MP 18-19 Feuille Info n° 05 : Algorithmes de tri — Du 12/12/18 au 16/01/19

Exercice 1. Trier à la main par insertion la liste $[12, 5, 8, 16, 13, 42, 19]$

Solution

$$\begin{array}{ccc}
[12,5,8,16,13,42,19] & & [] \\
[5,8,16,13,42,19] & 12 & [] \\
[5,8,16,13,42,19] & & [12] \\
[8,16,13,42,19] & 5 & [12] \\
[8,16,13,42,19] & & [5,12] \\
[16,13,42,19] & 8 & [5,12] \\
[16,13,42,19] & & [5,8,12] \\
[13,42,19] & 16 & [5,8,12] \\
[13,42,19] & & [5,8,12,16] \\
[42,19] & 13 & [5,8,12,16] \\
[42,19] & & [5,8,12,13,16] \\
[19] & 42 & [5,8,12,13,16] \\
[19] & & [5,8,12,13,16,42] \\
[] & 19 & [5,8,12,13,16,42] \\
[] & & [5,8,12,13,16,19,42]
\end{array}$$

Exercice 2. Trier à la main par quicksort la liste $[12, 5, 8, 16, 13, 42, 19]$

Solution

$$[12, 5, 8, 16, 13, 42, 19]$$

On prend 12 comme pivot. On obtient :

$$[5, 8] \quad [12] \quad [16, 13, 42, 19]$$

On prend 5 dans la liste de gauche et 16 dans celle de droite

$$[] \quad [5] \quad [8] \quad [12] \quad [13] \quad [16] \quad [42, 19]$$

On termine avec 42 pour pivot de la dernière liste et 16 dans celle de droite

$$[] \quad [5] \quad [8] \quad [12] \quad [13] \quad [16] \quad [19] \quad [42] \quad []$$

Exercice 3. Calcul de complexité dans les algorithmes basés sur le paradigme "diviser pour régner".
Notations : Soit x un réel. On note $\lfloor x \rfloor$ la partie entière du réel x et $\lceil x \rceil$ le plus petit entier supérieur ou égal à x.
On considère un algorithme récursif qui pour un argument de taille n, fait appel à ce même algorithme pour deux arguments de taille moitié de n (donc comme on travaille avec des tailles entières, $\left\lfloor \frac{n}{2} \right\rfloor$ et $\left\lceil \frac{n}{2} \right\rceil$) effectue un certain nombre d'autres opérations.
On note $C(n)$ la "complexité temporelle" de cet algorithme.

1. Montrer que la complexité vérifie : $\exists\, (b_n)_{n\in\mathbb{N}^*} \in \left(\mathbb{R}^+\right)^2, \forall n \in \mathbb{N}^*, C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + b_n$.
2. On suppose que la suite $(b_n)_{n\in\mathbb{N}^*}$ est croissante positive. Montrer que $(C(n))_{n\in\mathbb{N}^*}$ est croissante.
3. Exprimer $C(2^p)$ en fonction de p et des b_k.
4. On suppose qu'il existe $\lambda \in \mathbb{R}_+^*$ tel que : $\forall n \in \mathbb{N}^*, b_n = \lambda\sqrt{n}$. Montrer que $C(n) = \Theta(n)$.
5. On suppose qu'il existe $\lambda \in \mathbb{R}_+^*$ tel que : $\forall n \in \mathbb{N}^*, b_n = \lambda n$. Montrer que $C(n) = \Theta(n \ln(n))$.
6. On suppose qu'il existe $\lambda \in \mathbb{R}_+^*$ tel que : $\forall n \in \mathbb{N}^*, b_n = \lambda n^2$. Montrer que $C(n) = \Theta\left(n^2\right)$.
7. On suppose qu'il existe $\lambda \in \mathbb{R}_+^*$ tel que : $\forall n \in \mathbb{N}^*, b_n = \lambda n \ln(n)$. Montrer que $C(n) = \Theta\left(n \ln^2(n)\right)$.

