Structure d'Union-find Corrigé

1)
```
def creerPartitionEnSingletons(n) :
    parent = [ i for i in range(n) ]
```

2)
```
def representant(parent,i) :
    j = i
    while parent[j] != j :  j = parent[j]
    return j
```

La complexité est en $O(n)$, atteinte dans le cas d'une représentation filiforme : $0 \to 1 \to 2 \to ... \to (n-1)$
Ce qui correspond au tableau `parent = [1,2,3,...,n-2,n-1,n-1]`

3)
```
def fusion(parent,i,j) :
    parent[representant(parent,i)] = representant(parent,j)
```

4) Considérons la suite suivante :

```
fusion(parent, 0, 1)
fusion(parent, 0, 2)
fusion(parent, 0, 3)
...
fusion(parent, 0, n-1)
```

Partant de la partition en n singletons, cette succession de fusions aboutit à la partition en un seul groupe. La complexité $C(n)$ de cette suite de fusions vérifie la relation de récurrence
$C(n) = C(n-1) + Kn$, donc $C(n) \sim \frac{1}{2}Kn^2$: la complexité est quadratique.

5) On crée une liste de listes selon la valeur des représentants : ainsi, dans la liste L de l'algorithme ci-dessous, $L[i]$ est la liste de représentant i si i est un représentant, et $L[i]$ est la liste vide sinon.
Puis on supprime les listes vides pour cnstruire la liste souhaitée $L2$ des groupes.

```
def partitionTableau(parent) :
    n = len(parent)
    for i in range(n) :  L[representant(parent,i)].append(i)
    L2 = []
    for liste in L :
        if liste != [] :  L2.append(liste)
    return L2
```

6)
```
def fusion(parent,rang,i,j) :
    a = parent[i] ; b = parent[j]
    if rang[a] < rang[b] :  parent[b] = a
    elif rang[a] > rang[b] :  parent[a] = b
    elif a != b :
        parent[b] = a ; rang[a] = rang[a] + 1
```

Lors d'une fusion de deux classes, la fusion des deux classes conduit à un arbre où l'un des représentant devient un fils de l'autre.

Lorsque le rang du fils est strictement inférieur à celui du père, le rang n'est pas modifié.
Mais lorsque les deux classes sont distinctes et ont le même rang, la fusion des classes conduit à un arbre dont la hauteur est augmentée de 1 par rapport à ceux des classes.

7) Supposons la propriété vraie pour les deux classes (distinctes) que l'on fusionne.
Notons k et l les rangs deux classes.
- Si les rangs k et l sont différents, le rang de l'union des classes vaut $\max(k, l)$.
Or, la réunion contient au moins $2^k + 2^l$ éléments, donc a fortiori au moins $2^{\max(k,l)}$ éléments.
- Si les rangs k et l sont égaux, le rang de l'union vaut $k + 1$.
Or, la réunion contient au moins $2^k + 2^k = 2^{k+1}$ éléments.
Donc la fonction `fusion` préserve bien l'invariant considéré.

8) L'invariant est vérifiépour la partition initiale en singletons (car $2^0 = 1$). Par 7), il est donc vérifié pour toutes les partitions obtenues.
Ainsi, on a pour toute classe de cardinal m et de rang k, on a $m \geq 2^k$, donc $k \leq \log m \leq \log n$.
Or, FIND a un coût en $O(k)$, où k est le rang de la classe considérée. D'où une complexité $O(\log n)$.

9)
```
def representant(parent,i) :
    j = i ; stock = []
    while parent[j] != j :
        stock.append(j) ; j = parent[j]
    for k in stock :  parent[k] = j
    return j
```

Même complexité que la première version (il y a seulement plus de mémoires utilisées).

10) On reprend la même fonction que celle proposée à la quesiton 5).
Pour justifier la complexité linéaire, il suffit de justifier la complexité linéaire de l'instruction :
`for i in range(n) :  L[representant(parent,i)].append(i)`

Chaque arête du graphe associée à la forêt d'arbres n'est considérée qu'une seule fois (puisque'ensuite remplacée par une arête pointant directement d'un sommet vers la racine) et il y a $O(n)$ arêtes (dans un arbre à r sommets, il y a $r-1$ arêtes). D'autre part, d'autres arêtes sont ajoutées qui pointent d'un sommet directement vers son représentant. Il y a ici aussi $O(n)$. Donc au total, le nombre d'appels à `parent` est en $O(n)$ lors de l'éxecution de l'instruction.

11)
```
def cycle(parent,a) :
    return representant(parent,a[0]) == representant(parent,a[1])
```

12)
```
def foretCouvrante(n,A) :
    B = [] ; parent = creerPartitionEnSingletons(n)
    for a in A :
        if not cycle(parent,a)
            B.append(a)
            fusion(parent, a[0], a[1])
    return B
```

Le coût en est en $O(m \log n)$ pour les tests `cycle(parent,a)` et $O(n)$ pour les fusions et la construction de B (qui contient au plus $(n-1)$ arêtes. D'où une complexité en $O(n + m \log n)$.