CORRIGÉ ÉPREUVE D'INFO CONCOURS BLANC (d'après X MP-PC 2012)

```python
from copy import copy
import matplotlib.pyplot as plt
import math as m

from copy import copy
import matplotlib.pyplot as plt
import math as m

tab = [3, 2, 5, 8, 1, 34, 21, 6, 9, 14, 8]
tab1 = copy(tab) # le même, pour les essais

##############
# Question 1
##############

def CalculeIndiceMaximum(tab, a, b):
    maxi = tab[a]
    imax = a
    for i in range(a+1, b+1):
        if tab[i] > maxi:
            maxi = tab[i]
            imax = i
    return imax

# essai
# print(CalculeIndiceMaximum(tab, 0, 10))

##############
# Question 2
##############

def NombrePlusPetit(tab, a, b, val):
    nb = 0
    for i in range(a, b+1):
        if tab[i] <= val:
            nb += 1
    return nb

# essai
# print(NombrePlusPetit(tab, 0, 10, 5))

##############
# Question 3
##############

# pour cette question, on a écrit une procédure qui calcule simultanément
# le nombre d'éléments du tableau qui sont strictement plus petits et
# strictement plus grands qu'une valeur val donnée.

def NombrePlusPetitPlusGrand(tab, a, b, val):
    nbmin, nbmax = 0, 0
    for i in range(a, b+1):
        if tab[i] < val:
            nbmin += 1
        else:
            if tab[i] > val:
                nbmax += 1
```

```
    return nbmin, nbmax

def MedianNaif(tab, a, b):
    nsurdeux = (b - a + 1) // 2
    for i in range(a, b+1):
        nbmin, nbmax = NombrePlusPetitPlusGrand(tab, a, b, tab[i])
        if nbmin <= nsurdeux and nbmax <= nsurdeux:
            return tab[i]

# essai
# print(MedianNaif(tab, 0, 10))

# Complexité

# Dans la procédure NombrePlusPetitPlusGrand, il y a une boucle effectuée n=b-a+1 fois
# Dans le meilleur des cas, le médian est le 1er élément du tableau, donc la boucle
# dans MedianNaif n'est effectuée qu'1 fois ---> complexité O(n)
# Dans le pire des cas, le médian est à la dernière place et cette boucle est
# effectuée n fois --> complexité O(n^2)
# Si le médian est à la i-ème place, cette boucle est effectuée i fois donc le
# nombre de boucles en moyenne dans MedianNaif est 1/n*sum(i, i=1 à n)=(n+1)/2
# d'où une complexité totale de O(n(n+1)/2)=O(n^2).

##############
# Question 4
##############

def Fusion(A, B):
    C = []
    la = len(A)
    lb = len(B)
    i = 0
    j = 0
    while i < la and j < lb: # on continue tant que les deux listes ne sont pas vides
        ai, bj = A[i], B[j]
        if ai < bj :
            C.append(ai)
            i += 1
        else :
            C.append(bj) ;
            j += 1
    # à ce stade, une (au plus) des deux listes est vide, et on ajoute les éléments de l'autre list
    if i == la :
        C.extend(B[j:])
    else :
        C.extend(A[i:])
    return C

def TriFusion(L):
    # tri de complexité O(nln(n))
    n = len(L)
    if n == 1:
        return L
    # il est plus efficace en fait d'utiliser ici le tri par insertion pour n "petit"
    m = n // 2
    #    on utilise ensuite le slicing pour définir les sous-listes, puis on les trie et on renvoie
    return Fusion( TriFusion(L[:m]) , TriFusion(L[m:]) )

def MedianAvecTri(tab, a, b):
    L = TriFusion(tab[a:b+1])
    return L[len(L)//2]
```

