

COMPLEXITÉ.

1. Problématique

Un exemple rencontré en TP : recherche d'un élément dans une liste triée

On dispose de deux algorithmes classiques :

- La recherche séquentielle
- la recherche par dichotomie

Pour comparer les deux algorithmes, évaluons le temps d'exécution de chacun d'eux en Python :

Recherche séquentielle.

- le programme

```
def recherche(x, L):  
    """recherche x dans la liste croissante L"""  
    k = 0  
    n = len(L)  
  
    while k < n and L[k] < x :  
        # Inv. : L[0] <= L[1] <= ... <= L[k] < x  
        k = k + 1  
  
    # en sortie de boucle on a :  
    # k = n ou bien L[k] >= x > L[k-1]  
    return (k < n and L[k] == x)
```

- le temps d'exécution pour la recherche de 10^7 dans la liste des nombres pairs de 0 à $2 \cdot 10^7$.

```
from time import clock  
def chrono_recherche():  
    L = [2*k for k in range(10**7 + 1)]  
    x = 10**7  
    t0 = clock()  
    reponse = recherche(x, L)  
    t1 = clock()  
    return reponse, t1 - t0
```

- Réponse :

Recherche par dichotomie.

- Le programme :

```
def dichotomie (x, L):
    """ Renvoie True ssi x appartient à L
        par une recherche dichotomique
        lorsque L est triée """
    g = 0
    d = len(L)
    # Si x est dans L alors il est dans L[g:d].
    # Ça sera notre invariant de boucle.
    # remarque : g inclus et d exclu, comme d'habitude
    while g < d-1:
        m = (g+d) // 2
        if x < L[m]:
            d = m
        else:
            g = m
    return x == L[g]
```

- le temps d'exécution pour la recherche de 10^7 dans la liste des nombres pairs de 0 à $2 \cdot 10^7$ est alors :

Manifestement, la méthode par dichotomie est plus efficace mais cette façon de procéder possède plusieurs inconvénients :

- elle nécessite de programmer les deux méthodes alors qu'on ne souhaite garder que la plus efficace.
- elle dépend de l'ordinateur sur lequel on travaille.
- elle dépend des valeurs que l'on teste.

Complexité temporelle

Pour résoudre un problème, il existe souvent plusieurs algorithmes. Il est important de savoir évaluer leur efficacité, les problèmes posés portant souvent sur des données importantes (par exemple, pouvoir retrouver les notes d'un candidat aux concours sur les 16000 participants, ou encore le traitement d'une image de 18 millions de pixels...).

Pour cela, on introduit une mesure appelée **complexité temporelle**, qui permet de donner un ordre de grandeur du temps d'exécution de l'algorithme. Pour la calculer, on évalue le nombre d'actions élémentaires du programme, ce nombre étant proportionnel au temps d'exécution, en considérant que l'exécution d'une tâche élémentaire prend toujours le même temps (temps dépendant de l'opération).

On appelle **opération élémentaire** toute opération :

- algébrique : +, -, *, /
- de comparaison : =, <, >, <=, >=
- d'affectation
- d'affichage.

Exemple. Soit n un entier. On peut programmer le calcul de la somme des entiers de 0 à n :

```
def somme(n):  
    """ calcule la somme des n premiers entiers """  
    S = 0  
    for i in range(n+1):  
        S = S + i  
    return S
```

Les opérations élémentaires sont : les affectations, les additions, les comparaisons, les affichages. Dénombrons-les.

On dénombre $2n + 4$ affectations, $2n + 2$ additions, $n + 2$ comparaisons et 1 affichage : la complexité temporelle est proportionnelle à n .

Remarque. En pratique, il n'est pas utile d'évaluer toutes les opérations élémentaires. En général, on s'attache à une opération significative. Dans le calcul d'une somme, ce sera le nombre d'additions ; dans la recherche d'un maximum, ce sera le nombre de comparaisons.

Comparaison d'algorithmes

Exemple. Reprenons le calcul de la somme, connaissant une expression de la somme en fonction de n , ce qui donne directement l'algorithme suivant :

```
def somme(n):  
    """ calcule la somme des n premiers entiers """  
    return n * (n+1) / 2
```

On dénombre alors trois opérations algébriques ! La complexité est constante : le temps d'exécution ne dépend pas de la valeur de n .

2. Expression de la complexité

Objectif : comparer les algorithmes résolvant un même problème grâce à leur complexité.

Taille du problème

Pour déterminer la complexité temporelle, on cherche à évaluer le nombre d'opérations élémentaires en fonction de la taille des données du problème.

Exemple.

- Dans les problèmes sur les entiers, comme le calcul de la somme des premiers entiers, l'affichage des diviseurs..., on évaluera la complexité en fonction de l'entier n paramètre de la fonction.
- Dans les problèmes sur les listes, comme la recherche d'un élément ou la recherche d'un maximum, on évaluera la complexité en fonction de l'entier n la longueur de la liste.

Remarque. Dans les problèmes sur les entiers, si n est très grand, il arrive qu'on évalue la complexité en fonction du nombre de chiffres de n au lieu de n lui-même.

