Avant de se lancer dans l'écriture des programmes, il faut bien lire l'énoncé : le fait que la valeur de $a[i]$ soit "un entier positif quelconque" empêche toute boucle sur l'ensemble des valeurs contenues dans le tableau a. L'exemple du numéro de sécurité sociale, qui en France comporte 13 chiffres décimaux, est révélateur puisque 10^{13} octets reste une taille de données prohibitive, même avec les gros ordinateurs actuels. De même pour le temps de calcul associé.

Lorsque l'énoncé demande d'imprimer des valeurs (sortie un peu surannée), je choisis de renvoyer une liste contenant les valeurs à imprimer, ce qui laisse à l'utilisateur le choix de ce qu'il veut faire du résultat !

I. Dépouillement du premier tour

Question 1 Puisqu'une complexité en $O\left(n^2\right)$ (*temps quadratique*) est tolérée, une double boucle calculant les scores des candidats successifs, dans l'ordre où ils apparaissent dans a fera l'affaire. Cela dit, il vaut toujours mieux éviter de faire des calculs pour rien ! Pour cela, on utilise classiquement des boucles `while`, mais l'instruction `return` de Python permet de sortir d'une boucle `for` dès que l'on a trouvé un gagnant ou que l'on est sûr de ne jamais en trouver ; à ce propos, il est inutile d'aller au bout du tableau, puisque si aucun candidat de la première moitié n'a été déclaré gagnant, les nouveaux candidats ne figurant que dans la seconde moitié ne peuvent pas être majoritaires.
Pour les valeurs successives de i, je calcule le score du candidat $a[i]$ à l'aide d'une seconde boucle `for`. Pour cette dernière, deux remarques pour l'optimiser :

- d'une part je peux commencer le décompte à partir de l'indice i dans le tableau (inutile de repartir du début, puisque les candidats $a[0], \ldots, a[i-1]$ ont déjà été traités et ne sont pas majoritaires, sinon le résultat aurait déjà été renvoyé) ;
- d'autre part je peux arrêter de compter dès que le score atteint la majorité absolue, je peux alors renvoyer le numéro du gagnant.

Justifions pour finir la condition de sortie $2i \geq n$: lorsqu'on n'a pas trouvé de gagnant dans $a[0:i]$, le score maximum de $a[i]$ parmi $a[i], \ldots, a[n-1]$ est $n-i$, or si $n-i \leq i$, alors $2(n-i) \leq n$ et donc $a[i]$ ne peut être majoritaire.

```
def gagnant_tour1(a):
    n=len(a)
    majorite=1+n//2
    for i in range(n//2):
        score=1
        for j in range(i+1,n):
            if a[j]==a[i]:
                score+=1
        if score>=majorite:
            return a[i]
    return -1
```

Compte tenu de la double boucle, la complexité est bien un $O\left(n^2\right)$.
Cette première version, malgré les astuces mises en œuvre pour l'optimiser, a le défaut de recalculer les scores (partiels) de candidats déjà traités, lorsqu'ils ne sont pas majoritaires. On pourrait envisager de tenir à jour la liste des candidats traités, mais cela compliquerait le programme sans améliorer la complexité dans le pire des cas, par exemple si tous les candidats sont distincts et reçoivent une voix chacun !

Question 2 Lorsque le tableau est trié par ordre croissant, un candidat majoritaire se reconnaîtra à un "palier", tranche de tableau du type $a[i:j]$ de valeurs toutes égales à $a[i]$ avec $j-i > m$, où m vaut $n//2$. En effet il y a $j-i$ valeurs dans ledit sous-tableau. Inutile ici de balayer toutes les valeurs du sous-tableau pour calculer le score de $a[i]$, il suffit de comparer $a[i+m]$ à $a[i]$ lorsqu'on rencontre $a[i]$ pour la première fois en parcourant le tableau : si $a[i+m] = a[i]$, alors $a[i]$ est majoritaire (le tableau étant trié, les $m+1$ valeurs de $a[i]$ à $a[i+m]$ sont toutes égales à $a[i]$!) ; sinon, le score de $a[i]$ est au plus égal à m et donc $a[i]$ n'est pas majoritaire.
Mêmes remarques qu'à la question 1 pour le contrôle de la boucle `for`.

```
def gagnant_tour1_bis(a):
    n=len(a)
    m=n//2
    for i in range(m):
        if a[i+m]==a[i]:
            return a[i]
        i+=1
    return -1
```

On pourrait insérer une seconde boucle `while` pour ne pas traiter plusieurs fois les candidats minoritaires figurant plusieurs fois ; cela ferait économiser quelques additions mais compliquerait le programme sans changer l'ordre de grandeur de la complexité, qui est un $O(n)$.

Question 3 À partir d'un tableau désordonné, il sera plus efficace de le trier puis d'utiliser le programme de la question 2, ce qui donnera une complexité totale en $O(n \ln n)$, puisque n est négligeable devant $n \ln n$.