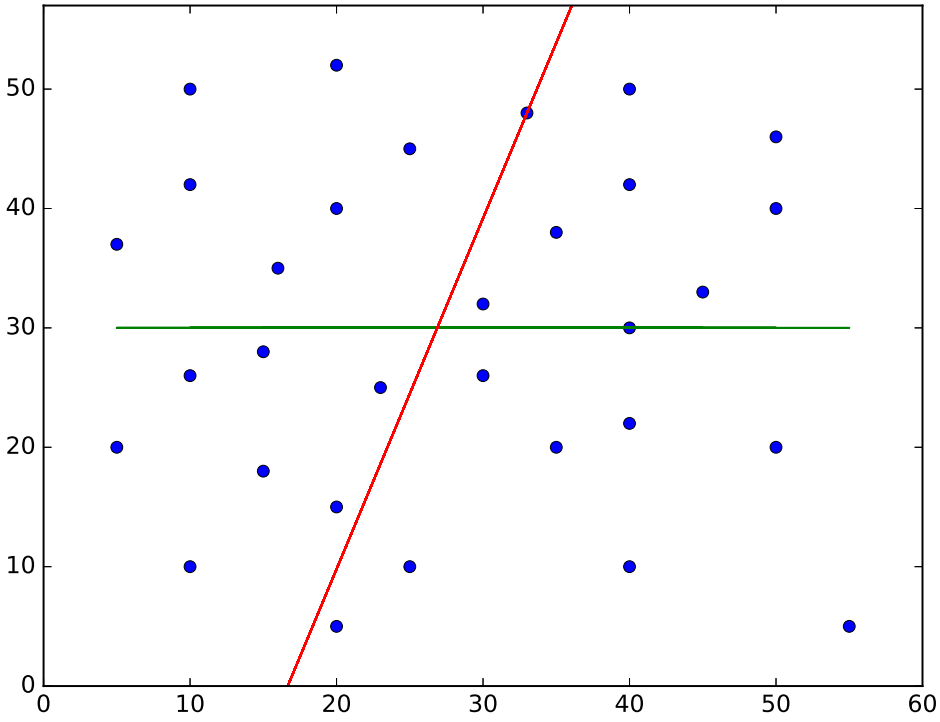
```python

# essai
# print(MedianAvecTri(tab, 0, 10))

##############
# Question 5
##############

def Partition(tab, a, b, indicePivot):
    n = b - a + 1
    nouv_liste = [0]*n
    ig, id = 0, n-1
    p = tab[indicePivot]
    for i in range(a, b+1):
        if tab[i] < p:
            nouv_liste[ig] = tab[i]
            ig += 1
        else:
            if tab[i] > p:
                nouv_liste[id] = tab[i]
                id -= 1
    for i in range(ig, id + 1):
        nouv_liste[i] = p
    tab[a:b+1] = copy(nouv_liste)
    return ig + a

# essai
# print(Partition(tab, 0, 10, 3), tab)

# Ci-dessous une version de Partition qui fait directement le travail en place
# Il s'agit de l'algorithme de Dijkstra

def Partition2(tab, a, b, indicePivot):
    ig, im, id = a-1, a, b
    # indices gauche, milieu, droite pour les 3 parties du tableau
    # les éléments d'indices <= à ig sont <p
    # les éléments d'indices i tels que ig <i <im sont égaux à p
    # ceux d'indices i tels que i>id sont >p
    p = tab[indicePivot]
    print(tab)
    while im <= id:
        if tab[im] == p:
            im += 1
        elif tab[im] > p:
            tab[im], tab[id] = tab[id], tab[im]
            id -= 1
        else:
            ig += 1
            tab[ig], tab[im] = tab[im], tab[ig]
            im += 1
    return ig + 1

# essai
# print(Partition2(tab1, 0, 10, 3), tab1)

##############
# Question 6
##############

def ElementK(tab, a, b, k):
    if a == b:
```

```
            return tab[a]
        i = Partition(tab, a, b, a)
        if i-a+1 > k:
            return ElementK(tab, a, i-1, k)
        elif i-a+1 == k:
            return tab[i]
        else:
            return ElementK(tab, i+1, b, k-i+a-1)

    # essai
    # print(ElementK(tab, 0, 10, 5))

    ##############
    # Question 7
    ##############

    # La fonction partition appliquée à un tableau de taille n fait n comparaisons; dans le pire
    # des cas évoqué par l'énoncé, la fonction ElementK va être successivement appliquée à un
    # tableau de n éléments, puis de n-1, puis de n-2 etc... Le nombre de comparaisons sera donc
    # de n + (n-1) + (n-2) + .... = n(n+1)/2; la complexité est en O(n^2) !

    ##############
    # Question 8
    ##############

    def IndiceMedianNaif(tab, a, b):
        # renvoie l'indice de l'élément médian et non plus sa valeur
        nsurdeux = (b - a + 1) // 2
        for i in range(a, b+1):
            nbmin, nbmax = NombrePlusPetitPlusGrand(tab, a, b, tab[i])
            if nbmin <= nsurdeux and nbmax <= nsurdeux:
                return i

    def ChoixPivot(tab, a, b):
        n = b - a + 1
        nb_paquets = n // 5
        reste = n % 5
        if nb_paquets == 0:
            return IndiceMedianNaif(tab, a, b)
        liste_medians = []
        for i in range(0, nb_paquets):
            liste_medians.append(IndiceMedianNaif(tab, a+5*i, a+5*i+4))
        if reste != 0:
            liste_medians.append(IndiceMedianNaif(tab, a+5*nb_paquets, b))
        for i in range(len(liste_medians)):
            tab[a+i], tab[liste_medians[i]] = tab[liste_medians[i]], tab[a+i]
        return ChoixPivot(tab, a, a + len(liste_medians)-1)

    # essai
    # print(ChoixPivot(tab,0,10))

    def ElementK2(tab, a, b, k):
        if a == b:
            return tab[a]
        indicePivot = ChoixPivot(tab, a, b)
