Corrigé du QCM n°7

1. La programmation récursive est :

1. un mal nécessaire
2. articulée autour d'un ou plusieurs cas de base et d'un ou plusieurs appels récursifs
3. à privilégier par rapport à la programmation itérative
4. une approche adaptée à la stratégie diviser pour régner

La programmation récursive permet d'obtenir d'excellentes complexités sur des situations où la stratégie « diviser pour régner » s'applique : FFT (Fast Fourier Transform), tris, ... Un programme récursif contient un ou plusieurs cas de bases et un ou plusieurs appels récursifs (récursions). La programmation récursive ne permet pas de faire toujours mieux que la programmation itérative : l'empilement des récursions induit une complexité spatiale et impose des précautions relativement à la hauteur de la pile de récursion.

2. Identifier des algorithmes dont l'écriture récursive est plus simple que l'écriture itérative :

1. calcul d'un coefficient binomial
2. exponentiation rapide
3. recherche de doublons dans une liste
4. recherche d'un élément dans une liste

L'écriture récursive de l'exponentiation rapide ne requiert pas de connaître l'écriture binaire d'un nombre. Le calcul d'un coefficient binomial récursif permet de coder la relation $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$ sans variable à initialiser et sans boucle `for`.

3. On considère la fonction `fibo` suivante :

```
def fibo(n):
    if n<=1:
        return 1
    else:
        return fibo(n-1)+fibo(n-2)
```

Sa complexité temporelle est :

1. exponentielle
2. logarithmique
3. linéaire
4. quasi-logarithmique

La complexité temporelle de `fibo(n)` est en $O(\varphi^n)$ avec $\varphi = \dfrac{1+\sqrt{5}}{2}$ ce qui en fait une complexité exponentielle.

4. L'algorithme de la FFT est en :

1. O(N log(N))
2. O(log(N))
3. O(N^2)
4. O(N)

Avec $N = 2^p$, on a $$\frac{T(2^p)}{2^p} - \frac{T(2^{p-1})}{2^{p-1}} = O(1)$$

Après téléscopage, on en déduit $T(2^p) = O(p2^p)$ d'où $T(N) = O(N \log(N))$.

5. Le tri par insertion a une complexité temporelle :

1. dans le pire des cas en $O(n^2)$
2. en $O(n^2)$
3. en $O(n \log(n))$
4. dans le meilleur des cas en $O(n \log(n))$

Le tri par insertion est en $O(n)$ dans le meilleur des cas et en $O(n^2)$ dans le pire des cas.

6. Identifier le(s) tri(s) utilisant un pivot pour placer les éléments :

1. le tri rapide pas en place
2. le tri rapide en place
3. le tri par insertion
4. le tri fusion

Les algorithmes de tri rapide en place et pas en place utilisent un pivot pour la partition de la liste.

7. Hormis le cas d'une liste déjà triée, quelle est la complexité optimale pour un tri :

1. $O(n^2)$
2. $O(n)$
3. $O(\log(n))$
4. $O(n \log(n))$

La complexité optimale pour un tri sur une liste quelconque est en $O(n \log(n))$.

8. Le tri rapide a une complexité temporelle :

1. en $O(n \log(n))$
2. en $O(n^2)$
3. dans le pire des cas en $O(n^2)$
4. dans le meilleur des cas en $O(n \log(n))$

Le tri rapide a une complexité temporelle en $O(n \log(n))$ dans le meilleur des cas et en $O(n^2)$ dans le pire des cas.

9. Le tri fusion a une complexité temporelle :

1. en $O(n \log(n))$
2. en $O(n^2)$
3. dans le pire des cas en $O(n^2)$
4. dans le meilleur des cas en $O(n \log(n))$

Le tri fusion a une complexité temporelle en $O(n \log(n))$ dans tous les cas.

10. Pour éviter le pire cas lors d'un tri rapide, on peut :

1. utiliser une version en place
2. choisir un pivot au hasard
3. choisir un pivot à la fin
4. utiliser une version pas en place

Si on choisit un pivot en début ou en fin de liste, le cas d'une liste triée par ordre croissant ou décroissant produit la situation la pire pour le tri rapide. Avec un choix de pivot au hasard, on évite ce pire cas. Le caractère en place ou pas en place n'a pas d'incidence sur ce phénomène.