Estimation de la complexité

Dans les exemples de la première partie, on a dénombré de façon précise le nombre d'opérations élémentaires. Mais ce n'est pas toujours possible et cela n'est pas nécessaire.

Il est important d'estimer la complexité lorsque la taille du problème devient grande. Dans ce cas, on peut simplifier le nombre d'opérations en ne tenant compte que du terme dominant.

Exemples. Si on a compté $n^2 + 3n$ opérations, on dira que la complexité est en $O(n^2)$.

Si on a compté $5n^2 + 3n$ opérations, on dira aussi que la complexité est en $O(n^2)$. Le facteur constant devant n^2 n'est pas significatif car la complexité se veut simplement proportionnelle au temps d'exécution.

Remarque. Soient u et v deux suites réelles. $u_n = O(v_n) \iff$ la suite $\left(\frac{u_n}{v_n}\right)$ est bornée

Parallèle entre la complexité en O et le temps d'exécution

Ci-dessous un tableau des ordres de grandeur des temps d'exécution d'un problème de taille 10^6 sur un ordinateur à un milliard d'opérations par seconde selon les catégories principales de complexité.

Complexité	Nom courant	Un exemple	Temps pour $n = 10^6$	Remarques
$O(1)$	temps constant	somme des entiers (algo2)	1 ns	Le temps d'exécution ne dépend pas des données traitées, ce qui est assez rare.
$O(\log n)$	logarithmique	recherche d'un élément par dichotomie	10 ns	Cela correspond à une exécution quasi-instantanée. A cause du codage binaire, c'est souvent la fonction $\log_2$ qui intervient mais cela n'a pas d'importance car la complexité est définie à un facteur près.
$O(n)$	linéaire	recherche des diviseurs d'un entier	1 ms	Le temps d'exécution d'un tel algorithme ne devient supérieur à 1mn que pour des données de taille comparable à celle des mémoires vives disponibles actuellement. Le problème de la gestion de la mémoire se posera donc avant celui de l'efficacité en temps.
$O(n^2)$	quadratique	tris simples	1/4 h	Cette complexité reste acceptable pour des données de taille moyenne ($n < 10^6$).
$O(n^k)$	polynomiale		30 ans si $k = 3$	n^k est le terme de plus haut degré d'un polynôme en n . Il n'est pas rare de voir des complexités en $O(n^3)$ ou $O(n^4)$.
$O(2^n)$	exponentielle		plus de 10^{300000} milliards d'années	Un algorithme d'une telle complexité est impraticable sauf pour de très petites données ($n < 50$).

3. Approfondissement

Coûts minimum, maximum et moyen

Le nombre d'opérations effectuées par l'algorithme peut dépendre des valeurs sur lequel il est testé.

Par exemple, dans la recherche d'un élément dans une liste, le nombre d'opérations ne sera pas le même selon si l'élément est dans la liste ou non, ou encore s'il est en début ou fin de liste.

On s'intéresse donc à trois types de coût :

- le coût minimum : il correspond au meilleur des cas. Il permet de prévoir le temps minimum d'exécution.
- le coût maximum : il correspond au pire des cas. Il permet d'estimer si l'algorithme est "réaliste".
- le coût moyen : il correspond à la valeur moyenne des coûts sur l'ensemble des données possibles. Il doit tenir compte de la fréquence d'apparition d'une donnée. C'est un problème difficile.

Remarque. Si la complexité dans le meilleur des cas et le pire des cas est du même ordre alors le temps d'exécution est relativement indépendant des données. Il ne dépend donc que de la taille des données.

Complexité et analyse numérique

Considérons une suite qui converge vers ℓ telle que $\forall n \in \mathbb{N}, |u_n - \ell| \leq \frac{1}{n^3}$.

On veut déterminer une valeur approchée de ℓ à 10^{-9} près.

Théoriquement, il faut calculer u_{1000} mais, en pratique, si la calculatrice fait des erreurs d'arrondis à 10^{-12} près, et qu'il faut n^2 opérations pour calculer u_n , les erreurs de calcul s'accumulent ce qui donne une erreur de l'ordre de 10^{-6} .

Il n'est pas possible d'obtenir une valeur approchée à 10^{-9} avec cette calculatrice. Il faudra utiliser un outil plus performant.

Complexité spatiale

Un autre critère pour évaluer l'efficacité d'un algorithme est la place nécessaire en mémoire pour le faire fonctionner. Cela revient à estimer la taille des variables utilisées au cours de l'algorithme. C'est moins crucial aujourd'hui car les ordinateurs sont très performants.

4. Limites du modèle

La définition de la complexité donnée dans ce chapitre constitue un *modèle* et, comme tout modèle, n'est qu'un reflet imparfait de la réalité.

Malheureusement, dans certains cas, les hypothèses sous-jacentes à ce modèle ne tiennent pas. Il est, par exemple, irréaliste de penser qu'une opération arithmétique ait un coût unitaire : par exemple, l'addition de deux nombres entiers à n chiffres nécessite de lire tous les chiffres et d'écrire ceux du résultat et demande donc un temps de calcul proportionnel à n .

Toutefois, le modèle proposé convient bien à une large classe de problèmes et sera donc pertinent dans beaucoup de situations.