        i = Partition(tab, a, b, indicePivot)
        if i-a+1 > k:
            return ElementK2(tab, a, i-1, k)
        elif i-a+1 == k:
            return tab[i]
        else:
```

```
241         return ElementK2(tab, i+1, b, k-i+a-1)
242
243 def IndiceMedian(tab, a, b):
244     tab2 = copy(tab)
245     median = ElementK2(tab2, a, b, (b-a+1)//2+1)
246     # +1 car le i-ème élément est celui d'indice i-1 en Python
247     for i in range(a, b+1):
248         if tab[i] == median:
249             return i
250
251 # essai
252 # print(ElementK2(tab, 0, 10, 6))
253 # print(IndiceMedianNaif(tab,0,10))
254 # print(IndiceMedian(tab,0,10))
255
256 ##############
257 # Question 9
258 ##############
259
260 def CoupeY(tabX, tabY):
261     return tabY[IndiceMedian(tabY, 0, len(tabY) - 1)]
262
263 ##############
264 # Question 10
265 ##############
266
267 def Angle(x, y, x2, y2):
268     # renvoie l'angle des droites modulo pi
269     ux = x2 - x
270     uy = y2 - y
271     theta = 2*m.atan(uy/(ux + m.sqrt(ux**2 + uy**2)))
272     if theta < 0:
273         theta += m.pi
274     return theta
275
276 def DemiDroiteMedianeSup(tabX, tabY, x, y):
277     angles_au_dessus = []
278     for i in range(len(tabY)):
279         if tabY[i] > y:
280             angles_au_dessus.append(Angle(x, y, tabX[i], tabY[i]))
281     return angles_au_dessus[IndiceMedian(angles_au_dessus, 0, len(angles_au_dessus)-1)]
282
283 ##############
284 # Question 11
285 ##############
286
287 def VerifieAngleSecondeDroite(tabX, tabY, x, y, theta):
288     angles_en_dessous = []
289     for i in range(len(tabY)):
290         if tabY[i] < y:
291             angles_en_dessous.append(Angle(x, y, tabX[i], tabY[i]))
292     l = len(angles_en_dessous)
293     lsurdeux = m.ceil(l/2)
294     nbinf, nbsup = NombrePlusPetitPlusGrand(angles_en_dessous, 0, l-1, theta)
295     if nbinf > lsurdeux:
296         return -1
297     if nbsup > lsurdeux:
298         return 1
299     return 0
300
301 ##############
```

```
# Question 12
##############

def SecondeMediane(tabX, tabY, y):

    def essaye(gauche, droite):
        # cette fonction récursive DOIT s'arréter puisque la théorie dit
        # qu'il existe une solution au problème.
        x = (gauche + droite) / 2
        theta = DemiDroiteMedianeSup(tabX, tabY, x, y)
        result = VerifieAngleSecondeDroite(tabX, tabY, x, y, theta)
        if result == 0:
            return x, theta
        elif result == 1:
            return essaye(x, droite)
        else:
            return essaye(gauche, x)

    xmin, xmax = min(tabX), max(tabX)
    return essaye(xmin, xmax)

# le programme pour essayer tout cela
# données des exemples
Points =[ [10,50], [33,48], [40,50], [40,42], [45,33], [16,35], [23,25], [50,40],\
          [15,18], [30,26], [55,5], [20,40], [40,30], [25,45], [40,10], [5,20],\
          [35,38], [5,37], [50,46], [35,20], [25,10], [10,10], [10,42], \
          [20,15], [20,52], [10,26], [30,32], [50,20], [40,22], [20,5], [15,28]]
tabX = [t[0] for t in Points]
tabY = [t[1] for t in Points]

y = CoupeY(tabX, tabY)
x, theta = SecondeMediane(tabX, tabY, y)

plt.plot(tabX, tabY, "o")
DroiteH = [y]*len(tabX)
SecondeDroite=[m.tan(theta) * (X - x) + y for X in tabX]
plt.plot(tabX, DroiteH)
plt.plot(tabX, SecondeDroite)
plt.xlim(min(tabX)-5,max(tabX)+5)
plt.ylim(min(tabY)-5,max(tabY)+5)
plt.show()
```

