Complexité

En informatique, il existe souvent plusieurs algorithmes permettant de traiter un même problème. On souhaite généralement choisir ceux qui minimisent le temps d'exécution et la quantité de mémoire utilisée, mais ces derniers sont difficiles à évaluer précisément car ils dépendent de la machine sur lequel le programme est exécuté.

L'analyse de la **complexité d'un algorithme** est l'étude formelle de son efficacité, indépendamment des facteurs matériels.

Définition

La **complexité en espace** d'un algorithme est la place occupée en mémoire par son exécution. La **complexité en temps** d'un algorithme est le nombre d'opérations élémentaires qu'il effectue. Ici on s'intéressera essentiellement à la complexité en temps.

Avec Python, on considérera comme opérations élémentaires :

- les opérations `+`, `-`, `*`, `//`, `%` sur les entiers (compris entre -2^{63} et $2^{63}-1$),
- les opérations `+`, `-`, `*`, `/` sur les flottants,
- les affectations,
- les comparaisons (`==`, `<`, etc.) entre entiers, flottants ou caractères,
- l'accès à un élément d'une liste (`L[i]`), d'un tuple ou d'une chaîne de caractères,
- l'ajout d'un élément à la fin d'une liste (`append`).

Bien évidemment, la complexité dépend des données à traiter. Considérons par exemple la fonction suivante qui, recevant un entier naturel n, renvoie la somme $1+2+\ldots+n$:

```
def somme(n):
    s = 0
    for i in range(1, n+1):
        s = s + i
    return s
```

Quand on l'exécute, cette fonction effectue n additions et $n+1$ affectations, soit $2n+1$ opérations élémentaires (on ne tient pas compte de l'incrémentation de `i` dans la boucle).

Exercice 1

1) Combien d'opérations élémentaires sont effectuées par la fonction `moyenne` suivante quand on l'applique à une liste de longueur n ?

```
def moyenne(L):
    s = 0
    for i in range(len(L)):
        s = s + L[i]
    return s / len(L)
```

2) Les deux fonctions suivantes reçoivent une liste de nombres et renvoient sa variance. Tester ces fonctions sur des listes de grande taille. Laquelle est préférable ?

```
def variance(L):
    s = 0
    for i in range(len(L)):
        s = s + (L[i]-moyenne(L))**2
    return s / len(L)
```

```
def variance(L):
    s = 0
    m = moyenne(L)
    for i in range(len(L)):
        s = s + (L[i]-m)**2
    return s / len(L)
```

Complexité dans le meilleur et dans le pire des cas

La complexité d'un algorithme ne dépend pas seulement de la taille des données à traiter. On cherche généralement à déterminer la complexité **dans le meilleur des cas** et la complexité **dans le pire des cas**.

Considérons par exemple la fonction suivante, qui reçoit une liste d'entiers et renvoie la somme de ses termes pairs :

```
def somme_pairs(L):
    s = 0
    for i in range(len(L)):
        if L[i] % 2 == 0:
            s = s + L[i]
    return s
```

Quand on l'applique à une liste de longueur n, cette fonction fait :

– n accès à `L`, n calculs de modulo, n comparaisons et 1 affectation si tous les termes de la liste sont impairs (c'est le meilleur des cas).

– $2n$ accès à `L`, n calculs de modulo, n comparaisons, $n+1$ affectations et n additions si tous les termes de la liste sont pairs (c'est le pire des cas).

Exercice 2 Déterminer le nombre de comparaisons et d'affectations effectuées par la recherche du minimum d'une liste de longueur n dans le meilleur et le pire des cas :

```
def minimum(L):
    m = L[0]
    for i in range(1, len(L)):
        if L[i] < m:
            m = L[i]
    return m
```

Complexité asymptotique

On s'intéresse généralement à ce qui se passe lorsque la taille des données à traiter est grande : on parle de **complexité asymptotique**. Le nombre exact d'opérations est rarement nécessaire, une approximation suffit. On utilisera la notation suivante. Soient (u_n) et (v_n) deux suites réelles positives. On note :

$u_n = O(v_n)$ s'il existe $C \in \mathbb{R}^+$ tel que $u_n \leqslant Cv_n$ à partir d'un certain rang.

Si, par exemple, un algorithme effectue $3n+5$ opérations élémentaires sur des données de taille n, on dira que sa complexité asymptotique est en $O(n)$ (car $3n+5 \leqslant 4n$ pour n assez grand).

