

02 - Complexité - Vérification des algorithmes

1 Complexité en temps et complexité en mémoire

Pour traiter un même problème, il existe souvent plusieurs algorithmes. Si le nombre d'opérations à effectuer est peu important et que les données d'entrée de l'algorithme sont de faibles tailles, le choix de la solution importe peu.

En revanche, lorsque le nombre d'opérations et la taille des données d'entrée deviennent importants, deux paramètres deviennent déterminants : le temps d'exécution (qu'on appelle aussi **coût de l'algorithme**) et l'occupation mémoire.

Définition 1

Complexité en temps : La complexité en temps donne le nombre d'opérations effectuées lors de l'exécution d'un programme.

Complexité en mémoire (ou en espace) : La complexité en mémoire est une estimation de la mémoire utilisée lors de l'exécution d'un programme.

2 Complexité en mémoire

Les calculs de complexité en mémoire ne sont pas un objectif principal du programme en CPGE, mais il est demandé aux étudiants d'avoir des notions de l'espace mémoire nécessaire pour stocker certains types de données.

Exercice 1 (*Extrait CCINP PSI 2026*) :

Le sujet abordait différents algorithmes utilisés dans diverses branches de l'astrophysique..

Le SDSS (Sloan Digital Sky Survey) est un programme de surveillance systématique du ciel. Un télescope balaie chaque nuit une portion du ciel à la recherche de galaxies lointaines et des systèmes automatisés traitent les photographies obtenues pour repérer et classifier les objets astronomiques qui s'y trouvent en notant un grand nombre d'informations utiles aux astronomes. Nous nous contenterons ici d'en utiliser un sous-ensemble pertinent pour donner l'idée générale des manipulations.

I.1 - Observation du ciel et poids des données

Entre autres choses, le SDSS collecte deux types de données primaires :

- les photographies du ciel dans une couleur donnée ;
- les spectres pris pour certains objets lumineux du ciel détectés sur les photographies précédentes.

Une photographie est une matrice de 2048 pixels de largeur pour 1489 de hauteur, chaque pixel pouvant contenir 256 valeurs possibles (entiers de 0 à 255).

Q1. Donner le nombre minimal de bits nécessaires pour encoder la valeur d'un pixel.
En déduire la taille approximative d'une photographie (arrondie au mégaoctet près).

Un spectre consiste à noter l'intensité lumineuse provenant de l'objet observé en fonction de la longueur d'onde. Les spectres considérés notent cette valeur sur un flottant en 64 bits pour des longueurs d'onde allant de 380 à 920 nm par pas de 0,1 nm.

Q2. Déterminer l'espace disque occupé par un spectre (arrondi au kilo-octet près) si on se contente de stocker les valeurs d'intensité lumineuse pour chaque longueur d'onde sous forme d'une succession de flottants.

3 Complexité en temps

3.1 Principe de calcul de la complexité

Remarquons qu'en général, le temps d'exécution d'un algorithme va varier en fonction d'un paramètre que l'on appelle la taille du problème. (Par exemple une fonction qui calcule la factorielle d'un entier n aura une complexité qui dépend de n)

Pour une donnée d'entrée de taille n , la complexité de l'algorithme est notée $C(n)$. C'est le **nombre d'opérations élémentaires** intervenant dans l'algorithme.

Par convention, une affectation, une multiplication, une addition, une comparaison, correspondent chacune à une opération élémentaire.

Evaluer la complexité d'un algorithme est difficile, car :

- les différentes opérations élémentaires ne demandent pas exactement le même temps de calcul
- le temps d'exécution va dépendre de la puissance de la machine utilisée et du langage pour coder.

Ce qui nous intéresse n'est pas le temps précis d'exécution d'un programme, mais l'ordre de grandeur de ce temps en fonction de la taille des données. On cherchera donc à donner des **ordres de grandeur** de ces complexités.

Pour cela on utilise la notation de Landau "**O**".

Exercice 2 : Considérons la fonction `factorielle` implémentée de la façon suivante :

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

Question : déterminer la complexité de cet algorithme en fonction de n .

Remarque : Mathématiquement il ne serait pas faux de dire que l'algorithme précédent a une complexité $C(n) = O(n^2)$ (en effet $\frac{C(n)}{n^2}$ est bien borné puisqu'il tend vers 0), mais cela manquerait de précision pour l'utilisation que l'on veut en faire en informatique.

Ainsi lorsque l'on parlera de complexité en $O(n)$ en informatique, on sous-entend " $O(n)$ et pas mieux". On rencontre parfois la notation $\Theta(n)$ qui contient cette précision.

