TD 12 : Complexité (2)

Exercice 1 *(Le tri par sélection).* On considère l'algorithme suivant qui détermine la position du plus petit élément d'un tableau T parmi les éléments compris entre les indices i et $n-1$:

---

algorithme $min\_a\_partir(T, i)$
  T tableau de nombres de taille n indexé de 0 à $n-1$
  i nombre entier tel que $0 \leqslant i \leqslant n-1$
  Résultat : un entier $j_{\min}$ tel que $i \leqslant j_{\min} \leqslant n-1$ tel que $T[j_{\min}] = \min(T[i], \ldots, T[n-1])$
début algorithme
  $j_{\min} \longleftarrow i$
  pour j **allant de** $i+1$ **à** $n-1$ **faire**
    si $T[j] < T[j_{\min}]$ **alors**
      $j_{\min} \longleftarrow j$
    fin si
  fin pour
  renvoyer $j_{\min}$
fin algorithme

Puis on considère l'algorithme de tri par sélection qui réordonne par ordre croissant les éléments d'un tableau T :

---

algorithme $tri\_selection(T)$
  T tableau de nombres de taille n
  Résultat : le tableau T est trié
début algorithme
  pour i **allant de** 0 **à** $n-2$**:**
    $j \longleftarrow min\_a\_partir(T, i)$
    échanger $T[i]$ et $T[j]$
  fin pour
fin algorithme

(a) Donner un ordre de grandeur du temps d'exécution de $min\_a\_partir(T, i)$ en fonction de i et de n.
(b) Donner un ordre de grandeur du temps d'exécution de $tri\_selection(T)$ en fonction de n.
(c) Pour un tableau T de taille n, donner le nombre de comparaisons ainsi que le nombre d'échanges réalisés par $tri\_selection(T)$ en fonction de n.

Exercice 2 *(Écriture d'un nombre en base* 10*).* On considère l'algorithme suivant qui, à partir d'un enter n, construit la liste dont les éléments sont les chiffres de n dans son écriture en base 10 (par exemple si $n = 107$, le résultat renvoyé est la liste $[7, 0, 1]$).

---

algorithme $liste\_chiffres(n)$
  n nombre entier positif
  L liste des chiffres de n en base 10 (convention : L est vide si $n = 0$)
début algorithme
  $L \longleftarrow$ liste vide
  tant que $n > 0$ **faire**

```
    r ← reste de la division euclidienne de n par 10
    ajouter r à L
    n ← quotient de la division euclidienne de n par 10
  fin tant que
  renvoyer L
fin algorithme
```

(a) On considère un entier n qui s'écrit avec p chiffres en base 10.
- Donner un ordre de grandeur du temps d'exécution de *liste_chiffres*(n) en fonction de p.
- Donner un encadrement de n en fonction de p, en déduire un encadrement de p en fonction de n.
- Donner un ordre de grandeur du temps d'exécution de *liste_chiffres*(n) en fonction de n.

(b) Écrire une fonction **`somcube`**, d'argument n, renvoyant la somme des cubes des chiffres du nombre entier naturel n. Donner un ordre de grandeur du temps d'exécution de cette fonction en fonction de n.

(c) Écrire une fonction **`egaux_somcubes`**, d'argument N, renvoyant la liste des nombres entiers naturels inférieurs à N qui sont égaux à la somme des cubes de leurs chiffres. Donner un ordre de grandeur de son temps d'exécution en fonction de N.