II. Dépouillement du deuxième tour

Question 4

- *Première version, avec tableau auxiliaire* : on peut décider de dépouiller les votes en stockant le nombre de voix de chaque candidat dans un tableau auxiliaire *score*. On doit déclarer un tableau de taille n, puisque dans le pire des cas tous les candidats sont distincts et ont une unique voix ! Pour prendre en compte les candidats recevant plusieurs voix, on initialise le tableau *score* à 0 et, pour chaque vote $a[i]$, on incrémente $score[i\_min]$, où i_min est le plus petit des indices k tels que $a[k] = a[i]$. Ainsi tous les votes pour le candidat de numéro $a[i]$ seront comptabilisés dans la même case du tableau *score*. On profite du parcours pour tenir à jour la variable $score\_max$, qui contiendra le score du ou des vainqueurs. Il suffira pour conclure de parcourir les éléments du tableau *score* pour lister les gagnants. Par construction chacun n'apparaîtra qu'une fois.

```
def gagnants_tour2v1(a):
    n=len(a)
    score=[0]*n
    score_max=0
    for i in range(n):
        # on commence par déterminer l'indice du premier qui a voté pour a[i]
        i_min=0
        while a[i_min]!=a[i]:
            i_min+=1
        # on ajoute 1 vote dans la bonne pile de bulletins
        score[i_min]+=1;
        # on en profite pour tenir à jour le meilleur score
        if score[i_min]>score_max:
            score_max=score[i_min]
    gagnants=[]
    for i in range(n):
        if score[i]==score_max:
            gagnants.append(a[i])
    return gagnants
```

Ici la complexité est un $O(np)$ où p est le nombre de candidats distincts ayant obtenu au moins une voix, donc a fortiori un $O\left(n^2\right)$.

- *Deuxième version, sans tableau auxiliaire* : dans le contexte étudié, n risque d'être très grand et créer un tableau annexe de taille n peut encombrer la mémoire. On peut donc préférer une méthode peut-être plus coûteuse en temps (tout en restant quadratique), mais sans création de tableau annexe. Pour chaque i, on compte le nombre de voix obtenues par $a[i]$. On stocke dans une variable *score_max* le nombre maximum de voix recueillies par un candidat. Puis l'on fait une deuxième passe où on recompte le nombre de voix obtenues par chaque $a[i]$; si ce nombre est égal à *score_max*, il fait partie des gagnants et on mémorise $a[i]$. Pour que les noms des gagnants ne soient affichés qu'une seule fois, on calculera le nombre exact de voix recueillies par $a[i]$ uniquement à sa première apparition, en comptant en fait le nombre de j tels que $a[j] = a[i]$, seulement pour $i \leq j < n$. Ainsi, dans les apparitions ultérieures de $a[i]$, ce nombre sera strictement inférieur à *score_max* et $a[i]$ ne sera pas de nouveau considéré comme gagnant.

  Cette fois-ci la complexité est vraiment de l'ordre de n^2 !

```
def gagnants_tour2(a):
    n=len(a)
    score_max=0
    for i in range(n):
        score=1
        for j in range(i+1,n):
            if a[j]==a[i]:
                score+=1
        if score>score_max:
            score_max=score
    gagnants=[]
    for i in range(n):
        score=1
        for j in range(i+1,n):
            if a[j]==a[i]:
                score+=1
        if score==score_max:
            gagnants.append(a[i])
    return gagnants
```

```
def gagnants_tour2_bis(a):
    n=len(a)
    score_max=0
    d=0
    while d<n:
        f=d
        while f<n and a[f]==a[d]:
            f+=1
        if f-d>score_max:
            score_max=f-d
        d=f;
    gagnants=[]
    d=0
    while d<n:
        f=d
        while f<n and a[f]==a[d]:
            f+=1
        if f-d==score_max:
            gagnants.append(a[d])
        d=f
    return gagnants
```

Question 5 Ici, aucun intérêt à utiliser un tableau auxiliaire, puisque la complexité sera de toute façon de l'ordre de n (voir le programme à droite ci-dessus).

On parcourt le tableau trié en calculant la longueur de chaque "palier" (sous-tableau constant), cette longueur correspondant au score du candidat. Une boucle `while` s'impose, puisqu'on avance par sauts dans le tableau, la longueur de chaque saut étant déterminée par une autre boucle `while` destinée à calculer la longueur du palier. Noter que, pour chaque nouveau candidat étudié, l'indice d correspond au début du palier, la variable f contenant en sortie de la boucle interne l'indice de fin du palier augmenté de 1 (c'est, soit n si l'on est au bout du tableau, soit le premier indice où l'on trouve un nouveau candidat). Donc $f - d$ est bien le score du candidat $a[d]$ et f est bien la valeur à donner à d avant de boucler. Le premier parcours permet de déterminer le score maximum. Le second parcours permet de détecter et de mémoriser les gagnants.

