiellement correct *et* s'il termine pour ces entrées.

2.1 Invariant de boucle

Définition

Invariant

Un **invariant de boucle** est une propriété des variables qui est vraie avant la première itération et qui reste vraie après chaque itération. Combinée à la condition de sortie, elle permet de démontrer le résultat annoncé.

Pour prouver qu'une propriété est un invariant, on suit trois étapes :

1. **Initialisation** : elle est vraie avant la boucle ;
2. **Conservation** : si elle est vraie avant une itération, elle l'est encore après ;
3. **Conclusion** : à la sortie de boucle, on utilise la condition d'arrêt de la boucle dans l'invariant pour démontrer que le résultat est celui attendu.

Remarque : Le variant et l'invariant jouent des rôles différents : le premier prouve que la boucle *s'arrête*, le second explique *ce qu'elle calcule*.

2.2 Exemple avec une boucle `for` : somme des entiers

Reprenons la fonction `somme_entiers(n)` avec $n \geq 0$. On souhaite obtenir $1+2+\cdots+n = \dfrac{n(n+1)}{2}$.

```
def somme_entiers(n):       # precondition : n >= 0
    s = 0
    for i in range(1, n + 1):
        s = s + i           # ajoute l'entier courant
    return s
```

On rappelle que $1+2+\cdots+m=\dfrac{m(m+1)}{2}$ [2]. Avant l'itération d'indice i, la propriété suivante est un invariant :

$$s=\sum_{k=1}^{i-1}k=\frac{(i-1)i}{2}.$$

- Initialisation : Pour $i=1$, la somme est vide et $s=0$.
- Conservation : Si l'égalité est vraie avant une itération, l'instruction `s = s + i` donne $s=\sum_{k=1}^{i}k$: elle sera donc vraie au début de l'itération suivante.
- Conclusion : La condition d'arrêt de boucle est $i=n$. Dans ce cas on a bien $s=\sum_{k=1}^{n}k=\dfrac{n(n+1)}{2}$, ce qui est le résultat attendu. *Pour $n=0$, la boucle est vide et la formule reste vraie.*

Exemple

Autre exemple : puissance de deux

```
1 def puissance_de_deux(n):     # precondition : n >= 0
2     p = 1
3     for i in range(n):
4         p = 2 * p
5     return p
```

Après k itérations, $p=2^k$, c'est notre invariant de boucle.

- Initialisation : pour $k=0$, on a $2^0=1$, ce qui est bien le cas.
- Conservation : si l'égalité est vraie avant une itération, l'instruction `p = 2 * p` donne $p=2^{k+1}$.
- Conclusion : En sortie de boucle, il y a eu n donc 2^n, ce qui est le résultat attendu.

2.3 Exemples avec une boucle `while`

2.3.1 Factorielle

On définit $n!=1\times2\times\cdots\times n$ pour $n\geq1$, et $0!=1$.

```
1 def factorielle(n):     # precondition : n >= 0
2     p, q = 1, n
3     while q > 0:
4         p = p * q
5         q = q - 1
6     return p
```

Question 5 : Montrer que l'algorithme se termine à l'aide d'un variant de boucle.
Réponse 5 : La variable q est un variant : elle est un entier strictement positif tant que la boucle s'exécute, puis elle diminue de 1 sans devenir négative. La boucle termine donc.

Question 6 : Montrer la correction de l'algorithme à l'aide d'un invariant de boucle.
Réponse 6 : L'invariant est $p\,q!=n!$.

- Initialisation : avant la boucle, $p=1$ et $q=n$, donc $p\,q!=n!$.
- Conservation : si $q>0$ et si la propriété est vraie, les nouvelles valeurs $p'=pq$ et $q'=q-1$ vérifient $p'q'!=pq(q-1)!=pq!=n!$.
- Conclusion : la boucle termine avec $q=0$. L'invariant donne $p\times0!=n!$, donc la valeur renvoyée est $p=n!$.

2. L'astuce est d'additionner terme à terme les lignes suivantes : $S=1+2+3+\cdots+(m-1)+m$ et $S=m+(m-1)+(m-2)+\cdots+2+1$. On obtient m paires valant chacune $m+1$. D'où $2S=m(m+1)$, d'où le résultat.