Et voici la figure obtenue :



Vous trouverez ci-après un extrait de l'excellent livre **Data Structures and Algorithms** de **Aho, Hopcroft, Ullman** où est expliqué l'algorithme de recherche du médian en temps linéaire dont s'inspire directement l'énoncé.

$$\left(\frac{k_1}{k_1+k_2}\right)\log(k_1) + \left(\frac{k_2}{k_1+k_2}\right)\log(k_2)+1.$$

Since $k_1 + k_2 = k$, we can express the average depth as

$$\frac{1}{k}(k_1\log(2k_1) + k_2\log(2k_2)) \qquad (8.13)$$

The reader can check that when $k_1 = k_2 = k/2$, (8.13) has value log$k$. The reader must show that (8.13) has a minimum when $k_1 = k_2$, given the constraint that $k_1 + k_2 = k$. We leave this proof as an exercise to the reader skilled in differential calculus. Granted that (8.13) has a minimum value of log$k$, we see that T was not a counterexample at all.

8.7 Order Statistics

The problem of computing *order statistics* is, given a list of n records and an integer k, to find the key of the record that is k^{th} in the sorted order of the records. In general, we refer to this problem as "finding the k^{th} out of n." Special cases occur when $k = 1$ (finding the minimum), $k = n$ (finding the maximum), and the case where n is odd and $k = (n + 1)/2$, called finding the *median*.

Certain cases of the problem are quite easy to solve in linear time. For example, finding the minimum of n elements in $O(n)$ time requires no special insight. As we mentioned in connection with heapsort, if $k \leq n/\log n$ then we can find the k^{th} out of n by building a heap, which takes $O(n)$ time, and then selecting the k smallest elements in $O(n + k \log n) = O(n)$ time. Symmetrically, we can find the k^{th} of n in $O(n)$ time when $k \geq n - n/\log n$.

A Quicksort Variation

Probably the fastest way, on the average, to find the k^{th} out of n is to use a recursive procedure similar to quicksort, call it *select*(i, j, k), that finds the k^{th} element among $A[i], \ldots, A[j]$ within some larger array $A[1], \ldots, A[n]$. The basic steps of *select* are:

1. Pick a pivot element, say v.

2. Use the procedure *partition* from Fig. 8.13 to divide $A[i], \ldots, A[j]$ into two groups: $A[i], \ldots, A[m-1]$ with keys less than v, and $A[m], \ldots, A[j]$ with keys v or greater.

