Dépouillement d'élections (d'après X PSI 2005)

Partie I. Dépouillement au premier tour

Question 1.

```
def estGagnant(k, vote):
    n = len(vote)
    s = 0
    for v in vote:
        if v == k:
            s += 1
    return 2 * s > n
```

La variable `s` dénombre le nombre d'électeurs ayant voté pour le candidat `k`.

Question 2.

a) La fonction principale teste chaque votant à la recherche d'un vainqueur au premier tour (nécessairement unique s'il en existe) :

```
def gagnantTour1(vote):
    n = len(vote)
    for k in range(n):
        if estGagnant(k, vote):
            return k
```

b) La fonction `estGagnant` réalise un nombre d'opérations élémentaires proportionnel au nombre n de votants donc la complexité de cette fonction est un $O(n)$. La fonction `gagnantTour1` fait n fois appel à la fonction `est_gagnant` donc sa complexité est en $O(n^2)$.

Question 3.

```
1  def gagnantTour1Trie(vote):
2      n = len(vote)
3      c, s = vote[0], 1
4      for k in range(1, n):
5          if vote[k] == c:
6              s += 1
7          else:
8              c, s = vote[k], 1
9          if 2 * s > n:
10             return c
```

Cette fonction maintient l'invariant suivant : au moment où l'on examine le vote de l'électeur `k`, `c` désigne le candidat choisi par l'électeur `k-1` et `s` le nombre de voix que ce candidat a obtenu auprès des électeurs précédents.
La ligne 3 initialise ces invariants. Si le candidat `k` vote lui aussi pour `c`, il suffit d'incrémenter `s` (lignes 5-6) ; dans le cas contraire on ré-initialise ces invariants (lignes 7-8) puisque plus personne ne votera pour `c`, le tableau étant supposé trié. Enfin, les lignes 9-10 permettent de détecter si à un moment donné le candidat `c` est devenu majoritaire.

Question 4. La fonction `gagnantTour1Trie` est de complexité linéaire puisque le nombre d'opérations élémentaires réalisé et proportionnel à n. Mais pour pouvoir l'utiliser il faut au préalable trier le tableau, ce qui se réalise en un temps au pire proportionnel à $n\log n$. Sachant que $O(n\log n)+O(n)=O(n\log n)$ la complexité totale de cette méthode est un $O(n\log n)$, meilleure donc que la complexité obtenue à la question 2.

Partie II. Dépouillement du deuxième tour

Question 5.

```
def nombreDeVoix(k, vote):
    s = 0
    for v in vote:
        if v == k:
            s += 1
    return s
```

Question 6.

a) On définit la fonction suivante :

```
1  def gagnantsTour2(vote):
2      gagnants = []
3      vmax = 0
4      for k in range(len(vote)):
5          s = nombreDeVoix(k, vote)
6          if s > vmax:
7              gagnants = [k]
8              vmax = s
9          elif s == vmax:
10             gagnants.append(k)
11     return gagnants
```

Cette fonction utilise deux invariants : on parcourt la liste des candidats en notant `vmax` le nombre maximal de votes obtenu par un candidat déjà examiné et en notant `gagnants` la liste du ou des candidats ayant obtenu ce nombre de votes. Les lignes 7 et 8 mettent ces invariants à jour lorsqu'on découvre un candidat ayant obtenu strictement plus de votes ; la ligne 10 met à jour l'invariant `gagnants` lorsqu'on découvre un candidat ayant obtenu un nombre de votes égal à `vmax`.

b) La fonction `nombreDeVoix` réalise un nombre d'opérations élémentaires proportionnel au nombre n de votants donc la complexité de cette fonction est un $O(n)$. La fonction `gagnantsTour2` fait n fois appel à la fonction `est_gagnant` ; puisque les autres opérations sont de complexité constante sa complexité est en $O(n^2)$.

