CORRIGÉ DEVOIR SURVEILLÉ 1

1 Algorithmes de calcul de racines carrées

A- 1.

```
def suite_heron(a, n):
    x = a
    for i in range(n):
        x = 0.5*(x+a/x)
    return x
```

2. Il s'agit de la première suite du TP récursivité. Pour avoir un coût linéaire il ne faut faire qu'un seul appel récursif à la fonction `suite_heron_rec` et « ranger » donc le résultat dans une variable. Pour une fonction récursive, ne pas oublier le critère d'arrêt, ici $n = 0$.

```
def suite_heron_rec(a, n):
    if n ==0 :
        return a
    x = racine_heron_rec(a, n-1)
    return  0.5*(x+a/x)
```

3.

```
def racine_heron(a):
    return suite_heron(a, 6)
```

B- 1.

	4	5	3	6	9	2	1	3
-	4					41	× 1 =	41
		5	3					
-		4	1					
		1	2	6	9	423	× 3 =	1269
-	1	2	6	9				
				0				

On a donc $\boxed{\sqrt{45369} = 213}$.

2. Il s'agit ici de décomposer le nombre en base 100 : on peut se reporter au TP récursivité où en demandait la décomposition en binaire (base 2), c'est le même principe.

```
def decoupe(N):
    if N == 0:
        return []
    else :
        L = decoupe(N//100)
        L.append(N%100)
        return L
```

3.

```
def chiffre_suivant_racine(r, reste):
    for m in range(10):
        if (r*10+m)*m > reste :
            return m-1
```

4.

```
def racine_potence(N):
    L = decoupe(N)
    racine = 0
    D = 0
    for i in range(len(L)):
        reste = D*100 + L[i]
        m = chiffre_suivant_racine(2*racine,reste)
        D = reste - (2*racine*10+m)*m
        racine = racine*10 + m
    return racine
```

5. Puisque $\sqrt{a} = 10^{-p}\sqrt{10^{2p}a}$, pour obtenir la racine carrée avec p chiffres après la virgule, il suffit d'appliquer l'algorithme précédent à $10^{2p}a$ puis de multiplier par 10^{-p}.

```
def racine_potence_precision(N,p):
    return racine_potence(N*100**p)*10**(-p)
```

C- 1.

10|75|84			premier impair	racine
			1	
10	$-(1+3+5)$	$= \boxed{1}$	$(5+1)\times 10+1 = 61$	3
175	$-(61+63)$	$= 51$	$(63+1)\times 10+1 = 641$	32
5184	$-(641+643+645+647+649+651+653+655)$	$= 0$		328

On a donc $\boxed{\sqrt{107584} = 328}$.

2.

```
def nombre_d_impairs(N, d_imp):
    if N-d_imp < 0:
        return 0, N
    else :
        compteur , reste = nombre_d_impairs(N-d_imp, d_imp+2)
        return compteur + 1, reste
```

3.

```
def racine_goutte_a_goutte(N):
    L = decoupe(N)
    reste = 0
    d_impair = 1
    racine = 0
    for i in range(len(L)):
        n = reste*100+L[i]
        compteur, reste = nombre_d_impairs(n, d_impair)
        d_impair = (d_impair + 2*compteur-1)*10+1
        racine = racine*10 + compteur
    return racine
```

D- Et dans ma calculatrice ?
Dans votre calculatrice c'est une version de l'algorithme de Héron qui est programmée, en effet c'est celle qui nécessite le moins d'opérations : elle converge en environ 6 étapes quelque soit la taille de a, et à chaque itération on a une addition et un quotient.

2 Vote par approbation

1.

```
def a_un_doublon(L):
    n = len(L)
    for i in range(n):
        for j in range(i):
            if L[i] == L[j]:
                return True
    return False
```

Ou mieux, avec un dictionnaire :

```
def a_un_doublon(L):
    D = {}
    for x in L:
        if x in D:
            return True
        else:
            D[x] = None
    return False
```

2.

```
def vote_valide(Lvotes):
    for b in Lvotes:
        if a_un_doublon(b):
            return False
    return True
```

3. On écrit la fonction `cle_max` qui prend en argument un dictionnaire `D` et renvoie la clé de valeur maximale dans ce dictionnaire. Cette fonction a déjà été vue lors du TP systèmes de votes.

```
def cle_max(D):
    vmax = 0
    for c in D:
        if D[c] > vmax:
            vmax = D[c]
            cmax = c
    return cmax
```

Pour `vainqueur` on remplit le dictionnaire des votes, puis on utilise la fonction précédente pour déterminer le vainqueur.

```
def vainqueur(Lvotes):
    D = {}
    for b in Lvotes:
        for c in b:
            if c in D:
                D[c] += 1
            else:
                D[c] = 1
    return cle_max(D)
```

4.

```
def strategie(L, v, c):
    R = []
    v_avant_c = True
    for x in L:
        if x != v:
            R.append(x)
            if x == c:
                v_avant_c = False

        else:
            if v_avant_c:
                R.append(v)
            return R
```

5. On crée d'abord une fonction `vainqueur_et_challenger` qui à partir des listes de préférences des votants détermine le vainqueur et le challenger. Pour cela on crée le dictionnaire des votes et on détermine les deux clés de plus grandes valeurs. À nouveau, nous avons déjà rencontré cette fonction lors du TP sur des systèmes de vote.

```
def vainqueur_et_challenger(Lvotes):
    D = {}
    for b in Lvotes:
        for c in b:
            if c in D:
                D[c] += 1
            else:
```

```
                D[c] = 1
    v = cle_max(D)
    vmax = 0
    for c in D:
        if c != v and D[c] > vmax:
            vmax = D[c]
            cmax = c
    return v, cmax
```

Ensuite, à l'aide de cette fonction, on écrit une boucle qui s'arrête lorsqu'on a atteint l'équilibre.

```
def limite(Lprefs):
    Lvotes = [[L[0]] for L in Lprefs]
    v, c = vainqueur_et_challenger(Lvotes)
    Lvotes2 = [strategie(pref,v, c) for pref in Lprefs]
    while Lvotes != Lvotes2:
        Lvotes = Lvotes2
        v, c = vainqueur_et_challenger(Lvotes)
        Lvotes2 = [strategie(pref,v, c) for pref in Lprefs]
    return vainqueur(Lvotes)
```

Remarque : S'il existe un vainqueur de Condorcet (voir TP sur les ystèmes de vote) alors la limite est le vainqueur de Condorcet, sinon cela ne converge pas (ce qui est un peu embêtant car notre fonction ne termine pas dans ce cas, il faudrait donc l'améliorer).

Fin