DM_2 Corrigé

1. La fonction `quicksort` est définie récursivement ainsi :

```
def quicksort(L):
    if len(L) <= 1:
        return L[:]
    p = L[0]
    smaller, larger = [], []
    for i in range(1, len(L)):
        if L[i] < p:
            smaller.append(L[i])
        else:
            larger.append(L[i])
    return quicksort(smaller) + [p] + quicksort(larger)
```

   La copie dans le cas **len**`(L) <= 1` évite que des modifications apportées aux résultat de `quicksort(L)` ne modifient aussi `L`.

2. Dans le pire cas, la complexité $C(n)$ est en $\mathcal{O}(n^2)$. Par exemple si la liste est triée, la liste `smaller` est vide et la liste `larger` est de taille $n-1$. Ceci s'appliquant à chaque appel récursif, la relation de récurrence suivante s'applique :

$$C(n) \leqslant cn + C(n-1),$$

   pour une certaine constante c et tout $n > 0$, d'où le résultat.

   Dans le meilleur cas, les deux listes `smaller` et `larger` sont de tailles égales (à une unité près en fonction de la parité), c'est-à-dire le pivot est la médiane de la liste, **à chaque appel récursif**. On retrouve alors la complexité $\mathcal{O}(n\log(n))$ vue en cours.

   Un exemple pour $n = 10$: `[5, 3, 2, 1, 4, 8, 7, 6, 9, 10]`.

3. La fonction `median` ne fait qu'extraire l'élément d'indice $\frac{n}{2}$ de la liste après tri :

```
def median(L):
    sorted_L = quicksort(L)
    return sorted_L[len(L) // 2]
```

   Sa complexité est celle de `quicksort`, soit $\mathcal{O}(n^2)$ dans le pire cas et $\mathcal{O}(n\log(n))$ dans le meilleur cas.

4. Il faut faire attention au fait que l'on renvoie maintenant un élément de la liste. On sait où chercher l'élément en fonction de la taille de la sous-liste `smaller` :

```
def find(i, L):
    if len(L) <= 1:
        return L[0]
    p = L[0]
    smaller, larger = [], []
    for j in range(1, len(L)):
        if L[j] < p:
            smaller.append(L[j])
        else:
            larger.append(L[j])
    if i < len(smaller):
        return find(i, smaller)
    elif i == len(smaller):
        return p
    else:
        return find(i - len(smaller) - 1, larger)
```

5. Il faut bien sûr trouver l'élément de rang la moitié de la longueur de la liste, soit exécuter l'appel :

```
find(len(L) // 2, L)
```

6. Le pire cas de la fonction `find` est atteint lorsque la liste est triée (ainsi `larger` est de taille $n-1$ si `L` est de taille n) et que l'on veut accéder à l'élément de rang $n-1$. On obtient la même relation de récurrence que celle vue en 2 et donc une complexité en $\mathcal{O}(n^2)$.

   Le meilleur cas est atteint lorsque `i` est le rang de l'élément situé dans la première case de `L` : il n'y aura dans ce cas pas d'appel récursif et la complexité est en $\mathcal{O}(n)$.

7. On procède de manière naïve : la taille de la liste étant une constante, la complexité sera de toute manière constante.

```
def median_of_5(L):
    for i in range(5):
        m = L[i]
        smaller = []
        for j in range(5):
            if L[j] < m:
                smaller.append(L[j])
        if len(smaller) == 2:
            return m
```

8. Le cas de base est cette fois-ci un peu différent : le découpage se faisant par groupes de 5 éléments, il faut tenir compte des listes de longueur inférieure à 5. Pour ces listes, on fait appel à la fonction `find` : la taille étant bornée, le résultat sera obtenu en temps constant donc cela n'affectera pas la complexité de la fonction.

   Ne connaissant pas l'indice du pivot, on doit aussi faire attention lors de la répartition entre `smaller` et `larger` afin de ne pas insérer le pivot dans une de ces listes.

```
def find_bis(i, L):
    if len(L) < 5:
        return find(i, L)
    # calcul du pivot
    groups_medians = []
    for j in range(len(L) // 5):
        groups_medians.append(median_of_5(L[5*j:5*j+5]))
    p = find_bis(len(groups_medians) // 2, groups_medians)
    # suite de la fonction
    smaller, larger = [], []
    for j in range(1, len(L)):
        if L[j] < p:
            smaller.append(L[j])
        elif L[j] > p:
            larger.append(L[j])
    if i < len(smaller):
        return find_bis(i, smaller)
    elif i == len(smaller):
        return p
    else:
        return find_bis(i - len(smaller) - 1, larger)
```

