**Une correction possible du devoir**

**Exercice 1 – *Pile ou file***

**1. def** prochain(f):

```
    g=creer_file()              # on n'aurait pas besoin d'autre file si on connaissait la longueur de f
    x = defiler(f)
    while not file_vide(f):
        enfiler(g,defiler(f))
    enfiler(f,x)                # certains ont remplacé cette ligne et les deux suivantes par f = g
    while not file_vide(g):     # cela fonctionne en Python mais c'est discutable…
        enfiler(f,defiler(g))   # l'instruction f = g ne crée pas de copie, mais f pointe vers la file g
    return x                    # discutable car on agit sur une file d'une manière non définie dans
                                # l'énoncé, ce n'est a priori pas ce qui est attendu
```

**2. def** rang(f,x):

```
    g=creer_file()
    rang=0
    compteur = 0
    while not file_vide(f):
        y=defiler(f)
        enfiler(g,y)
        compteur += 1
        if y == x and rang == 0:
            rang = compteur
    while not file_vide(g):
        enfiler(f,defiler(g))
    return rang
```

**3. def** inverser(f):

```
    p=creer_pile()
    g=creer_file()
    while not file_vide(f):
        x = defiler(f)
        empiler(p,x)
        enfiler(g,x)
    while not file_vide(g):
        enfiler(f,defiler(g))
    while not pile_vide(p):
        enfiler(g,depiler(p))
    return g
```

**Exercice 2 – *Autour de la médiane***

Attention aux décalages d'indices ! L'énoncé définit les indices du tableau comme allant de 1 à *n* et rappelle gentiment que les indices des listes commencent à 0. Ainsi, *tab*[*a*] dans l'énoncé correspond à *tab*[*a*–1] dans le programme.

Il me semble envisageable de commencer l'exercice 2 par « on ajoute un terme à la liste tab, avec tab.insert[0,0]. Ainsi dans la suite, tab[a..b] sera obtenu sur Python avec tab[a :b]. N'étant pas entièrement convaincu par cette méthode, je ne l'ai pas utilisée dans ce corrigé.

**Question 1**

```
def calculIndiceMax(tab,a,b):
    indiceMax = a
    plusGrand = tab[a-1]            # l'indice a dans le tableau correspond à l'indice a–1 dans la liste
    for i in range(a,b):
        if tab[i] > plusGrand:
            indiceMax = i+1         # même problème, les indices des listes commencent à 0
            plusGrand=tab[i]
    return indiceMax
```

**Question 2**

```
def nombrePlusPetit(tab,a,b,val):
    nombre = 0
    for i in range(a-1,b):
        if tab[i]<=val:
            nombre += 1
    return nombre
```

ou solutions avec les listes en compréhension

```
def nombrePlusPetit(tab,a,b,val):
    return sum(1 for i in tab[a-1:b] if i<=val)

def nombrePlusPetit(tab,a,b,val):
    return sum(i<=val for i in tab[a-1:b])
```

**Question 3**

La forme de l'énoncé (légèrement adapté du sujet de l'option informatique PC X 2012) suggère que l'on agisse directement sur le tableau *tab*. La démarche est celle du tri rapide en place adapté aux tableaux. Traiter correctement cette question ne me semble pas rentable dans le temps limité d'une épreuve de concours. On cherchera donc une solution simple et rapide même si elle n'est pas aussi efficace ni dans l'esprit de l'exercice :

```
def partition2(tab, a, b, indicePivot):
    pivot = tab[indicePivot-1]
    tab1 = [x for x in tab[a-1:b] if x<pivot]
    tab[a-1:b] = tab1+[x for x in tab[a-1:b] if x==pivot]+[x for x in tab[a-1:b] if x>pivot]
    return len(tab1)+1
```

**Solution plus adaptée (mais trop longue à développer) :**

On partitionne le tableau *tab*[*a*..*b*] en utilisant les indices *p*1 et *p*2 tel que tous les éléments de la liste *tab* dont l'indice est compris entre *a* et *p*1 exclu sont strictement inférieurs à pivot, ceux dont l'indice est compris entre *p*1 et *p*2 exclu sont égaux à pivot, et ceux dont l'indice est supérieur ou égal à *p*2 sont strictement supérieurs à pivot.