De manière générale, on dira qu'un algorithme a une complexité en $O(f(n))$ si son coût est, à partir d'un certain rang, inférieur au produit de $f(n)$ par une constante.

⚠ Certaines instructions d'apparence simple sont loin de l'être. Par exemple :

- la calcul de la longueur d'une liste est une opération élémentaire (l'interpréteur la connaît a priori ou la déduit des adresses du premier et du dernier élément).
- les instructions comme `sum(liste)` , ou `if x in liste` ne sont pas élémentaires car elles impliquent un parcours de liste et donc leur temps d'exécution dépend de `len(liste)`.

3.2 Les différentes classes de complexité

Voici les principales classes de complexité que nous rencontrerons, pour indication les temps d'exécution sont donnés pour un ordinateur réalisant 1 milliard d'opérations à la seconde :

Ordre de grandeur	Nom de la complexité	Temps pour $n=10^6$	Commentaires
$O(1)$	Temps constant	1 ns	Le temps ne dépend pas des données traitées.
$O(\ln(n))$	Logarithmique	10 ns	Presque instantanée. Bien souvent, à cause du codage binaire de l'information, c'est $\log_2(n)$ qui apparaît, mais c'est la même chose car la complexité est définie à un facteur près.
$O(n)$	Linéaire	1 ms	Très rapide, le problème de la gestion de mémoire se pose souvent avant la gestion du temps.
$O(n \ln(n))$	Quasi-linéaire	10 ms	Presque aussi bien que linéaire.
$O(n^2)$	Quadratique	15 mn	Pour n plus grand que un million, cette complexité n'est plus acceptable.
$O(n^k)$	Polynomiale	30 ans si $k=3$	Acceptable pour des n petits
$O(2^n)$	Exponentielle	$>10^{300\,000}$ milliard d'années	Acceptable pour des n très petits ($n < 50$), sinon c'est interdit!!

3.3 Le cas des fonctions récursives

Si n est la taille du problème, et si $C(n)$ est la complexité correspondante, le mieux est d'établir une relation de récurrence sur la suite $(C(n))$ et d'en déduire l'expression générale de $C(n)$ en fonction de n , ou au moins une relation de la forme $C(n) = O(f(n))$ au voisinage de $+\infty$.

Exercice 3 : Ecrire une fonction récursive permettant de calculer $n!$ si n est un entier naturel, et déterminer la complexité de cette fonction.

3.4 Complexité dans le meilleur et dans le pire des cas

Exercice 4 : Compléter le script suivant permettant de tester si un entier naturel est premier (le résultat retourné est un booléen)

```
1 def estpremier(n):
2     if n==1:
3         return False
4     for i in range (.....) :
5         if n%i==.....:
6             return ....
7     return ....
```

Soit p un nombre premier supérieur ou égal à 3. Quel est la complexité de cet algorithme si n est égal à p ? Quel est la complexité de cet algorithme si $n = p - 1$?

On constate sur cet exemple que pour des données de tailles similaires (ici p et $p - 1$), les coûts peuvent être très différents. Ici, le cas où n est premier est «*le pire des cas*» dans le sens où il s'agit du cas où il y aura le plus d'itérations possibles. Par opposition le cas où n est pair sera qualifié de «*meilleur des cas*».

A retenir :

Quand on parle de la complexité d'un algorithme sans préciser laquelle, on fait en général référence à la complexité temporelle **dans le pire des cas**.

4 Compléments sur les fonctions

4.1 Spécification d'une fonction

Une spécification est **destinée aux utilisateurs** et permet de les informer de la tâche effectuée par la fonction, de préciser les contraintes imposées par les paramètres et ce qui peut être attendu des résultats.

Elle est résumée dans la *docstring*, écrite au début du corps de la fonction **entre des triples guillemets**.

Elle est facultative.

Exemple 1 :

```
1 def factorielle (n):
2     """ Calcule la factorielle d'un entier naturel .
3     L'argument n doit être un entier naturel .
4     Le résultat renvoyé est le produit de tous les entiers de 1 à n (renvoie 1 pour n = 0). """
5     res=1
6     for i in range(1,n+1):
7         res=res*i
8     return res
```

4.2 Signature d'une fonction

La signature d'une fonction est **destinée aux développeurs** et permet de savoir comment l'utiliser.

Elle regroupe le **nom de la fonction**, la liste et le type de ses **paramètres**, ainsi que le **type du résultat** retourné.

