correction du DS2

(d'après X PSI/PT 2013)

Partie I

1. Comme on doit aussi trouver les indices des points d'altitude minimale et maximale, il vaut mieux ne pas utiliser les fonctions `min` et `max`

```
def Fenetre(H) :
    n = len(H)-1 # pour respecter la notation du texte
    hMin,hMax,iMin,iMax = H[0],H[0],0,0
    for i in range(1,n+1) :
        if H[i] < hMin :
            hMin = H[i]
            iMin = i - 1
        if H[i] > hMax :
            hMax = H[i]
            iMax = i - 1
    return hMin,hMax,iMin,iMax
```

2. On utilise la distance euclidienne et on tient compte du fait qu'on peut avoir $i > j$

```
def distanceAuSol(H,i,j) :
    if i > j :
        i,j = j,i
    S = 0
    for k in range(i,j) :
        S += sqrt(1+(H[k+1]-H[k])**2)
    return S
```

3. Ne pas oublier de rajouter P_0 et P_n

```
def Remarquables(H) :
    n = len(H)-1
    P=[0]
    for i in range(1,n) :
        if H[i] > H[i-1] and H[i] > H[i+1] :
            P.append(i)
    P.append(n)
    return P
```

4. On utilise les fonctions précédentes

```
def PlusLongBassin(H) :
    P = Remarquables(H)
    L = [distanceAuSol(H,P[i],P[i+1]) for i in range(len(P)-1)]
    return max(L)
```

Partie II

1. Comme précisé, soit on distingue les cas $i > j$ et $i < j$, soit on se ramène toujours à l'un des deux

```
def Legal(H,i,j,l,delta) :
    if j < i :
        j,i = i,j
    legislation = True
    beta = float((H[j] - H[i]) / (j - i))
    for k in range(i+1,j) :
        alpha = float((H[k] + Delta - H[i] - l) / (k - i))
        if beta <= alpha :
            legislation = False
    return legislation
```

2. On part d'un indice i (où on vient de planter un poteau), initialement 0, et on cherche l'indice j du premier point violant la législation ; le poteau suivant est alors à placer en $j-1$

```
def GloutonEnAvant(H,l,delta) :
    n = len(H) - 1
    i = 0
    poteaux = [0]
    while i < n :
        j = i+1
        suivant = False
        while not suivant and j <= n :
            if not Legal(H,i,j,l,delta) :
                suivant = True
            else :
                j += 1
        if suivant : # un point non légal a été trouvé
            i = j-1
        else : # boucle interrompue par j=n donc on plante en Pn
            i = n
        poteaux.append(i)
    return poteaux
```

3. La complexité de `Legal(H,i,j,l,delta)` est en $O(|j-i|)$ donc celle de `GloutonEnAvant(H,l,delta)` est en $O\left(\sum_{j=0}^{n-1} j\right) = O(n^2)$ car les deux boucles `while` imbriquées ne font en fait qu'un seul parcours de la liste (la boucle sur `i` repart depuis l'entier `j-1`).
La complexité quadratique vient des appels successifs à la fonction `Legal(H,i,j,l,delta)` qui calcule en fait de nombreuses fois la même chose. Pour améliorer la complexité, il suffirait de stocker au fur et à mesure les résultats de ces appels dans une liste (ou un dictionnaire) pour ne pas les recalculer à chaque fois.

4. C'est la même chose mais la recherche part de la droite du paysage

```
def GloutonAuPlusLoin(H,l,delta) :
    n = len(H) - 1
    i,j = 0,n
    poteaux = [0]
    while i < j :
        while not Legal(H,i,j,l,delta) :
            j -= 1
        i = j
        j = n
        poteaux.append(i)
    return poteaux
```

Partie III

1. On doit choisir de placer ou non un poteau aux points P_i avec $i \in [\![1, n-1]\!]$ donc il y a 2^{n-1} placements.
Pour chacun des 2^{n-1} placements, on doit en vérifier la légalité ce qui nécessite de calculer toutes les pentes possibles donc un coût en $O(n^2)$. Si on mémorise ces pentes une seule fois alors ce calcul sera négligeable devant le reste de l'algorithme.
Reste ensuite à calculer la longuer du fil de chaque configuration (cout $O(n)$) puis en déterminer le minimum (coût $O(2^{n-1})$).
Le coût global est donc $O(n2^{n-1}) = O(n2^n)$.