Avant la boucle **for**, les éléments d'indice strictement inférieur à *p*1 sont strictement inférieurs à pivot, tab[*p*1] = pivot, et *p*2 = *p*1 + 1. Ensuite, on parcourt la liste de *p*2 à *b*–1 (dernier élément) en déplaçant les éléments inférieurs ou égaux au pivot de manière à respecter la partition décrite ci-dessus.

```
def partition(tab, a, b, indicePivot):
    pivot = tab[indicePivot-1]
    # p1 représente le premier indice i tel que tab[i] >= pivot
    p1 = a-1
    while tab[p1] < pivot:
        p1 += 1
    tab[p1],tab[indicePivot-1]=pivot,tab[p1]
    # p2 représente le premier indice i tel que tab[i] > pivot
    p2 = p1 + 1
    # A ce stade, tous les élements de tab[a-1:p1] sont strictement inférieurs à pivot et tab[p1] = pivot
    for i in range(p2,b):
```

```
        if tab[i]==pivot:
            tab[i],tab[p2] = tab[p2],pivot
            p2 += 1
        if tab[i]<pivot:
            tab[i],tab[p1],tab[p2]=tab[p2],tab[i],pivot
            p1 += 1
            p2 += 1
    return p1+1     # l'indice dans le tableau est égal à l'indice dans la liste augmenté de 1
```

**Question 4**

```
def elementK(tab, a, b, k):
    if k==1 and a==b:
        return tab[a-1]             # else facultatif puisqu'il y a un return
    p=tab[a-1]
    i = partition(tab,a,b,a)
    if i-a+1>k:
        return elementK(tab, a, i-1, k)
    elif i-a+1==k:
        return p
    else:
        return elementK(tab, i+1, b, k-i+a-1)
```

**Question 5**

En supposant les éléments distincts (ce qui est le plus souvent le cas), si le pivot est le plus grand élément, on partitionne à chaque étape un tableau de $m$ éléments en un tableau de $m$–1 élément, un tableau contenant uniquement le pivot et un tableau vide (éléments plus grands que le pivot). On élimine donc à chaque étape un seul élément. Il faut donc réaliser $n$–1 étapes. A chaque étape, on appelle la fonction partition qui est en O($n$). La complexité totale est en O($n^2$) (de l'ordre de $n^2/2$ en fait).

**Question 6**

*Il serait beaucoup plus simple de construire une liste contenant les éléments médians des paquets de 5, mais l'algorithme demande implicitement de placer les éléments médians au début du tableau. On doit donc utiliser les indices des éléments. On peut reprendre la fonction elementK qui calcule les médians, en la modifiant pour qu'elle retourne l'indice du médian et non pas sa valeur.*

```
def indiceElementK(tab, a, b, k):   # renvoie l'indice du k-ème élément dans le tableau tab[a..b]
    if k==1 and a==b:
        return a
    i = partition(tab,a,b,a)
    if i-a+1>k:
        return indiceElementK(tab, a, i-1, k)
    elif i-a+1==k:
        return i
    else:
        return indiceElementK(tab, i+1, b, k-i+a-1)

def choixPivot(tab,a,b):
    if b-a<5:
        return elementK(tab,a,b,int((b+1-a)/2))
    else:
        deb = pos = a-1         # pos est l'indice permettant d'effectuer les permutations
```

```
    while deb + 4 < b:      # deb est l'indice marquant le début d'un groupe de 5 éléments
        i = indiceElementK(tab,deb+1,deb+5,3)-1
        tab[pos],tab[i]=tab[i],tab[pos]
        pos += 1
        deb += 5
    if deb < b:
        i = indiceElementK(tab,deb+1,b,int((b+1-deb)/2))-1
        tab[pos],tab[i]=tab[i],tab[pos]
        pos += 1
    return choixPivot(tab,a,pos)    # tab[a-1,pos-1] dans la liste correspond à tab[a..pos] dans le tableau
```

On peut également programmer choixPivot avec une boucle **for**.
Dans ce cas, on commence par une division euclidienne de $(b + 1 - a)$ par 5.