Solution

1. OK
2. Récurrence forte.
$C(n+1) - C(n) = \left(\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) - \left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right) + \left(\left(\left\lceil \frac{n+1}{2} \right\rceil\right) - \left(\left\lceil \frac{n}{2} \right\rceil\right)\right) + (b_{n+1} - b_n) \geqslant 0$
3. $C\left(2^{p+1}\right) = 2C\left(2^p\right) + b_{2^{p+1}}$. Ainsi $C\left(2^p\right) = 2^p C(1) + 2^p \sum_{k=1}^{p} \frac{b_{2^k}}{2^k}$

4. Si $b_n = \lambda\sqrt{n}$. Alors $C(2^p) = 2^p C(1) + \lambda 2^p \dfrac{1-2^{-(p+1)/2}}{1-\frac{1}{\sqrt{2}}} \sim \alpha 2^p$.
Si $n \in \mathbb{N}^*$, et $p \in \mathbb{N}$ tel que $2^p \leqslant n \leqslant 2^{p+1}$, on a par croissance, $C(2^p) \leqslant C(n) \leqslant C(2^{p+1})$ et donc $C(n) \sim \alpha n$: $\boxed{C(n) = \Theta(n)}$
5. Si $b_n = \lambda n$. Alors $C(2^p) = 2^p C(1) + \lambda 2^p p \sim \lambda p 2^p$.
Si $n \in \mathbb{N}^*$, et $p \in \mathbb{N}$ tel que $2^p \leqslant n \leqslant 2^{p+1}$, on a par croissance, $C(2^p) \leqslant C(n) \leqslant C(2^{p+1})$ et donc $C(n) \sim \alpha n \ln(n)$: $\boxed{C(n) = \Theta(n\ln(n))}$
6. Si $b_n = \lambda n^2$. Alors $C(2^p) = 2^p C(1) + \lambda 2^p \dfrac{2^p-1}{2-1} \sim \lambda (2^p)^2$.
Si $n \in \mathbb{N}^*$, et $p \in \mathbb{N}$ tel que $2^p \leqslant n \leqslant 2^{p+1}$, on a par croissance, $C(2^p) \leqslant C(n) \leqslant C(2^{p+1})$ et donc $C(n) \sim \lambda n^2$: $\boxed{C(n) = \Theta\left(n^2\right)}$
7. Si $b_n = \lambda n \ln(n)$. Alors $C(2^p) = 2^p C(1) + \lambda 2^p \dfrac{p(p+1)}{2}\ln(2) \sim \alpha 2^p p^2$.
Si $n \in \mathbb{N}^*$, et $p \in \mathbb{N}$ tel que $2^p \leqslant n \leqslant 2^{p+1}$, on a par croissance, $C(2^p) \leqslant C(n) \leqslant C(2^{p+1})$ et donc $C(n) \sim \lambda n (\ln(n))^2$: $\boxed{C(n) = \Theta\left(n (\ln(n))^2\right)}$

Exercice 4. Tri par sélection.
Le tri par sélection consiste, comme pour le tri par insertion, à maintenir à chaque étape i, le sous-tableau $t[0:i]$ trié. En revanche on impose aussi que tous les élḿents restants sont supérieurs à ceux de $t[0:i]$ et donc à $t[i-1]$. De plus, alors que le tri par insertion, on insérait $t[i]$ dans le sous-tableau trié, dans le tri pas sélection, on détermine le minimum de $t[i:n]$ que l'on place à la suite du sous-tableau trié.

1. Ecrire une fonction qui réalise le tri par sélection
2. Démontrer la correction de ce tri
3. Evaluer sa complexité en temps, en espace, dans le meilleur cas, dans le pire cas.

Solution

1.
```
def triselect(tab):
    n = len(tab)
    for i in range(n-1):
        pos = posmini(tab, i)
        tab[i], tab[pos] = tab[pos],
            tab[i]
```

```
def posmini(tab,borninf):
    n = len(tab)
    ind = borninf
    mini = tab[ind]
    for i in range(ind + 1, n):
        if tab[i] < mini :
            ind , mini = i, tab[i]
    return(ind)
```

On a un invariant de boucle :
A la fin de chaque étape, le sous tableau $tab[0:i]$ est trié et contient des éléments tous plus petits que les éléments du reste du tableau .

Exercice 5. Tri à bulles.