2.3.2 Division euclidienne

Dans `division_euclidienne(a, b)`, la terminaison a déjà été prouvée à l'aide du variant r. Pour la correction, on choisit l'invariant $a = bq + r$.

```
def division_euclidienne(a, b):  # preconditions : a >= 0, b > 0
    q, r = 0, a
    while r >= b:
        q = q + 1
        r = r - b
    return q, r
```

- Initialisation : avant la boucle, $q = 0$ et $r = a$, donc $bq + r = a$.
- Conservation : une itération remplace q par $q+1$ et r par $r-b$. Ainsi $b(q+1)+(r-b) = bq+r = a$.
- Conclusion : à la sortie, la condition $r \geq b$ est fausse et $r \geq 0$. Avec l'invariant, on a $a = bq + r$ et $0 \leq r < b$: le couple renvoyé est le quotient et le reste de la division euclidienne.

2.3.3 Multiplication

```
def multiplication_1(a, b):
    m = 0
    while b > 0:
        m = m + a
        b = b - 1
    return m
```

Question 7 : Pour `multiplication_1(a, b)`, la valeur de `b` change dans la boucle. En notant b_0 sa valeur initiale, proposer un invariant qui explique la valeur finale de `m`.

Réponse 7 : L'invariant est $m + ab = ab_0$.

- Initialisation : $m = 0$ et $b = b_0$, donc $m + ab = ab_0$.
- Conservation : une itération donne $m' = m + a$ et $b' = b - 1$. Ainsi $m' + ab' = (m + a) + a(b - 1) = m + ab = ab_0$.
- Conclusion : la boucle termine avec $b = 0$; l'invariant donne alors $m = ab_0$, qui est le produit annoncé.

> **Remarque :** Ce n'est pas toujours facile de trouver l'invariant de boucle. Une astuce est de chercher ce qui est transféré d'une variable à une autre à chaque itération. Dans l'exemple précédent, à chaque tour m s'ajoute a et b perd 1. Ainsi la quantité ab perd a à chaque itération, ce qui signifie que $m + ab = \textit{constante}$. Il reste à trouver la constante avec les conditions initiales.

3 Jeu de tests

Une preuve raisonne sur toutes les entrées autorisées. Un **jeu de tests** vérifie le comportement du programme sur un nombre fini de cas choisis. Il est très utile pour découvrir une erreur, mais même un grand nombre de tests réussis ne prouve pas l'absence de défaut.

On choisit notamment un cas ordinaire, un cas limite, un cas où la boucle ne s'exécute pas et, si nécessaire, un cas proche d'une précondition interdite. Les sorties attendues doivent être déterminées indépendamment du programme testé.

Appel de fonction	Résultat attendu	Ce que le test vérifie
`factorielle(0)`	1	boucle non exécutée
`factorielle(1)`	1	une seule itération
`factorielle(5)`	120	cas ordinaire
`division_euclidienne(10,3)`	(3,1)	reste non nul
`division_euclidienne(9,3)`	(3,0)	division exacte

En Python, `assert condition` arrête le programme avec une erreur si la condition est fausse. On peut l'utiliser dans un fichier de tests :

```
assert factorielle(0) == 1
assert factorielle(5) == 120
assert division_euclidienne(10, 3) == (3, 1)
```

Remarque : Les préconditions doivent être traitées séparément : tester `division_euclidienne(10, 0)` ne vérifie pas la spécification annoncée, qui suppose $b > 0$. On peut choisir de rejeter explicitement une telle entrée dans une version plus robuste de la fonction.

4 Complexité

L'exécution d'un programme mobilise un certain nombre de ressources, dont les deux principales sont **le temps de calcul** du processeur et **l'occupation de la mémoire**. La conception d'un programme doit donc **limiter ces besoins** autant que possible.

Il est souvent très difficile de connaître précisément l'efficacité d'un algorithme, car cela dépend de nombreux paramètres physiques liés à l'ordinateur (processeur, mémoire, etc.), mais aussi du langage de programmation utilisé ainsi que du compilateur ou de l'interpréteur. La complexité est une évaluation de l'utilisation de ces ressources et permet de comparer les ressources nécessaires à différents algorithmes.