3. If $k \leq m - i$, so the k^{th} among $A[i], \ldots, A[j]$ is in the first group, then call *select*$(i, m-1, k)$. If $k > m - i$, then call *select*$(m, j, k - m + i)$.

Eventually we find that *select*(i, j, k) is called, where all of $A[i], \ldots, A[j]$ have the same key (usually because $j=i$). Then we know the desired key; it is the key of any of these records.

As with quicksort, the function *select* outlined above can take $\Omega(n^2)$ time in the worst case. For example, suppose we are looking for the first element, but by bad luck, the pivot is always the highest available key. However, on the average, *select* is even faster than quicksort; it takes $O(n)$ time. The principle is that while quicksort calls itself twice, *select* calls itself only once. We could analyze *select* as we did quicksort, but the mathematics is again complex, and a simple intuitive argument should be convincing. On the average, *select* calls itself on a subarray half as long as the subarray it was given. Suppose that to be conservative, we say that each call is on an array 9/10 as long as the previous call. Then if $T(n)$ is the time spent by *select* on an array of length n, we know that for some constant c we have

$$T(n) \leq T(\frac{9}{10}n) + cn \qquad (8.14)$$

Using techniques from the next chapter, we can show that the solution to (8.14) is $T(n) = O(n)$.

A Worst-Case Linear Method for Finding Order Statistics

To guarantee that a function like *select* has worst case, rather than average case, complexity $O(n)$, it suffices to show that in linear time we can find some pivot that is guaranteed to be some positive fraction of the way from either end. For example, the solution to (8.14) shows that if the pivot of n elements is never less than the $\frac{n}{10}^{\text{th}}$ element, nor greater than the $\frac{9n}{10}^{\text{th}}$ element, so the recursive call to *select* is on at most nine-tenths of the array, then this variant of *select* will be $O(n)$ in the worst

case.

The trick to finding a good pivot is contained in the following two steps.

1. Divide the *n* elements into groups of 5, leaving aside between 0 and 4 elements that cannot be placed in a group. Sort each group of 5 by any algorithm and take the middle element from each group, a total of $\lfloor n/5 \rfloor$ elements.

2. Use *select* to find the median of these $\lfloor n/5 \rfloor$ elements, or if $\lfloor n/5 \rfloor$ is even, an element in a position as close to the middle as possible. Whether $\lfloor n/5 \rfloor$ is even or odd, the desired element is in position $\lfloor \frac{n+5}{10} \rfloor$.

This pivot is far from the extremes, provided that not too many records have the pivot for key.† To simplify matters let us temporarily assume that all keys are different. Then we claim the chosen pivot, the $\lfloor \frac{n+5}{10} \rfloor^{\text{th}}$ element of the $\lfloor \frac{n}{5} \rfloor$ middle elements out of groups of 5 is greater than at least $3\lfloor \frac{n-5}{10} \rfloor$ of the *n* elements. For it exceeds $\lfloor \frac{n-5}{10} \rfloor$ of the middle elements, and each of those exceeds two more, from the five of which it is the middle. If $n \geq 75$, then $3\lfloor \frac{n-5}{10} \rfloor$ is at least $n/4$. Similarly, we may check that the chosen pivot is less than or equal to at least $3\lfloor \frac{n-5}{10} \rfloor$ elements, so for $n \geq 75$, the pivot lies between the 1/4 and 3/4 point in the sorted order. Most importantly, when we use the pivot to partition the *n* elements, the k^{th} element will be isolated to within a range of at most $3n/4$ of the elements. A sketch of the complete algorithm is given in Fig. 8.25; as for the sorting algorithms it assumes an array $A[1], \ldots, A[n]$ of recordtype, and that recordtype has a field *key* of type keytype. The algorithm to find the k^{th} element is just a call to *select*(1,*n*,*k*), of course.

```
    function select ( i, j, k: integer ): keytype;
      { returns the key of the k^th element in sorted order
        among A[i], . . . ,A[j] }
      var
        m: integer; { used as index }
      begin
(1)       if j-i < 74 then begin { too few to use select recursively }
```

```
(2)             sort A[i], . . . ,A[j] by some simple algorithm;
(3)             return (A[i + k - 1 ].key)
            end
            else begin { apply select recursively }
(4)             for m := 0 to (j - i - 4) div 5 do
                  { get the middle elements of groups of 5
                    into A[i], A[i + 1], . . . }
(5)               find the third element among A[i + 5*m] through
                    A[i + 5*m + 4] and swap it with A[i +
m];
(6)             pivot := select(i, i+(j-i-4) div 5, (j -
i - 4) div 10);
                  { find median of middle elements. Note j - i - 4
                    here is n - 5 in the informal description above }
(7)             m := partition(i, j, pivot);
(8)             if k <= m - i then
(9)               return (select(i, m - 1, k))
                else
(10)              return (select(m, j, k-(m - i)))
            end
        end; { select }
```