Exemple : pour rechercher si un élément appartient ou non à une liste donnée, on peut simplement parcourir cette liste jusqu'à ce qu'on trouve l'élément cherché :

```
def recherche(x, L):
    for i in range(len(L)):
        if L[i] == x:
            return True
    return False
```

Notons n la longueur de `L`. Dans le meilleur des cas, `x` est en première position de `L`, et on ne fait qu'une comparaison et un accès à `L` : l'algorithme est en $O(1)$. Dans le pire des cas, `x` est en dernière position de `L` ou n'appartient pas à `L`, et on fait n comparaisons et n accès à `L` : l'algorithme est en $O(n)$.

Exercice 3 Déterminer la complexité asymptotique de la recherche naïve dans un tableau bidimensionnel de taille $n \times p$:

```
def recherche_2D(x, t):
    n, p = len(t), len(t[0])
    for i in range(n):
        for j in range(p):
            if t[i][j] == x:
                return True
    return False
```

Vocabulaire

Complexité	Nom	Exemples
$O(1)$	constante	opération élémentaire
$O(\ln n)$	logarithmique	recherche dichotomique dans une liste triée
$O(n)$	linéaire	parcours d'une liste
$O(n \ln n)$	quasi-linéaire	tri rapide (en moyenne), tri fusion, tri par tas
$O(n^2)$	quadratique	tri sélection, tri insertion, tri à bulles
$O(n^3)$	cubique	produit matriciel naïf
$O(a^n)$	exponentielle	problème du sac à dos
$O(n!)$	factorielle	calcul naïf du déterminant

Exercice 4 En supposant qu'un ordinateur effectue une opération élémentaire toutes les 10 nanosecondes, combien de temps lui faut-il pour exécuter un algorithme de complexité constante, logarithmique, etc. lorsque les données sont de taille $n = 10^6$?

Remarques

1) Certaines fonctions et techniques de Python, même apparemment simples (copie, concaténation, insertion ou suppression d'un élément dans une liste, slicing, etc.), ont leur propre complexité. Si on les utilise, il faut en tenir compte pour les calculs de complexité. Voir par exemple `wiki.python.org/moin/TimeComplexity` pour les listes.

2) En pratique, on peut utiliser la fonction `time` du module éponyme pour mesurer le temps d'exécution d'un programme. Elle renvoie le temps écoulé depuis **l'epoch** (le 1er janvier 1970 à minuit). On l'importe en faisant `from time import time`. Considérons par exemple les scripts suivants :

```
t = time()
L = []
for i in range(10**5):
    L.append(i)
print(time()-t)
```

```
t = time()
L = []
for i in range(10**5):
    L = L + [i]
print(time()-t)
```

Ils affichent respectivement `0.01078653335571289` et `11.595031261444092`.

Explication : dans le deuxième cas, la liste `L` est recopiée à chaque itération. En considérant que copier une liste de longueur n revient à faire n affectations, on fait ainsi $1+2+\ldots+10^5 \approx 10^{10}/2$ affectations.

Exercice

On veut écrire une fonction `tous_distincts` qui, appliquée à une liste, renvoie `True` si ses éléments sont tous distincts et `False` sinon. Ainsi `tous_distincts([2, 5, 6, 8, 4])` doit renvoyer `True` et `tous_distincts([2, 8, 6, 8, 4])` doit renvoyer `False`.

1) On propose de comparer les éléments de `L` deux à deux de la manière suivante :

```
def tous_distincts(L):
    for i in range(len(L)):
        for j in range(i):
            if L[i] == L[j]:
                return False
    return True
```

Appliquer cette fonction à la main aux listes `[2, 5, 6, 8, 4]` et `[2, 8, 6, 8, 4]`. Déterminer sa complexité asymptotique (en fonction de la longueur n de la liste) dans le pire et le meilleur des cas.

2) En Python un **ensemble** (objet de type `set`) est une structure de données dans laquelle les objets n'apparaissent qu'une seule fois. L'ajout d'un élément et les tests d'appartenance sont en $O(1)$. On propose la fonction suivante :

```
def tous_distincts(L):
    vus = set()           # ensemble vide
    for x in L:
        if x in vus:      # test d'appartenance
            return False
        else:
            vus.add(x)    # ajout d'un élément
    return True
```

Appliquer cette fonction à la main aux listes `[2, 5, 6, 8, 4]` et `[2, 8, 6, 8, 4]`. Déterminer sa complexité asymptotique (en fonction de la longueur n de la liste) dans le pire et le meilleur des cas.