4.1 Complexité temporelle et spatiale

Définition

Deux ressources

La **complexité temporelle** mesure l'évolution du nombre d'opérations effectuées lorsque la taille des données augmente.
La **complexité spatiale** mesure l'évolution de la mémoire supplémentaire nécessaire.

Le temps affiché par une horloge dépend de la machine, de Python et d'autres programmes en cours d'exécution. Pour comparer les algorithmes, on compte plutôt des opérations élémentaires dans un modèle simplifié. **On s'intéresse surtout à la complexité temporelle.**

Exemple

Deux façons d'échanger des valeurs

Les deux algorithmes suivants échangent les valeurs de deux variables numériques `x` et `y`.

Algorithme 1 : une variable temporaire conserve l'ancienne valeur de `x`.

```
temp = x
x = y
y = temp
```

Algorithme 2 : les anciennes valeurs sont retrouvées par des calculs.

```
x = x + y
y = x - y
x = x - y
```

Instructions	Algorithme 1	Algorithme 2
Variables nommées	`x`, `y`, `temp`	`x`, `y`
Affectations	3	3
Opérations arithmétiques	0	3 : une addition, deux soustractions

Le premier algorithme utilise une variable supplémentaire, mais effectue moins d'opérations. Le second illustre donc un compromis possible entre mémoire et calcul. Ces comptages servent à comprendre les deux ressources ; ils ne prédisent pas à eux seuls le temps réel d'exécution ni la mémoire réellement allouée par Python.

4.2 Instructions élémentaires et coût

Définition

Expression et instruction

Une **expression** est un calcul qui produit une valeur, par exemple `s + x` ou `len(L)`. Une **instruction** demande à Python d'effectuer une action, par exemple l'affectation `s = 8`, une condition `if` ou une boucle `for`. Une instruction peut contenir une ou plusieurs expressions : dans `s = s + x`, l'expression `s + x` est évaluée avant que sa valeur soit affectée à `s`.

Pour estimer un coût, on compte les opérations nécessaires à l'évaluation des expressions et à l'exécution des instructions. Pour la suite on considère comme ayant un coût constant

- une affectation,
- une opération arithmétique sur des nombres de taille bornée,
- une comparaison ou un accès direct à un élément d'une liste,
- une évaluation d'un test, comparaison...

Une expression qui combine un nombre fixé de telles opérations a donc un coût constant.

Remarque : Une fonction Python n'est pas nécessairement élémentaire : `sum(L)` parcourt la liste `L`, et son coût dépend de `len(L)`. De même, les entiers de Python peuvent avoir une taille arbitraire ; le coût réellement payé par certaines opérations croît avec leur nombre de chiffres. Le modèle de coût constant est une approximation adaptée aux premiers exemples.

Exemple

Coût d'une moyenne

Considérons une liste L non vide, de longueur n :

```
def moyenne(L):
    n = len(L)
    s = 0
    for x in L:
        s = s + x
    return s / n
```

Comptons les opérations significatives selon une convention simple, sans détailler le contrôle de la boucle :

Instruction	Opérations comptées	Coût
`n = len(L)`	appel, lecture de la longueur, affectation	3
`s = 0`	affectation	1
`s = s + x`	addition et affectation, répétées n fois	$2n$
`return s / n`	division	1

On obtient ainsi $C(n) = 3 + 1 + 2n + 1 = 2n + 5$. Un comptage plus détaillé des tests et des passages dans la boucle changerait les coefficients, mais pas la croissance linéaire du coût.

