Complexité ou coût d'un algorithme

Temps d'exécution d'un programme

"Facteurs de temps d'exécution d'un programme"

Quand on a écrit un programme pour résoudre un problème, il est naturel de se poser la question de son temps d'exécution.

Un programme traduit un **algorithme** dans un **langage de programmation** et un programme s'exécute sur une **machine**. Le temps d'exécution va dépendre de ces trois facteurs :

- **l'algorithme** choisi pour résoudre le problème peut coûter plus ou moins d'opérations élémentaires (calculs arithmétiques et logiques)
- **le langage de programmation** peut utiliser plus ou moins bien le jeu d'instructions de la machine et l'implémentation d'un même algorithme peut êtres plus ou moins rapide selon le langage choisi
- enfin **la machine** sur laquelle s'exécute le programme peut exécuter plus ou moins d'instructions par seconde

Outils

"Mesure du temps d'exécution d'un programme"

Le module `time` de Python fournit différentes fonctions de manipulation ou mesure du temps.

Fonction	Syntaxe	Action
`time.perf_counter`	`time.perf_counter()`	Donne un mesure du temps en secondes depuis une origine des temps
`time.sleep`	`time.sleep(n)`	Bloque l'exécution du programme pendant `n` secondes

On peut alors écrire une fonction `mesure_temps` pour mesurer le temps d'un progamme donné sous la forme d'une fonction `prog` (sans paramètre) :

```
import time

def mesure_temps(prog):
    """Mesure le temps d'exécution de la fonction prog
    qui ne prend pas d'argument."""
    debut = time.perf_counter()
    prog()
    fin = time.perf_counter()
    return fin - debut
```

"Exercice 1 : boucles et temps d'exécution"

"Question 1"

Donner le temps d'exécution de la fonction suivante :

```
def prog1():
    for j in range(4):
        time.sleep(1)
    for j in range(6):
        time.sleep(1)
```

Vérifier votre réponse en utilisant la fonction `mesure_temps` définie ci-dessus.

Réponse :

.....

"Question 2"

Donner le temps d'exécution de la fonction suivante :

```
def prog2():
    for j in range(4):
        for j in range(6):
            time.sleep(1)
```

Vérifier votre réponse en utilisant la fonction `mesure_temps` définie ci-dessus.

Réponse :

.....

"Question 3"

Donner le temps d'exécution de la fonction suivante :

```
def prog3():
    for j in range(3):
        for j in range(3):
            for k in range(3):
                time.sleep(1)
```

Vérifier votre réponse en utilisant la fonction `mesure_temps` définie ci-dessus.

Réponse :

.....

"Question 4"

Donner le temps d'exécution de la fonction suivante :

```
def prog4():
    n = 2 ** 16
    while n >= 1:
        n = n // 2
        time.sleep(1)
```

Vérifier votre réponse en utilisant la fonction `mesure_temps` définie ci-dessus.

Réponse :

.....

"Question 5"

Donner le temps d'exécution de la fonction suivante :

```
def prog5():
    n = 5
    while n != 0:
        n = n - 2
        time.sleep(1)
```

Réponse :

.....

"Calcul du temps d'exécution avec des boucles"

- Le temps d'exécution d'une séquence d'instructions est la somme des temps d'éxécution de chaque instruction.
- En particulier si toutes les itérations d'une boucle ont le même temps d'exécution, le temps d'exécution total est le produit du nombre d'itérations par le temps d'exécution d'une itération.
- Pour un succession de boucles, on additionne les temps d'exécution de chaque boucle.
- Pour des boucles imbriquées, le temps d'exécution du bloc le plus interne est multiplié par le produit des nombres d'itérations de chacune des boucles imbriquées.

Coût ou complexité temporelle d'un algorithme

"Exercice 2"

On veut énumérer toutes les séquences de n bits (une séquence de n bits est une suite de n chiffres 0 ou 1).

✓ "Question 1"

Compléter le tableau suivant :

n	Nombre de séquences de n bits
1	2
2	4
3	...
4	...

✓ "Question 2"

Donner une formule pour le nombre de séquences de n bits.

Réponse :

.....

