TD n° 6 : Complexité des algorithmes

EXERCICE 1 *Notation de Landau*

Simplifier les écritures suivantes :

1. $\Theta(n+1)$
2. $O(3n+3)$
3. $\Theta(\frac{n(n+1)}{2})$
4. $O(n^2e^{42} + 3 \times 2^{n-1})$
5. $O(\log_2(n+1) + \ln(2n^2))$
6. $\Theta(\max(n,m))$

EXERCICE 2 Vrai ou faux ?

1. Une complexité en $\Theta(n)$ est toujours mieux qu'une complexité en $\Theta(n^2)$.
2. Une boucle `for` a une complexité en $\Theta(n)$.
3. La copie d'une liste de taille n prend un temps $\Theta(n)$.
4. Un $\Theta(n)$ est toujours un $O(n)$.
5. On peut considérer que la méthode `.append()` pour les listes s'exécute en $\Theta(1)$.
6. La taille d'un entier n est l'entier n lui-même.

EXERCICE 3 Calculer le nombre d'opérations élémentaires des fonctions suivantes. Que choisir comme taille de l'entrée ? Justifier. Donner enfin la complexité asymptotique, avec les notations de Landau.

PYTHON
```
def machin(n):
    for i in range(n):
        print(i)
    for j in range(n):
        print(j)
```

PYTHON
```
def bidule(n):
    for i in range(n):
        for j in range(i + 1, n):
            print(i, j)
```

PYTHON
```
def truc(n):
    for i in range(n):
        for j in range(n):
            print(i, j)
```

PYTHON
```
def chose(n):
  for i in range(2, n - 2):
    for j in range(i - 2, i + 3):
        print(i, j)
```

EXERCICE 4 *Complexité des opérations usuelles*

Pour les complexités, on donnera un ordre de grandeur, avec les notations de Landau, et non pas un décompte précis des opérations élémentaires.

1. Écrire une fonction `egal` qui vérifie si deux listes sont égales, sans utiliser directement l'égalité sur les listes. Donner sa complexité dans le meilleur et dans le pire cas, en temps et en espace. Que penser de la complexité de `L1 == L2` si `L1` et `L2` sont deux listes ?
2. Écrire une fonction `mininmum` qui renvoie le plus petit élément d'une liste non vide. Donner sa complexité dans le meilleur et dans le pire cas, en temps et en espace. Que penser de la complexité de `min(L)` si `L` est une liste ?

3. On suppose que `L1` et `L2` sont des listes de tailles respectives n_1 et n_2. Pour chaque opération suivante, décrire [1] succinctement l'algorithme qui pourrait être implémenté en PYTHON et en donner la complexité.

   a) `L1 + L2` b) `L1.extend(L2)` c) `L1[i:j]`

EXERCICE 5 *Retour sur la concaténation*

Calculer la complexité des deux fonctions ci-dessous. Comparer le temps mesuré avec le temps estimé sur un ordinateur à 100 millions d'opérations par seconde. Commenter.

PYTHON

```python
def creation_naive(n):
    L = []
    for _ in range(n):
        L = L + [0]

# %timeit creation_naive(10 ** 5)
# 28.8 s ± 293 ms per loop

def creation_avec_append(n):
    L = []
    for _ in range(n):
        L.append(0)

# %timeit creation_avec_append(10 ** 5)
# 11.5 ms ± 85.7 µs per loop
```

EXERCICE 6 *Retour sur la suite de Fibonacci*

Un élève propose la fonction suivante pour déterminer si un entier $k \in \mathbb{N}$ est un entier de Fibonacci, c'est-à-dire s'il existe $n \in \mathbb{N}$ tel que $k = f_n$, avec $(f_n)_{n\geq 0}$ la suite de Fibonacci de premiers termes 0 et 1.