1. Montrer que $n-1$ comparaisons et échanges entre éléments consécutifs permettent de placer l'élément maximal en queue d'un tableau à trier.
2. En déduire un nouvel algorithme de tri (appelé tri à bulles) et l'implémenter en Python
3. Calculer sa complexité temporelle

Solution

1. L'algorithme suivant :

```
for i in range(len(tab)-1,0,-1):
 if tab[i-1]> tab[i]:
   x = tab[i]
   tab[i], tab[i-1] = tab[i-1], x
```

de sortie

$[2,5,6,4,8,3,1]$	2
$[2,5,6,4,8,3,1]$	5
$[2,5,4,6,8,3,1]$	4
$[2,5,4,6,8,3,1]$	6
$[2,5,4,6,3,8,1]$	3
$[2,5,4,6,3,1,8]$	1

2.
```python
def tri_bulle(tab):
    n = len(tab)
    for j in range(n, 1, -1):
        for i in range(0,j-1):
            if tab[i+1]< tab[i]:
                x = tab[i]
                tab[i], tab[i+1] = tab[i+1], x
```

3. Comme on a deux boucles imbriquées, on montre aisément que la complexité est en $\Theta\left(n^2\right)$

Exercice 6. Tri stupide.
On considère le "tri" basé sur le principe suivant :
"Tant que le tableau n'est pas trié, le mélanger aléatoirement"

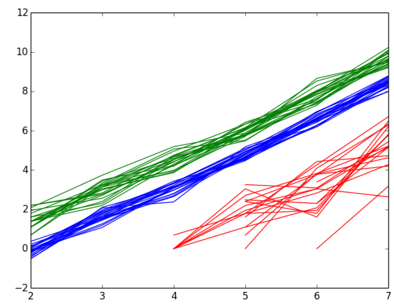
1. Ecrire une fonction testant si un tableau est trié. Evaluer sa complexité (on veut une complexité d'ordre de grandeur négligeable devant $n \ln(n)$).
2. Ecrire une fonction mélangeant les ĺéments d'un tableau. Evaluer sa complexitié. On utilisera la fonction randint(a,b) du module random et qui retourne un entier entre a et b choisi aléatoirement et on pourra utiliser la fonction del.
3. Ecrire une fonction effectuant le tri stupide.
4. Quelle est la complexité dans le meilleur des cas ?, dans le pire des cas ?, en moyenne ?

Solution

```python
def est_trie(tab):
    n = len(tab)
    if n < 2:
        return(True)
    for i in range(n-1):
        if tab[i]> tab[i+1]:
            return(False)
    return(True)
```

```python
import random
def melange(tab):
    tabinter, n = tab[:], len(tab)
    for k in range(n-1,-1,-1):
        ind = random.randint(0,k)
        x = tabinter[ind]
        tab[k] = x
        tabinter.remove(x)
```

—
```python
def tri_stupide(tab):
    compteur = 0
    while not(est_trie(tab)):
        melange(tab)
        compteur += 1
    return(compteur)
```

—


Dans ces courbes, on trace le logarithme du nombre maximum, moyen ou minimum de mélanges nécessaires sur plusieurs fois 20 tests en fonction de la taille du tableau Exemple de sortie :
([2, 3, 4, 5, 6, 7, 8], [6, 20, 119, 743, 4249, 19855, 306810], [0, 0, 0, 13, 9, 78, 40], [0.975, 4.625, 25.65, 119.15, 823.475, 5225.475, 41875.15])

Exercice 7. Tri par dénombrement.
On considère un tableau t qui n'est constitué que de chiffres (donc d'entiers entre 0 et 9).
On considère l'algorithme suivant :

— On crée 10 compteurs dénombrant les occurrences de 0, 1, ..., 9 dans le tableau t
— A partir de ces 10 compteurs, on crée le tableau ayant les mêmes éléments que t (avec les mêmes occurrences pour les chiffres répétés) mais classés dans l'ordre croissant

1. Ecrire cet algorithme en Python
2. Calculer sa complexité

Solution

```python
def tricasier10(tab):
    cpt = [0 for k in range(10)]
    for x in tab:
        cpt[x] += 1
    L=[]
    for k in range(10):
        L += [k]*cpt[k]
    return(L)
```

```python
def tricasier(tab):
    a = min(tab)
    b = max(tab)
    n = b-a+1
    cpt = [0 for k in range(n)]
    for x in tab:
        cpt[x-a] += 1
    L=[]
    for k in range(n):
        L += [k+a]*cpt[k]
    return(L)
```

La complexité est en $O(n+p))$ où p est le nombre de valeurs possibles (écart entre max et min dans le tricasier)