✓ "Question 3"

Compléter la fonction Python suivante qui énumère toutes les séquences de n bits :

```
def enum_seq_bits(n):
    lis_seq_bits = [[]]
    for k in range(n):
        lis_seq_bits2 = []
        for seq in lis_seq_bits:
            for bit in [0, 1]:
                lis_seq_bits2.append(      ...       ) # à compléter
        lis_seq_bits = ...           # à compléter
    return lis_seq_bits
```

"Question 4"

Calculer le nombre de séquences de bits que doit énumérer la fonction `enum_seq_bits` pour $n = 200$.

Sachant que l'âge de l'univers est d'environ 14 milliars d'année et qu'en 2026 un supercalculateur peut énumérer 10^{18} séquences de bits par secondes, est-il raisonnable d'exécuter `enum_seq_bits(200)` ?

Réponse :

.....
.....
.....

"Complexité ou coût d'un algorithme"

Avant d'exécuter un algorithme sur une entrée, il faut estimer son **coût d'exécution** en nombre d'opérations élémentaires (calculs arithmétiques et logiques). Ce coût temporel appelé **complexité temporelle** caractérise l'algorithme. On peut aussi mesurer la **complexité spatiale** d'un algorithme c'est-à-dire l'espace qu'il va utiliser en mémoire lors de son exécution.

Même si l'algorithme se termine théoriquement, en pratique il peut ne jamais se terminer même sur une machine très puissante, si sa complexité temporelle est mauvaise.

"Exercice 3"

"Question 1"

On considère la fonction ci-dessous de recherche du maximum dans une liste :

```python
def maximum(lis):
    assert len(lis) > 0, "lis doit être non vide"
    maxi = lis[0]  # affectation
    for i in range(len(lis)):
        if lis[i] > maxi:  # comparaison
            maxi = lis[i]  # affectation
    return maxi
```

On néglige les opérations qui font évoluer la variable de boucle `i`.
On évalue `maximum(lis)` avec une liste de taille `n`.

1. Décompter les nombres respectifs d'affectations et de comparaisons effectuées.
2. On considère que toutes les affectations et comparaisons ont le même coût temporel constant noté c. Exprimer en fonction de n et c le coût total en affectations et comparaisons.

Réponse :

.....
.....
.....

"Question 2"

On considère la fonction ci-dessous qui prend en paramètre une liste `lis` et qui renvoie un booléen indiquant si la liste `lis` contient au moins un doublon (un doublon est un élément qui apparaît au moins deux fois dans la liste).

```python
def doublons1(lis):
    rep = False
    for i in range(len(lis)):
        for j in range(i + 1, len(lis)):
            if lis[i] == lis[j]: # comparaison
                rep = True
    return rep
```

Montrer que le nombre de comparaisons effectuées par `doublons1(lis)` en fonction de la taille `n` de la liste `lis` est $\frac{n(n-1)}{2}$.

Réponse :

.....
.....

"Question 3"

On considère une autre version `doublons2` de la fonction précédente qui utilise un dictionnaire pour compter les occurrences de chaque élément de la liste `lis` .

```
def doublons2(lis):
    histo = dict()
    rep = False
    for e in lis:
        if e not in histo:
            histo[e] = 1
        else:
            rep = True
    return rep
```

Les opérations de recherche et d'affectation dans un dictionnaire ont un coût temporel constant noté c. Exprimer en fonction de n et c le coût total en opérations élémentaires de `doublons2(lis)` dans le cas où la liste `lis` ne contient pas de doublon. Comparer ce coût à celui de `doublons1(lis)` dans le même cas.

Réponse :

.....
.....
.....

"Question 4"

On considère la fonction de tri par sélection d'une liste d'entiers ci-dessous :

```
def tri_selection(lis):
    for i in range(len(lis)):
        mini = i
        for j in range(i + 1, len(lis)):
            if lis[j] < lis[mini]: # comparaison
                mini = j
        lis[i], lis[mini] = lis[mini], lis[i]
```

3. Entre les opérations d'affectation et de comparaison, lesquelles sont les plus nombreuses dans `tri_selection(lis)` ? Justifier votre réponse.
4. Montrer que le nombre de comparaisons effectuées par `tri_selection(lis)` en fonction de la taille `n` de la liste `lis` est $\frac{n(n-1)}{2}$.