Elle indique la façon exacte dont la fonction doit être appelée sans s'occuper de son implémentation interne.

Elle est facultative.

Exemple 2 :

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

5 Vérification d'un algorithme

5.1 Terminaison d'un algorithme

Si notre programme ne comporte pas de boucles, on est assuré qu'il s'exécutera en un temps fini. Il en va de même si le programme ne comporte que des boucles **for**.

Le problème est lorsqu'il y a une boucle **while**, il peut arriver sur un programme mal écrit qu'une boucle **while** ne se termine jamais. Pour prouver qu'une boucle **while** se termine on utilise un **variant de boucle** :

Définition 2

Un variant de boucle est une quantité (c'est souvent le simple contenu d'une variable) qui vérifie :

- (i) c'est un entier positif tout au long de la boucle.
- (ii) il décroît strictement à chaque itération.

On est ainsi assuré de sortir de la boucle, car comme il n'existe pas de suite infinie strictement décroissante d'entiers naturels, il ne peut y avoir qu'un nombre fini d'itérations.

Exercice 5 : Compléter le script suivant permettant d'effectuer la division euclidienne de a par b , en n'utilisant que des additions ou soustractions :

```
1 def div_euclidienne (a: int, b: int) -> tuple[int, int]:
2     ''' a et b sont des entiers positifs, b est non nul.
3     Retourne le quotient et le reste de la division de a par b '''
4     q=0
5     r=a
6     while r>=b:
7         q+=1
8         r=r-b
9     return q,r
```

Prouver la terminaison de la boucle en montrant que r est un variant de boucle.

5.2 Correction d'un algorithme

Il s'agit de prouver que notre algorithme renvoie bien le résultat attendu.

On utilise pour cela **un invariant de boucle**.

Un invariant de boucle est une propriété (une expression booléenne), qui est vraie avant l'entrée dans la boucle et qui reste vraie après chaque passage dans la boucle. Plus précisément :

Définition 3

Une propriété P est un invariant si :

- (i) P est vraie avant d'entrer dans la boucle
- (ii) Si elle est vérifiée avant une itération, elle est vérifiée après celle-ci
- (iii) Enfin cette propriété, composée avec le fait que la condition de continuation du while devienne fausse ou que la boucle for se termine, doit permettre de vérifier que le programme retourne bien le résultat attendu.

Exercice 6 : On reprend la fonction permettant d'effectuer la division euclidienne de a par b .

Montrer que la propriété $P : "a = bq + r \text{ et } r \geq 0"$ est un invariant de boucle, et prouver la correction de cet algorithme.

6 Exercices

Exercice 7 : 1. Que fait la fonction ci-dessous ? On attend une réponse donnée sous forme d'une phrase claire.

```
1 def mystere(L1: list[int]) -> list[tuple[int, int]]:
2     n = len(L1)
3     L2 = []
4     for i in range(n):
5         for j in range(n):
6             L2.append((L1[i], L1[j]))
7
8     return L2
```

2. Déterminer la complexité de cette fonction.

Exercice 8 : Voici le script de la fonction de tri à bulles, qui permet de trier en place une liste d'entiers, par ordre croissant.

On rappelle que le principe de ce tri consiste, par parcours successifs, à faire remonter les plus grands éléments de la

liste à la fin de cette liste. (au premier parcours, par échange successifs, le plus grand élément se retrouvera à la fin de la liste)

```

1 def tri_bulles (L):
2     n=len(L)
3     for j in range(n):
4         for i in range(n-j-1):
5             if L[i]>L[i+1]:
6                 L[i], L[i+1]=L[i+1],L[i]      #échange de deux éléments
7     return L

```

Donner, en justifiant, la complexité en temps dans le pire des cas, en fonction de la taille n de la liste L .

Exercice 9 (Extrait CCINP PSI 2026) :

Le sujet abordait différents algorithmes utilisés dans diverses branches de l'astrophysique..

On définit la distance entre deux spectres (vus comme des vecteurs à N composantes)

$$d(S_1, S_2) = \sqrt{\frac{1}{N} \sum_{i=0}^{N-1} (S_{1,i} - S_{2,i})^2}.$$

Informatiquement, un spectre est manipulé sous forme d'une liste de N flottants.

Q7. Écrire une fonction `distance(S1, S2)` qui, étant donné deux listes de flottants de même taille $S1$ et $S2$ représentatives de deux spectres, renvoie la distance entre les deux spectres comme définie ci-dessus.