PYTHON

```python
def est_fibo(k):
    """Vérifie si un entier k est un nombre de Fibonacci."""
    # On a f_n + 1 ≥ n pour n ≥ 0, donc si un nombre
    # de fibonacci f_n convient, alors 0 ≤ n ≤ k + 1
    nombres_de_fibonnaci = []
    for i in range(k + 2):
        nombres_de_fibonnaci.append(fibo(i))
    for i in range(k + 2):
        if k in nombres_de_fibonnaci:
            return True
    return False
```

1. Calculer la complexité de cette fonction.
2. Commenter tout ce qu'il y a à commenter.

1. On ne demande pas d'écrire la fonction en PYTHON, mais de décrire l'algorithme en une ou deux phrases.

EXERCICE 7 *Neufs*

On considère la fonction `neufs` ci-dessous, qui prend en argument un entier $n > 0$.

PYTHON
```python
def neufs(n):
    chiffres = []
    while n > 0:
        chiffres.append(n % 10)
        n = n // 10
    res = 0
    for k in range(len(chiffres)):
        i = k
        while i < len(chiffres) and chiffres[i] == 9:
            i += 1
        res = max(res, i - k)
    return res
```

1. Donner la complexité de cette fonction en considérant comme taille de l'entrée l'entier n.
2. Que pourrait-on choisir d'autre pour la taille de l'entrée et quelle serait alors la complexité ?
3. Que fait cette fonction ? Remarquons que nous n'avions pas besoin de comprendre son fonctionnement pour analyser sa complexité.
4. Modifier la fonction pour avoir une complexité en temps en $O(\log n)$.
5. Modifier la fonction précédente pour avoir une complexité en espace en $O(1)$.

EXERCICE 8 *Retour sur la recherche par dichotomie*

Voici une des nombreuses implémentations possibles de la recherche par dichotomie dans un tableau trié :

PYTHON
```python
def recherche_dichotomique(element, tableau):
    """Vérifie si un élément appartient à
    un tableau trié croissant de taille non nulle
    par recherche dichotomique."""
    fin = len(tableau) - 1
    debut = 0
    while debut < fin:
        milieu = (debut + fin) // 2
        if element == tableau[milieu]:
            debut = fin = milieu
        elif element < tableau[milieu]:
            fin = milieu - 1
        else:
            debut = milieu + 1
    return debut == fin and element == tableau[debut]
```

On note t le tableau, n sa taille, x l'élément et d, m, f pour `debut`, `milieu`, `fin`, respectivement.

1. Donner la complexité spatiale de cette fonction.

2. Donner la complexité temporelle de cette fonction dans le meilleur cas, que l'on précisera.
3. Montrer que la complexité temporelle dans le pire cas est en $O(n)$.

Cette borne n'est cependant pas très serrée. On peut dire beaucoup mieux.

4. Montrer par récurrence sur $k \in \mathbb{N}$ qu'après k itérations de la boucle, $f - d < \frac{n}{2^k}$.
5. En déduire que la complexité dans le pire cas est en fait en $O(\log n)$.
6. Identifier un pire cas et montrer que cette borne est serrée et que la complexité dans le pire cas est en $\Theta(\log n)$.

EXERCICE 9 *Retour sur les tris*

Pour chacun des algorithmes de tri ci-dessous, donner précisément sa complexité, en ne considérant comme opérations élémentaires *que* les comparaisons entre éléments des listes. La fonction `echange`, qui échange deux cases dans un tableau, a donc par exemple une complexité nulle. Justifier ensuite précisément la terminaison et la correction de ces deux algorithmes de tri.

1. Tri par sélection

PYTHON
```python
def indice_min(tab, i):
    """Indice du premier élément minimal du tableau tab[i:].
    On suppose que `i` est un indice valable."""
    i_mini = i
    for j in range(i + 1, len(tab)):
        if tab[j] < tab[i_mini]:
            i_mini = j
    return i_mini

def tri_selection(tab):
    """Tri un tableau par la méthode du tri par selection."""
    for i in range(len(tab)):
        echange(tab, i, indice_min(tab, i))
```

2. Tri par insertion

PYTHON
```python
def insere(tab, i):
    """Insère tab[i] à la bonne place dans tab[:i] supposé trié
    en faisant des échanges successifs."""
    k = i
    while k > 0 and tab[k - 1] > tab[k]:
        echange(tab, k - 1, k)
        k -= 1

def tri_insertion(tab):
    """Tri un tableau par la méthode du tri par insertion."""
    for i in range(len(tab)):
        insere(tab, i)
```