Fig. 8.25. Worst-case linear algorithm for finding k^{th} element.

To analyze the running time of *select* of Fig. 8.25, let n be $j - i + 1$. Lines (2) and (3) are only executed if n is 74 or less. Thus, even though step (2) may take $O(n^2)$ steps in general, there is some constant c_1, however large, such that for $n \leq 74$, lines (1 - 3) take no more than c_1 time.

Now consider lines (4 - 10). Line (7), the partition step, was shown in connection with quicksort to take $O(n)$ time. The loop of lines (4 - 5) is iterated about $n/5$ times, and each execution of line (5), requiring the sorting of 5 elements, takes some constant time, so the loop takes $O(n)$ time as a whole.

Let $T(n)$ be the time taken by a call to *select* on n elements. Then line (6) takes at most $T(n/5)$ time. Since $n \geq 75$ when line (10) is reached, and we have argued that if $n \geq 75$, at most $3n/4$ elements are less than the pivot, and at most $3n/4$ are equal to or greater than the pivot, we know that line (9) or (10) takes time at most $T(3n/4)$. Hence, for some constants c_1 and c_2 we have

$$T(n) \leq \begin{cases} c_1 & \text{if } n \leq 74 \\ c_2 n + T(n/5) + T(3n/4) & \text{if } n \geq 75 \end{cases} \qquad (8.15)$$

The term $c_2 n$ in (8.15) represents lines (1), (4), (5), and (7); the term $T(n/5)$ comes from line (6), and $T(3n/4)$ represents lines (9) and (10).

We shall show that $T(n)$ in (8.15) is $O(n)$. Before we proceed, the reader should now appreciate that the "magic number" 5, the size of the groups in line (5), and our selection of $n = 75$ as the breakpoint below which we did not use *select* recursively, were designed so the arguments $n/5$ and $3n/4$ of T in (8.15) would sum to something less than n. Other choices of these parameters could be made, but observe when we solve (8.15) how the fact that $1/5 + 3/4 < 1$ is needed to prove linearity.

Equation (8.15) can be solved by guessing a solution and verifying by induction that it holds for all n. We shall chose a solution of the form cn, for some constant c. If we pick $c \geq c_1$, we know $T(n) \leq cn$ for all n between 1 and 74, so consider the case $n \geq 75$. Assume by induction that $T(m) \leq cm$ for $m<n$. Then by (8.15),

$$T(n) \leq c_2 n + cn/5 + 3cn/4 \leq c_2 n + 19cn/20 \qquad (8.16)$$

If we pick $c = \max(c_1, 20c_2)$, then by (8.16), we have $T(n) \leq cn/20 + cn/5 + 3cn/4 = cn$, which we needed to show. Thus, $T(n)$ is $O(n)$.

The Case Where Some Equalities Among Keys Exist

Recall that we assumed no two keys were equal in Fig. 8.25. The reason this assumption was needed is that otherwise line (7) cannot be shown to partition A into blocks of size at most $3n/4$. The modification we need to handle key equalities is to add, after step (7), another step like partition, that groups together all records with key equal to the pivot. Say there are $p \geq 1$ such keys. If $m - i \leq k \leq m - i + p$, then no recursion is necessary; simply return $A[m].key$. Otherwise, line (8) is unchanged, but line (10) calls $select(m + p, j, k - (m - i) - p)$.

Exercises