4.3 Notation O et ordres de grandeur

Calculer exactement le coût d'un algorithme devient vite fastidieux. Ce qui est surtout important, c'est son **comportement asymptotique** : comment le coût évolue quand la taille n des données devient grande. Pour des entrées de même taille, le coût peut varier ; on distingue alors souvent trois cas :

- le cas le plus favorable (complexité dans le meilleur des cas, souvent peu utile)
- le pire des cas (complexité dans le pire cas, très souvent celle que l'on étudie)
- le cas moyen (complexité en moyenne, parfois délicate à gérer et qui relève plus du travail du mathématicien)

Définition

Notation O

Dans l'étude du pire cas, le problème revient à chercher une fonction $f(n)$ majorant le coût $C(n)$ de l'algorithme.
C'est à dire qu'on écrit $C(n) = O(f(n))$ s'il existe une constante $K > 0$ et un rang n_0 tels que, pour tout $n \geq n_0$, on ait $C(n) \leq K f(n)$. Cette notation exprime une *majoration asymptotique* : on néglige les facteurs constants et les termes qui croissent moins vite.

Exemple

Pour la fonction `moyenne`, si $n \geq 5$, alors $C(n) = 2n + 5 \leq 3n$. On peut choisir $K = 3$ et $n_0 = 5$ dans la définition : $C(n) = O(n)$. On dit que sa complexité est **linéaire**. Il serait aussi vrai, mais moins informatif, d'écrire $C(n) = O(n^2)$ [a].

a. Ne le faites pas...

La Tableau suivant présente les complexités usuelles :

Ordre	Nom
$O(1)$	constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	quasi linéaire
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel

4.4 Meilleur et pire des cas

Le coût peut dépendre des données elles-mêmes, même à taille n fixée. Dans la recherche séquentielle suivante, l'élément peut être rencontré immédiatement ou après le parcours entier :

```
def contient(L, x):
    for element in L:
        if element == x:
            return True
    return False
```

Dans le meilleur des cas, la complexité est $O(1)$; dans le pire des cas, elle est $O(n)$, où $n = \texttt{len(L)}$. Sauf précision contraire, on étudie généralement le pire des cas. Pour une liste *triée*, une recherche dichotomique ramène le pire cas à $O(\log n)$ en divisant par deux l'intervalle de recherche à chaque itération.

4.5 Exercices d'application

Dans chaque exercice, on suppose $n \geq 1$. Compter d'abord combien de fois l'instruction `c = c + 1` est exécutée, puis en déduire la complexité temporelle en fonction de n.

Question 8 : Les deux boucles suivantes sont successives. Déterminer la complexité.

```
def successives(n):
    c = 0
    for i in range(n):
        c = c + 1
    for j in range(n):
        c = c + 1
    return c
```

Réponse 8 : Chaque boucle effectue n incrémentations, soit $n + n = 2n$ au total. La complexité est $O(n)$: deux boucles successives ne multiplient pas leurs nombres de passages.

Question 9 : Les deux boucles suivantes sont imbriquées. Déterminer la complexité.

```
def imbriquees(n):
    c = 0
    for i in range(n):
        for j in range(n):
            c = c + 1
    return c
```

Réponse 9 : Pour chacune des n valeurs de i, la boucle intérieure effectue n passages. Il y a donc $n \times n = n^2$ incrémentations et une complexité en $O(n^2)$.

Question 10 : Dans la fonction suivante, la borne de la boucle intérieure dépend de i. Déterminer la complexité.

```python
def triangulaire(n):
    c = 0
    for i in range(n):
        for j in range(i):
            c = c + 1
    return c
```

Réponse 10 : La boucle intérieure effectue successivement $0, 1, \ldots, n-1$ passages. Le total est $0+1+\cdots+(n-1) = n(n-1)/2$. La complexité est donc $O(n^2)$.

Question 11 : Déterminer ce que fait l'algorithme suivant.
Réponse 11 : Il compte combien de fois il est possible de diviser par 2 le nombre n.

Question 12 : Déterminer la complexité

```python
def moities(n):
    c, i = 0, n
    while i > 1:
        c = c + 1
        i = i // 2
    return c
```

Réponse 12 : Après un passage $i = \left\lfloor \frac{n}{2} \right\rfloor$, après k passages, $i = \left\lfloor \frac{n}{2^k} \right\rfloor$. On s'aperçoit que la boucle effectue $\lfloor \log_2 n \rfloor$ passages pour $n \geq 1$. La complexité est donc logarithmique en $O(\log n)$.