Question 7.

```
1  def gagnantsTour2Trie(vote):
2      gagnants = []
3      vmax = 0
4      c, s = vote[0], 1
5      for i in range(1, len(vote)):
6          if vote[i] == c:
7              s += 1
8              if s > vmax:
9                  vmax = s
10                 gagnants = [c]
11             elif s == vmax:
12                 gagnants.append(c)
13         else:
14             c, s = vote[i], 1
15     return gagnants
```

Dans cette question, outre les deux invariants `vmax` et `gagnants` qui sont semblables à ceux de la question précédente, on ajoute deux autres invariants : `c` qui désigne le candidat courant et `s` le nombre de voix obtenu par ce candidat parmi les votes déjà examinés.
La ligne 4 initialise ces deux invariants après le dépouillement du premier vote. On continue ensuite à dépouiller, sachant que les bulletins identiques sont consécutifs. Lorsque le bulletin suivant est identique au précédent (ligne 6) `s` est incrémenté, et les invariants `vmax` et `gagnants` éventuellement mis à jour comme dans la question précédente. Lorsque le bulletin suivant est différent du précédent (ligne 13) les invariants `c` et `s` sont ré-initialisés.

Question 8. La fonction `gagnantTour2Trie` est de complexité linéaire puisque le nombre d'opérations élémentaires réalisé et proportionnel à n. Si cette fonctions est précédée d'un tri de complexité $O(n\log n)$, la complexité totale de cette méthode est en $O(n\log n)+O(n)=O(n\log n)$.

Partie III. Dépouillement rapide du premier tour

Question 9. Notons p le nombre d'apparitions de x dans $\langle \texttt{vote}[0],\texttt{vote}[1],\dots,\texttt{vote}[k-1]\rangle$ et q son nombre d'apparitions dans $\langle \texttt{vote}[k],\texttt{vote}[k+1],\dots,\texttt{vote}[n-1]\rangle$. Par hypothèse on a $p+q>n/2$ et $p\leqslant k/2$ donc $q>(n-k)/2$, ce qui signifie que x est majoritaire dans $\langle \texttt{vote}[k],\texttt{vote}[k+1],\dots,\texttt{vote}[n-1]\rangle$.

Question 10.

a) Pour comprendre cette fonction, nous allons introduire deux suites de multi-ensembles $(F_k)_{1\leqslant k\leqslant n}$ et $(G_k)_{1\leqslant k\leqslant n}$.
Posons $F_1=\langle \texttt{vote}[0]\rangle$ et $G_1=\langle \texttt{vote}[1],\dots,\texttt{vote}[n-1]\rangle$.
Après la ligne 2, `c` est un élément majoritaire dans F_1 et $v=2p-\text{card}(F_1)$ où p est le nombre d'apparitions de `c` dans F_1.

Supposons qu'à l'étape k on dispose de l'invariant suivant : *s'il existe un élément majoritaire dans* F_k *celui-ci est égal à* `c`, *et* $v=2p-\text{card}(F_k)$ *où* p *est le nombre d'apparitions de* `c` *dans* F_k.

- Si $v=0$ l'invariant nous dit qu'il n'existe pas d'élément majoritaire dans F_k. On pose alors $F_{k+1}=\langle \texttt{vote}[k]\rangle$;
- dans le cas contraire ($v>0$) on pose $F_{k+1}=F_k\cup\langle \texttt{vote}[k]\rangle$.

Dans les deux cas on pose $G_{k+1}=\langle \texttt{vote}[k+1],\dots,\texttt{vote}[n-1]\rangle$.
Les lignes 6-9 mettent à jour `v` de sorte de maintenir à jour l'invariant $v=2p-\text{card}(F_{k+1})$ où p est le nombre d'apparitions de `c` dans F_{k+1}.
Lorsque `vote` contient un élément majoritaire, la question 9 nous permet d'affirmer qu'au cours de ce processus on a pour tout $k\in[\![1,n]\!]$:

- ou bien $c=x$;
- ou bien x est majoritaire dans G_k.

Puisque $G_n=\varnothing$ ne contient pas d'élément majoritaire, on peut donc affirmer que le résultat renvoyé par cette fonction est égal à x.

b) Cependant, lorsque `vote` ne contient pas d'élément majoritaire l'élément renvoyé par `gagnantPotentiel` est quelconque donc il n'est que le seul candidat à pouvoir être majoritaire. On définit donc la fonction suivante pour conclure :

```
def gagnantTour1Rapide(vote):
    c = gagnantPotentiel(vote)
    if estGagnant(c, vote):
        return c
```