Néanmoins, les observations sont souvent entachées d'erreurs du fait des rayons cosmiques ou de capteurs localement défectueux, ce qui pousse à ignorer certaines positions de chaque spectre. Les listes représentant les spectres contiennent donc par endroit la valeur `None` pour simuler l'absence de valeur acceptable.

Q8. Écrire une fonction `positions_a_rejeter(S1, S2)` qui prend en entrée deux listes et renvoie la liste des indices i pour lesquels soit $S1[i]$, soit $S2[i]$ vaut `None`.

Q9. Implémenter une fonction `est_absent(element, L)` qui renvoie `True` si `element` est absent de la liste L et `False` sinon. Donner la complexité temporelle dans le pire des cas de votre implémentation en fonction de $r = \text{len}(L)$.

On propose deux implémentations (qui ne sont pas forcément optimales en termes de complexité) du calcul de distance entre les spectres $S1$ et $S2$ en utilisant la liste `a_rejeter` des positions à rejeter sur l'ensemble des deux spectres.

<pre> def distance2(S1, S2, a_rejeter): d = 0 N = len(S1) m = N - len(a_rejeter) for i in range(N): if est_absent(i, a_rejeter): d = d + (S1[i]-S2[i])**2 return (d / m) ** 0.5 </pre>	<pre> def distance3(S1, S2, a_rejeter): d = 0 N = len(S1) m = N - len(a_rejeter) S1c = S1.copy() S2c = S2.copy() for i in a_rejeter: S1c[i] = 0 S2c[i] = 0 for i in range(N): d = d + (S1c[i]-S2c[i])**2 return (d / m) ** 0.5 </pre>
--	---

Q10. Énoncer une propriété invariante de boucle qui serait utile pour démontrer la correction de la fonction `distance2` (on ne demande pas de démonstration).

Q11. Estimer les complexités temporelles de chacune des deux implémentations `distance2` et `distance3` en fonction de $N = \text{len}(S1)$ et de $r = \text{len}(a_rejeter)$.

Exercice 10 : On souhaite écrire une fonction qui calcule le terme général de la suite définie par $u_n = \sum_{k=0}^n \frac{1}{k!}$

- (a) Ecrire une fonction `somme` répondant au problème, qui utilise la fonction `factorielle` créée dans l'exercice 2 (page 2).
(b) Déterminer la complexité de cet algorithme en fonction de n .
- (a) Ecrire un programme `somme_améliorée` qui effectue le calcul de u_n en utilisant moins d'opérations.
(b) Déterminer la complexité de cet algorithme en fonction de n .

Exercice 11 (Extrait CCINP PSI 2021) :

Le sujet traitait des montres connectées et de l'acquisition des données GPS

Lors d'une activité, on récupère toutes les secondes :

- les données du GPS : horaire (en seconde), latitude (en degré), longitude (en degré) et altitude (en mètre);
- la fréquence cardiaque.

Toutes les secondes, ces 5 données sont enregistrées dans un fichier texte, en les séparant par une virgule. Chaque ligne du fichier correspond à un "point" de mesure.

Exemple d'une ligne du fichier texte : "34108,47.911498,001.911526,0084.3,105\n".

L'horaire, en secondes, est un entier représenté avec 5 caractères. On garde 6 chiffres après la virgule pour les angles en degrés. La partie entière de l'altitude est représentée avec 4 caractères et on prend une précision au décimètre. La fréquence cardiaque est un entier représenté avec 3 caractères. Tous les caractères sont stockés sur 1 octet (ascii).

Q10. On suppose que l'on réalise une activité d'une heure. Donner l'ordre de grandeur de la taille mémoire en octets du fichier texte généré. Dans le cahier des charges, on impose que la montre peut sauvegarder 200 h d'activités. Donner l'ordre de grandeur de la taille mémoire nécessaire au stockage des données pour répondre au cahier des charges.

Exercice 12 : On considère la fonction Python suivante, censée calculer la puissance x^n d'un nombre réel x pour un entier naturel n :

```
1 def puissance(x, n):
2     res = 1.0
3     k = n
4     while k > 0:
5         res = res * x
6         k = k - 1
7     return res
```

- (a) Proposer un variant de boucle pour la fonction `puissance`.
(b) Démontrer la terminaison de cet algorithme à l'aide de ce variant.
- (a) Énoncer un invariant de boucle reliant la valeur de `res`, `x`, `k` et `n`.
(b) Prouver la correction de la fonction (initialisation, conservation et terminaison).