Spéléo-logique

Partie I : Validité d'un profil

Question 1. Un profil contenant au moins un `D`, la liste argument est nécessairement non vide. On commence par créer une fonction qui renvoie le pas opposé et on applique la définition.

```
def pas_oppose(p):
    if p == H: return B
    elif p == B: return H
    elif p == G: return D
    else: return G

def est_sans_rebroussement(g):
    n=len(g)
    if (g[0] == H) or (g[n-1] == B): return False
    for i in range(n-1):
        if g[i] ==  pas_oppose(g[i+1]): return False
    return True
```

Question 2. Je pars du principe que la descente ou la montée peuvent être vide.

On parcourt la partie descente du profil pour trouver un éventuel `H` en s'assurant ne pas tomber sur `G` puis lors de la remontée, on s'assure qu'il n'y a ni `G` ni `B`. On évite les rebroussements au début, à la fin et au fond de la vallée.

```
def est_une_vallee(g):
    n=len(g)
    i=0
    while i < n and g[i] != H: # parcours de l'éventuelle descente
        if g[i] == G: return False
        i += 1
    if i==0: return False #on ne doit pas commencer par un H (descente vide)
    elif g[i-1] == B: return False #rebroussement détecté
#si i=n (montée vide),  on s'assure de ne pas terminer par un B
    else:
        while i < n: # parcours de l'éventuelle montée
            if g[i] == G or g[i] == B: return False
            i += 1
        return True
```

Question 3. On fait attention à l'orientation de l'axe des ordonnées.

```
def voisin(x,y,d):
    if d == H: return (x,y-1)
    elif d== B: return (x,y+1)
    elif d== D: return (x+1,y)
    else: return (x-1,y)
```

Question 4. On part du point $(0,0)$ et on parcourt le profil pour chercher le voisin.

```
def liste_des_points(g):
    (x,y) = (0,0)
    res=[(x,y)]
    for d in g:
        (x,y) = voisin(x,y,d)
        res.append((x,y))
    return res
```

Question 5. On cherche les doublons dans la liste des points de `g`.

```
def est_simple(g):
    Points = liste_des_points(g)
    n = len(Points)
    for i in range(n-1):
        for j in range(i+1,n):
            if Points[i] == Points[j]: return False
    return True
```

Dans ce qui précède une étape de boucle intérieure se fait en temps constant. Le nombre de fois où cette étape est effectuée est :

$$\sum_{i=0}^{n-2}(n-i) = \frac{(n-1)(n+2)}{2} \underset{n\to+\infty}{\sim} \frac{n^2}{2} = \mathcal{O}(n^2)$$

La complexité de la fonction `est_simple(g)` est quadratique

Partie II : Vallée

Question 6. On parcourt la liste jusqu'au premier `H` (s'il existe).

On sait que le profil ne contient pas de `G` mais qu'il contient nécessairement un `D` et ne termine pas par un `B`. Lors de ce parcours, on sait que si l'on rencontre un `B`, on rencontrera nécessairement un `D` car la séquence `B,H` est interdite (pas de rebroussement).

La sous-suite de `D` juste avant le début de l'éventuelle montée ou à la fin de la liste, correspond au fond la vallée. Le début de cette suite correspond au point que doit renvoyer la fonction. Au début de chaque suite de `D`, le booléen (drapeau) `suite_D` signale que l'on parcourt une sous-suite de `D` et l'indice `c` en repère le début.

```
def fond(v):
    Points = liste_des_points(v)
    i = 0 # indice de parcours
    c = 0 # indice candidat de début de fondcde vallée
    n = len(v)
    suite_D = False # drapeau signalant une sous- suite de D
    while i <n and v[i] != H:# tant que l'on descend
        if v[i] == D and not suite_D:
            suite_D = True
            c = i # début d'une sous suite de D
        elif v[i] == B: # fin d'une sous suite de D
            suite_D = False
        i += 1
    return Points[c]
```

Question 7. On repère les plateaux comme correspondant à des sous-listes maximales de `v` composées uniquement de `D`. Un plateau termine soit par une direction autre que `D` soit à la fin de `v`.

La complexité de `liste_des_points(v)` est linéaire et chaque tour de boucle s'effectue en temps constant ainsi que les opérations hors de la boucle. La complexité de `plateaux(v)` ci-dessous est bien linéaire.

```
def plateaux(v):
    Points = liste_des_points(v)
    suite_D = False # = on est sur un plateau
    res = []
    n = len(v)
    for i in range(n):
        d = v[i]
        if d == D and not suite_D: #début d'un plateau
            suite_D = True
            (x0,y) = Points[i]
        elif d != D and suite_D: #fin d'un plateau
            (x1, _) = Points[i]
            suite_D = False
            res.append( (x0, x1, y) )
    if suite_D: #si v termine par D
        (x1, _) = Points[n]
        res.append( (x0, x1, y) )
    return res
```

Question 8. Dans cette fonction, on commence par rechercher l'indice du plateau le plus bas qui donne le côté d'ordonnée maximale du rectangle. Puis on remonte en parcourant la liste obtenue par `plateaux(v)`, à droite et à gauche via les indices `id` et `ig`.

