02 - Complexité

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.

Evaluer la complexité d'un algorithme est difficile, car cette complexité va notamment dépendre :

1. de la puissance de la machine sur laquelle l'algorithme est exécuté.
2. du langage et compilateur / interpréteur utilisé pour coder l'algorithme.

On cherchera donc à donner des ordres de grandeur de ces complexités.

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.

Exemple 1 : Avec la norme ASCII, un caractère est stocké sur 7 bits.

Question : combien de caractères différents peut-on ainsi stocker ?

Exercice 1 (*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.

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 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.

C'est **l'ordre de grandeur** de la complexité qui nous intéresse, on cherchera donc à écrire $C(n) = O(f(n))$ (« $C(n)$ est dominée par $f(n)$ »)

Exemple 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.

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 $+\infty$.

Exemple 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

Exemple 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 Exercices

Exercice 2 : On souhaite écrire une fonction qui calcule le terme général de la suite définie par $u_n = \sum_{k=0}^{n} \frac{1}{k!}$

1. (a) Ecrire une fonction **somme** répondant au problème, qui utilise la fonction **factorielle** créée dans l'exemple 2 (page 2).
   (b) Déterminer la complexité de cet algorithme en fonction de n .
2. (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 3 (*Recherche d'un élément dans une liste*) : .

1. Ecrire une fonction (non récursive) **est_present**, ayant pour paramètres d'entrée une liste **L** de nombres et un nombre **x**, et qui retourne un booléen indiquant si **x** est un élément de **L** ou pas.
2. Déterminer la complexité de cet algorithme en fonction de la taille de la liste.

Exercice 4 (*Recherche dichotomique d'un nombre dans une liste de nombres triée*) : .

Comme dans l'exercice précédent, il s'agit de créer une fonction qui permet de savoir si un élément **x** appartient à une liste **L**. Mais dans cet exercice on suppose que la liste de nombres **L** est **triée par ordre croissant**. Ceci va nous permettre d'avoir une méthode de recherche beaucoup plus efficace qu'un simple recherche linéaire comme dans l'exercice précédent.

Le principe utilisé s'appelle **la dichotomie**, et voici son fonctionnement :

(i) On définit la variable **mini**, initialisée au premier indice de la liste, et la variable **maxi**, initialisée au dernier indice de la liste.

(ii) On définit la variable **milieu**, égale à **(mini + maxi)//2**.

(iii)
- Si **x** est égal à **L[milieu]**, c'est fini.
- Si **x** est strictement inférieur à **L[milieu]**, il faut chercher **x** dans la première partie de la liste, donc on pose **maxi=milieu-1** , et on retourne au point (ii)
- Si **x** est strictement supérieur à **L[milieu]**, il faut chercher **x** dans la deuxième partie de la liste, donc on pose **mini=milieu+1** , et on retourne au point (ii).
- On continue tant que **mini≤maxi**.

1. En suivant ce principe, écrire une fonction **recherche_dicho**, ayant pour paramètres d'entrée une liste **L** de nombres rangés par ordre croissant et un nombre **x**, et qui retourne un booléen indiquant si **x** est un élément de **L** ou pas.
2. Déterminer la complexité de cette fonction. (on pourra tout d'abord déterminer la complexité dans le cas où la liste soit de taille 2^p, en déterminant le nombre maximal de tours de boucle effectués)
3. Ecrire une fonction **recherche_dicho_rec**, qui utilise aussi la recherche par dichotomie dans une liste triée, mais cette fois-ci en utilisant une programmation récursive.
4. Déterminer la complexité de cette fonction récursive.