Exercice 3 *(Recherche d'un élément majoritaire)*. On considère l'algorithme suivant qui détermine le nombre de fois où un élément x apparait dans un tableau T :

```
algorithme nbre_apparitions(x, T)
  T tableau de taille n
  x une valeur quelconque
  Résultat : le nombre de fois (éventuellement nul) où x apparait dans T
début algorithme
  nb ← 0
  pour i allant de 0 à n−1 faire
    si T[i] = x alors
      nb ← nb+1
    fin si
  fin pour
  renvoyer nb
fin algorithme
```

On considère ensuite l'algorithme suivant qui détermine un élément majoritaire du tableau (c'est à dire un élément dont le nombre d'apparitions est supérieur au sens large à tous les autres).

```
algorithme majoritaire(T)
  T tableau de taille n
  Résultat : un élément x_maj de T tel que :
    pour tout élément x de T, nbre_apparitions(x_maj, T) ⩾ nbre_apparitions(x, T)
début algorithme
  x_maj ← T[0]
  pour i allant de 1 à n−1 faire
```

```
    si nbre_apparitions(T[i], T) > nbre_apparitions(xmaj, T) alors
      x_maj ← T[i]
    fin si
  fin pour
  renvoyer x
fin algorithme
```

(a) Donner un ordre de grandeur du temps d'exécution de *nbre_apparitions*(x, T) en fonction de n.

(b) Donner un ordre de grandeur du temps d'exécution de *majoritaire*(T) en fonction de n.

(c) On considère l'algorithme suivant :

```
algorithme majoritaire2(T)
  T tableau de taille n
  Résultat : un élément x_maj de T tel que :
    pour tout élément x de T, nbre_apparitions(x_maj, T) ⩾ nbre_apparitions(x, T)
début algorithme
  x_maj ← T[0]
  n_maj ← nbre_apparitions(xmaj, T)
  pour i allant de 1 à n−1 faire
    x ← T[i]
    n_x ← nbre_apparitions(x, T)
    si nx > nmaj alors
      x_maj ← x
      n_max ← n_x
    fin si
  fin pour
  renvoyer x
fin algorithme
```

Comparer les temps d'exécution des algorithmes *majoritaire* et *majoritaire2*.

Corrections

Ex 1. Minimum : $O(n-i-1)$ ou $O(n-i)$, tri sélection : $O(n^2)$. Nombre de comparaisons :

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$$

Nombre d'échanges : $n-1$ dans le pire cas. Si on considère qu'on ne fait l'échange que si l'indice j obtenu est différent de i, alors dans le meilleur cas on ne fait aucun échange.

Ex 2. Le temps d'exécution de liste chiffres est $O(p)$. Si n s'écrit avec p chiffres en base 10, on a $10^{p-1}\leqslant n\leqslant 10^p$ donc $(p-1)\ln(10)\leqslant\ln(n)\leqslant p\leqslant 10$ et ainsi le temps d'exécution en fonction de n est $O(\ln(n))$. On utilise une variable s initialisée à 0 à laquelle on ajoute successivement les cubes des chiffres de n obtenus ainsi.

```
def somcubes(n):
    s = 0
    while n>0:
        s = s+(n%10)**3
        n = n//10
    return s

print(somcubes(12))
```

```
9
```

Ensuite :

```
def egaux_somcubes(N):
    L = []
    for n in range(0,N+1):
        # 0<=n<=N
        if n==somcubes(n):
            L.append(n)
    return L

print(egaux_somcubes(2000))
```

```
[0, 1, 153, 370, 371, 407]
```

Le temps d'exécution de **`somcubes(n)`** est $O(p)$ donc $O(\ln(n))$. L'appel de **`somcubes(n)`** à l'intérieur de **`egaux_somcubes`** a un temps d'exécution de l'ordre de $O(\ln(n))$ et c'est également $O(\ln(N))$ car on réalise cet appel pour tout n tel que $0\leqslant n\leqslant N$. Le temps d'exécution de **`somcubes(N)`** est donc de l'ordre de $O(N\ln(N))$.

Ex 3. Pour le nombre d'apparitions : $O(n)$. Pour majoritaire : $O(n^2)$. Le deuxième algorithme est plus efficace (on ne recalcule pas les nombres d'apparitions), mais l'ordre de grandeur de son temps d'exécution est quand même $O(n)$.