Question 6 Comme dans la première partie, un tri en $O(n \ln n)$ suivi d'un dépouillement en $O(n)$ donne un temps global en $O(n \ln n)$.

III. Dépouillement rapide du premier tour

Question 7 Je suppose x majoritaire dans E. Je note n le nombre de valeurs de E (comptées avec les répétitions éventuelles) et s le nombre d'occurrences de x dans E, de sorte que $s > n/2$. Soient y, z dans E et E' le multi-ensemble obtenu en retirant y, z de E. Le nombre de valeurs de E' est donc $n' = n - 2$. Pour déterminer le nombre s' d'occurrences de x dans E', deux cas se présentent :

- soit x n'est égal ni à y ni à z, alors $s' = s > n/2 > n'/2$
- soit x est égal à y ou à z (ou exclusif puisque $y \neq z$), alors $s' = s - 1 > n/2 - 1 = n'/2$.

Dans les deux cas, x est majoritaire dans E'.

Question 8 Je suppose x majoritaire dans $a[0:n]$ et je conserve les notations n, s précédentes. Je suppose également i entre 0 et n tel qu'il n'existe pas d'élément majoritaire dans $a[0:i]$ et je note t le nombre d'occurrences de x dans $a[i:n]$, qui comporte $n-i$ éléments. En particulier x n'est pas majoritaire parmi les i valeurs de $a[0:i]$, donc $s-t \leq i/2$, d'où $t \geq s - i/2 > n/2 - i/2$, ainsi $t > (n-i)/2$, c'est-à-dire que x est majoritaire dans $a[i:n]$.

Question 9

```
def gagnant_tour1_ter(a):
#algorithme de Boyer-Moore
    avance=0
    for c in a:
        if avance==0:
            x=c
            avance=1
        elif c==x:
            avance+=1
        else:
            avance-=1
    score=0
    for c in a:
        if c==x:
            score+=1
    if 2*score>len(a):
        return x
    else:
        return -1
```

Cet algorithme, dit de Boyer-Moore, a été publié en 1981 par Robert S. Boyer and J Strother Moore.
L'idée est d'appliquer le résultat du **8**, pour la "première passe" suggérée par l'énoncé. La variable x contiendra le seul candidat majoritaire possible.
Je maintiens l'invariant de boucle $\mathcal{P}_k$ suivant, k allant de 0 à $len(a)-1$: "après le traitement du candidat $c = a[k]$, il existe $i \in [\![0,k]\!]$ tel qu'il n'existe pas d'élément majoritaire dans $a[0:i]$ et tel que :

- soit x est majoritaire dans $a[i:k+1]$ avec `avance` voix d'avance sur le total des autres votes dans cette tranche,
- soit x est à égalité de voix avec l'ensemble des autres dans $a[i:k+1]$ (et dans ce cas `avance` vaut 0)".

Le compteur `avance` contient ainsi le nombre de voix supplémentaires obtenues par x dans la tranche $a[i:k+1]$, par rapport au total des voix obtenues par les autres candidats dans cette tranche.

$\mathcal{P}_0$ est vrai : après le traitement du premier candidat $c = a[0]$, $i = 0$ convient (!) et $x = c$ est bien majoritaire dans $a[0:1]$ avec une avance d'une voix.
Puis, en admettant $\mathcal{P}_{k-1}$, trois cas possibles après le traitement de $c = a[k]$:

- soit `avance` était nul, ce qui signifie qu'il n'y avait aucun élément majoritaire dans $a[i:k]$ (puisque x avait une voix d'avance à l'étape précédente, il était le seul élément à pouvoir rester majoritaire, or il se retrouve avec exactement la moitié des voix dans ladite tranche) ; alors k convient comme nouvelle valeur de i et je repars avec $x = c$ qui est bien majoritaire dans $a[i:k+1]$ avec une voix d'avance sur les autres (il n'y en a pas !) ;
- soit `avance` était non nul, avec $c = x$: alors je continue avec le même i et x reste majoritaire dans $a[i:k+1]$ avec une voix d'avance en plus ;
- soit `avance` était non nul, avec $c \neq x$: alors je continue avec le même i et, soit x reste majoritaire dans $a[i:k+1]$ avec une voix d'avance en moins (si `avance` reste strictement positive), soit x est à égalité de voix avec les autres dans $a[i:k+1]$ (si `avance` s'annule).

Je viens de prouver par récurrence que $\mathcal{P}_k$ est vrai pour tout k.

À la sortie de la première boucle `for`, x est le seul candidat majoritaire possible d'après la question **8**, il n'y a plus qu'à vérifier s'il l'est vraiment avec la "deuxième passe".

La complexité temporelle est clairement linéaire et la complexité spatiale constante !