2. La longueur d'un fil allant de P_0 à P_0 est bien nulle.
Pour un fil allant de P_0 à P_i de longueur minimale deux situations sont possibles :
— soit on peut aller directement de P_0 à P_i et dans ce cas $L_i = P_0P_i = P_0P_i + L_0$ (on peut calculer la distance au sol puisque le fil est parallèle au sol
— soit on doit passer par au moins un point intermédiaire ; si on note P_j le dernier point avant d'arriver en P_i alors on peut tirer un fil de P_j à P_i et la longueur minimale total d'un tel fil est bien $L_j + P_jP_i$.
La longueur minimale d'in fil de P_0 à àP_i est ensuite la longueur minimale de tous ces cas possibles.

3. On utilise une liste `Long` de taille $n+1$ pour mémoriser les longueurs des fils (une fonction récursive avec mémoïsation est possible mais le calcul de sa complexité serait plus difficile)

On commence par une fonction calculant la longueur P_iP_j (pas au sol)

```
def distance(H,i,j) :
    return sqrt((j - i)**2 + (H[j] - H[i])**2)
```

puis la fonction demandée

```
def Minimale(H,l,delta) :
    Long = [0]
    n = len(H) - 1
    for i in range(1,n+1) :
        L = []
        for j in range(i) :
            if Legal(H,j,i,l,delta) : # on vérifie si on peut tirer un
                fil
                L.append(distance(H,j,i)+Long[j])
        Long[i] = min(L)
    return L[n]
```

Pour i fixé, le remplissage de `L` est en $O(i^2)$ (à cause des utilisations de `Legal`, la détermination du minimum qui suit en $O(i)$. On fait cela n fois donc la complexité est en $O(n^3)$.

4. On s'inspire de ce que l'on a vu dans l'algorithme de Floyd-Warshall (mais avec une seule dimension ici). Il faut modifier la fonction précédente et ne pas utiliser la fonction `min` car c'est surtout le numéro du point où le minimum est atteint qui nous intéresse ; on introduit donc tous les indices précédents dans `L` avec une valeur absurde lorsqu'un fil ne peut pas joindre P_j et P_i (`float('inf')` puisqu'on recherchera un minimum)

```
def PosMinimale(H,l,delta) :
    n = len(H) - 1
    Long = [0]
    Opt = [-1] # pas de poteaux avant P0 donc une valeur absurde
    for i in range(1,n+1) :
        L = []
        for j in range(i) :
            if Legal(H,j,i,l,delta) : # on vérifie si on peut tirer un
                fil
                L.append(distance(H,j,i)+Long[j])
            else :
                L.append(float('inf')) # la valeur absurde
        # puis on recherche l'indice de poteaux précédent
        indPrec = 0
        for k in range(len(L)) :
            if L[k]<L[indPrec] :
                indPrec = k
        Opt.append(indPrec)
        Long.append(L[indPrec])
    # puis on reconstruit le placement des poteaux en partant de la fin
    poteaux = [n]
    k = n
    while k != 0 :
        k = Opt[k]
        poteaux.append(k)
    return poteaux[::-1] # et on remet dans le bon ordre
```

Le remplissage de cette deuxième liste ne modifie pas la complexité, pas plus que la reconstruction de la liste `poteaux` dont le coût est en $O(n)$ donc négligeable devant n^3. À nouveau, la complexité est en $O(n^3)$.