9. Comme vu en 5, on fait le bon appel à `find_bis` :

```
def median_bis(L):
    return find_bis(len(L) // 2, L)
```

10. Les différentes étapes du programme sont :
    - le test sur la longueur de `L` en $\mathcal{O}(1)$ car, **en Python**, le calcul de la longueur d'une liste se fait en temps constant ;
    - le calcul du pivot avec un calcul en $\mathcal{O}(n)$ pour les médianes des paquets de 5 éléments et un appel récursif de complexité $C\left(\frac{n}{5}\right)$, comme n est divisible par 5, soit en tout une complexité $\mathcal{O}(n) + C\left(\frac{n}{5}\right)$;
    - la répartition en deux sous-listes en $\mathcal{O}(n)$;
    - au pire, un appel récursif dont on doit encore borner la complexité.

    En somme, il existe une constante c telle que la complexité vérifie :

$$C(n) \leqslant cn + C\left(\frac{n}{5}\right) + f(n),$$

    où $f(n)$ est la complexité de l'éventuel appel récursif.
    Afin de borner la complexité de cet appel récursif, il suffit de majorer la taille des sous-listes `smaller` et `larger` puisque C est croissante. On se contente de majorer celle de `smaller` puisqu'un raisonnement analogue donnera la même borne pour `larger`.
    Parmi les $\frac{n}{5}$ médianes des paquets, la moitié sont strictement inférieures à `p`, puisque `p` est la médiane de ces médianes. Dans les paquets correspondant à ces dernières, au plus 5 éléments sont strictement

inférieurs à `p`. Comme n est divisible par 10, ceci correspond à au plus $5 * \frac{1}{2} * \frac{n}{5} = \frac{n}{2}$ éléments qui iront dans `smaller`. À ces éléments, il faut ajouter ceux qui sont strictement inférieurs à `p` et contenus dans des paquets dont la médiane est supérieure ou égale à `p`. Chacun de ces paquets ne peut contenir qu'au plus 2 éléments strictement inférieurs à `p`, et cela correspond au maximum à la moitié des paquets, ce qui revient à ajouter $2 * \frac{1}{2} * \frac{n}{5} = \frac{n}{5}$ éléments dans `smaller`. On obtient alors au plus $\frac{n}{2} + \frac{n}{5} = \frac{7n}{10}$ éléments dans `smaller`.

Finalement, le dernier appel récursif se fait en complexité inférieure à $C\left(\frac{7n}{10}\right)$ et la complexité globale vérifie :

$$C(n) \leqslant cn + C\left(\frac{n}{5}\right) + C\left(\frac{7n}{10}\right).$$

11. Nous procédons par analyse-synthèse comme indiqué par l'énoncé.
    - Analyse : supposons l'existence de c' et prenons n assez grand.

      Nous avons alors :

      $$\begin{aligned} C(n) &\leqslant cn + c'\left\lfloor \tfrac{n}{5} \right\rfloor + c'\left(7\left\lfloor \tfrac{n}{10} \right\rfloor + 4\right) \\ &\leqslant cn + c'\tfrac{n}{5} + c'\left(\tfrac{7n}{10} + 4\right) \\ &\leqslant c'\left(\tfrac{9n}{10} + 4\right) + cn \\ &\leqslant c'n \text{ dès lors que } 4c' + cn \leqslant \tfrac{c'n}{10}. \end{aligned}$$

      La condition $4c' + cn \leqslant \frac{c'n}{10}$ revient à $c' \geqslant \frac{10cn}{n-40}$ si $n > 40$. Notons que si $n \geqslant 80$, alors $\frac{n}{n-40} \leqslant 2$. Dans ce cas, $c' \geqslant 20c$ conviendrait pour satisfaire la condition.
    - Synthèse : notons $c' = \max\{C(80), 20c\}$.

      Comme C est croissante, pour tout $n \in [\![1, 80]\!]$, $C(n) \leqslant C(80) \leqslant c' \leqslant c'n$. Une récurrence forte en reprenant les inégalités de l'analyse pour démontrer l'hérédité permet alors de conclure que pour tout $n \geqslant 1$, $C(n) \leqslant c'n$.

    Par définition, cette borne signifie que `find_bis` et (donc) `median_bis` sont de complexité linéaire.