Corrigé : disque dur à deux têtes (X MP-PC 2006)

Partie I. Coût d'une séquence de déplacements

Question 1. J'ai choisi de maintenir une liste `pos` de longueur 2 de sorte que `pos[0]` et `pos[1]` représentent les positions respectives des deux têtes de lecture.

```
def cout(R, D):
    c = 0
    pos = [0, 0]
    for i in range(len(R)):
        c += abs(R[i] - pos[D[i]])
        pos[D[i]] = R[i]
    return c
```

Question 2. Chaque déplacement peut être réalisé par l'une ou l'autre des deux têtes, ce qui donne 2^n séquences de déplacements possible pour un bloc de requêtes de longueur n. Une recherche exhaustive d'une séquence de coût optimal aurait donc une complexité temporelle exponentielle.

Question 3. Les deux têtes de lecture étant interchangeables, il existe toujours une séquence de coût optimal commençant par 0.

Partie II. Coût optimal pour deux requêtes

Question 4. Le bloc de requêtes $[10,3]$ est traité pour un coût minimal de 13 par la séquence $[0,1]$.
Le bloc de requêtes $[3,10]$ est traité pour un coût minimal de 10 par la séquence $[0,0]$.

Question 5. Il suffit de choisir la tête la plus proche de r_1 : ce sera la première si $|r_0 - r_1| < r_1$ et la seconde dans le cas contraire. (En cas d'égalité, l'une ou l'autre des deux têtes convient).

Question 6. On met en œuvre la démarche décrite à la question précédente :

```
def optimal2(R):
    if abs(R[1] - R[0]) < R[1]:
        return [0, 0]
    else:
        return [0, 1]
```

Partie III. Coût optimal pour trois requêtes

Question 7. On suggère ici une stratégie gloutonne : pour chaque requête on déplace la tête de lecture la plus proche. Dans le cas de la succession de requêtes $[20,9,1]$ cela consiste à :

1. déplacer la première tête de lecture à la position 20 ;
2. déplacer la seconde tête de lecture à la position 9 ;
3. déplacer la seconde tête de lecture à la position 1 ;

soit la séquence de déplacements $[0,1,1]$ pour un coût total égal à $20+9+8=37$.

Question 8. La séquence de déplacements $[0,0,1]$ a un coût total égal à $20+11+1=32$ donc la stratégie gloutonne ne donne pas toujours la solution optimale.

Question 9. Il y a quatre séquences de déplacements possibles : $[0,0,0]$, $[0,0,1]$, $[0,1,0]$ et $[0,1,1]$. On se contente de les comparer entre elles :

```
def optimal3(R):
    opt = float('inf')
    for i in (0, 1):
        for j in (0, 1):
            c = cout(R, [0, i, j])
            if c < opt:
                opt, D = c, [0, i, j]
    return D
```

Partie IV. Coût optimal pour n requêtes

La partie précédente a montré que la stratégie gloutonne n'est pas optimale, nous allons donc mettre en place une stratégie dynamique.

Question 10. Lorsque $i < k-1$, la configuration (i,k) ne peut être atteinte qu'à partir de la configuration $(i,k-1)$ (c'est la tête ayant réalisé la requête r_{k-1} qui bouge de nouveau) pour un coût égal à $\text{coût}_{k-1}[i] + |r_k - r_{k-1}|$.

Question 11. Lorsque $i = k-1$, la configuration $(k-1,k)$ peut être atteinte à partir d'une configuration de type $(k-1,j)$ avec $j < k-1$ (c'est la tête n'ayant pas réalisé la requête r_{k-1} qui réalise la requête r_k) pour un coût égal à $\text{coût}_{k-1}[j] + |r_k - r_j|$. On a donc $\text{coût}_k[k-1] = \min\limits_{0 \leqslant j < k-1} \left(\text{coût}_{k-1}[j] + |r_k - r_j|\right)$.

Question 12. Les formules précédentes permettent d'obtenir les listes suivantes :

coût_4 :	18	17	17	13	-1	-1
coût_5 :	22	21	21	17	14	-1
coût_6 :	25	24	24	20	17	15

Question 13.

```
def suivant(C, R, k):
    CC = [-1 for _ in range(len(C))]
    for i in range(k - 1):                    # question 10
        CC[i] = C[i] + abs(R[k] - R[k - 1])   #
    CC[k - 1] = C[0] + abs(R[k] - R[0])                      #
    for j in range(1, k - 1):                                # question 11
        CC[k - 1] = min(CC[k - 1], C[j] + abs(R[k] - R[j]))  #
    return CC
```

Question 14.

```
def coutsOpt(R):
    couts = [[0] + [-1] * (len(R) - 2)]
    for k in range(1, len(R)):
        couts.append(suivant(couts[k - 1], R, k))
    return couts
```

Question 15. La fonction `suivant` a une complexité temporelle en $O(k)$. Sachant que $\sum\limits_{k=1}^{n} k = O(n^2)$, on en déduit que la fonction `coutOpt` a une complexité en $O(n^2)$.

Question 16. Les configurations finales sont de type (i,n) avec $i < n$ donc le coût optimal permettant de satisfaire toutes les requêtes est égal à $\min\limits_{0 \leqslant i < n} \text{coût}_n[i]$. Ceci conduit à la fonction :

```
def coutOptimal(R):
    couts = coutsOpt(R)
    return min(couts[-1])
```