Réponse :

.....

.....

.....

"Question 5"

On considère la fonction de tri par insertion d'une liste d'entiers `lis` ci-dessous :

```
def tri_insertion(lis):
    for i in range(0, len(lis)):
        tmp = lis[i]
        j = i
        while j > 0 and lis[j - 1] > tmp: # comparaison
            lis[j] = lis[j - 1]
            j = j - 1
        lis[j] = tmp
```

5. Donner une liste d'entiers de taille 5 pour laquelle `tri_insertion` effectue le minimum de comparaisons et d'affectations.
6. Donner une liste d'entiers de taille 5 pour laquelle `tri_insertion` effectue le maximum de comparaisons et d'affectations.
7. Montrer que pour trier une liste de taille n, dans le pire cas, le nombre d'opérations `j > 0` du test d'entrée dans la boucle `while` est $\frac{n(n+1)}{2}$.

Réponse :

.....
.....
.....

"Hiérarchie des complexités temporelles"

En général, on classe les algorithmes selon leur complexité temporelle en fonction de la taille de l'entrée `n`. On s'intéressé à la complexité temporelle lorsque `n` devient grand c'est pourquoi on on ne conserve que le terme dominant de la complexité temporelle et on utilise la notation O pour exprimer l'ordre de grandeur de la complexité temporelle asymptotique de l'algorithme.

Type de complexité	Notation O	Exemple d'algorithme
constante	$O(1)$	accès à un élément d'une liste ou d'un dictionnaire
logarithmique	$O(\log_2(n))$, nombre de bits de n en base 2	recherche dichotomique dans une liste triée
linéaire	$O(n)$	parcours d'une liste
quadratique	$O(n^2)$	tri par sélection, `doublons1`
exponentielle	$O(2^n)$	énumération de toutes les séquences de n bits

Les algorithmes de complexité exponentielle sont en général inutilisables pour des entrées de taille même modérée.

"Complexité dans le pire cas"

La complexité temporelle d'un algorithme dépend toujours de la taille n de l'entrée.

Elle peut aussi dépendre de la nature de l'entrée.

Par défaut, on donne la complexité dans le pire cas.

Algorithme de tri	Complexité dans le pire cas	Complexité dans le meilleur des cas
Tri par sélection	Quadratique	Quadratique
Tri par insertion	Quadratique si la liste est triée dans l'ordre inverse	Linéaire si la liste est déjà triée dans le bon ordre

"Exercice 4"

Si on ne connaît pas l'ordre de grandeur de la complexité temporelle, on peut le conjecturer expérimentalement en observant comment évolue le temps d'exécution lorsqu'on mutilplie la taille de l'entrée par un facteur donné. :

Fonction	Si on multiplie la taille de l'entrée par 2, la sortie est multipliée par	Si on multiplie la taille de l'entrée par 10, la sortie est multipliée par
$x \mapsto 10$	...	...
$x \mapsto x$	...	...
$x \mapsto x^2$	...	...
$x \mapsto x^3$	...	...
$x \mapsto \log_2(x)$	...	...
$x \mapsto 2^x$	...	...

"Exercice 5"

On donne les temps d'exécution d'un programme mesurés en fonction de la taille de l'entrée.

Taille de l'entrée	Temps d'exécution en secondes
1000	2

Taille de l'entrée	Temps d'exécution en secondes
2000	16
4000	128
8000	1024

1. Quelle conjecture peut-on faire sur l'ordre de grandeur de la complexité temporelle de cet algorithme ? On l'exprimera en fonction de la taille n de l'entrée.
   Réponse :
   .....
   .....
2. Dans cette question on utilisera l'ordre de grandeur $10^3 \approx 2^{10}$.
   a. Justifier que d'après le tableau précédent le temps d'exécution du programme considéré sur une entrée de taille 10^6 sera de l'ordre de 2^{31} secondes.
   Réponse :
   .....
   .....
   b. L'ordre de grandeur d'une année est de 2^{25} secondes. Combien d'années faudra-t-il attendre pour que le programme s'exécute complètement sur une entrée de taille 10^6 ? On donnera le résultat arrondi à l'entier le plus proche.
   Réponse :
   .....
   .....