Ensuite, on maintient trois variables entières `xmin, xmax, y` qui caractérisent le côté du rectangle courant par les abscisses de ses sommets et par leur ordonnée commune. Le largeur d'un rectangle s'obtient en effectuant la différence des deux abscisses du côté, et la hauteur par la différence de l'ordonnée du rectangle et celle du suivant s'il existe et vaut `-1` sinon.

```
def decomposition_en_rectangles(v):
    p = plateaux(v)
    i = 0
    listerec = []
    n = len(p)
    while i < len(p)-1 and p[i][2]<p[i+1][2]:
        i += 1# recherche du plateau du fond
    ig = i
    id = i
    xmin, xmax, y = p[i][0], p[i][1], p[i][2]
    while ig > 0 and id < len(p)-1:
        if p[ig-1][2] > p[id+1][2]:#on compare les ordonnées
            listerec.append([xmax-xmin, y-p[ig-1][2]])
            ig -= 1
            xmin, y = p[ig][0], p[ig][2]
        elif p[ig-1][2] < p[id+1][2]:
            listerec.append( (xmax-xmin,y-p[id+1][2]) )
            id += 1
            xmax, y = p[id][1], p[id][2]
        else:
            listerec.append( (xmax-xmin, y-p[id+1][2]) )
            id += 1
            ig -= 1
            xmin, xmax, y = p[ig][0], p[id][1], p[ig][2]
    while id < len(p)-1:
            listerec.append( (xmax-xmin,y-p[id+1][2]) )
            id+=1
            xmax, y= p[id][1], p[id][2]
    while ig > 0:
            listerec.append( (xmax-xmin,y-p[ig-1][2]) )
            ig-=1
            xmin, y= p[ig][0], p[id][2]
    listerec.append( (xmax-xmin,-1) )
    return listerec
```

La première boucle compte moins de tours que la longueur de la liste des plateaux : `p` qui est elle même de longueur inférieure à celle de `v`.

Cette fonction est bien linéaire en fonction de la taille de la liste argument.

Question 9. On commence par construire la liste des rectangles et on cumule les aires en restant inférieur à la valeur de `t`.

```
def hauteur_de_l_eau(t,v):
    Rectangles= decomposition_en_rectangles(v)
    n =  len(Rectangles)
    s = 0.0 # surface cumulée
    res = 0 # hauteur cumulée
    i = 0
    (w, h) = Rectangles[i]
    while i < n -1 and (s + w * h ) <= t:
        s += w * h
        res += h
        i += 1
        (w, h) = Rectangles[i]
    res += (t - s) / w
    return res
```

Partie III : Grottes à ciel ouvert

Question 10. On suit la consigne. La complexité est clairement linéaire.

```
def sommet(p):# je n'ose pas p[-1]
    res = p.pop()# la liste est non vide
    p.append(res)
    return res
```

```
def hierarchie_rectangles(g):
    [pile, origine, largeur, parent, enfants] = [[] for _ in range(5)]
    (x, y, icr) = (0, 0, -1) #icr pour indice courant du rectangle
    for d in g:
        if d == D: x += 1
        elif d == B:
            icr += 1
            y += 1
            origine.append((x, y))
            largeur.append(-1)
            enfants.append([])
            if pile == []:
                parent.append(-1)
                pile.append(icr)
            else:
                pere = sommet(pile)
                parent.append(pere)
                enfants[pere].append(icr)
                pile.append(icr)
        else:# alors d = H
            y -= 1
            nrec = pile.pop() # numéro de rectangle à fermer
            (xo, yo) = origine[nrec]
            largeur[nrec] = x - xo
    return origine, largeur, parent, enfants
```

Question 11. On utilise un parcours en profondeur du graphe dont la représentation correspond à la géométrie de la grotte. Je ne me sers pas de la liste `parents`.

J'utilise le fait que chaque rectangle (à part le numéro 0) est l'enfant d'un seul rectangle et que dans chaque liste d'enfants les rectangles sont rangés de gauche à droite.

Ainsi la fonction `traiter(s)` n'est appelée qu'une fois par rectangle et dans le bon ordre.

La complexité est donc linéaire en le nombre de rectangles donc linéaire en la longueur de la liste encodant le profil.

```
def ordre_remplissage_depuis_origine(parent, enfants):
    res = []
    def traiter(r):
        for e in enfants[r]:
            traiter(e)
        res.append(r)# les descendants sont tous dans la liste
    traiter(0)
    return res
```

Il s'agit du parcours de l'arbre en profondeur. La pile utilisée est celle de la récursivité.

Question 12. On reprend l'idée de la question 9. La complexité est toujours linéaire en fonction du nombre de rectangles donc en la longueur de la liste encodant le profil.

```
def hauteurs_eau_depuis_origine(t, largeur, parent, enfants):
    n = len(parent)
    hauteur = [0.0 for _ in range(n)]
    Ordre = ordre_remplissage_depuis_origine(parent, enfants)
    s = 0.0 # surface cumulée
    i = 0
    rect = Ordre[i]
    w = largeur[rect]
    while i < n-1 and (s + w) <=t:
        s += w
        hauteur[rect] = 1.0
        i += 1
        rect = Ordre[i]
        w = largeur[rect]
    hauteur[rect] = (t-s)/w
    return hauteur
```

Question 13. Au début, si la source est située sur un rectangle d'ordonnée localement maximale, seul ce rectangle se remplit. Sinon au plus deux rectangles, peuvent se remplir.

Si un rectangle est seul à être rempli, les rectangles suivants peuvent être soit le rectangle père soit deux rectangles situés à droite et à gauche.

Lorsque deux rectangles viennent d'être simultanément remplis, le rectangle suivant à être rempli est soit le rectangle père soit un rectangle à droite s'il est situé à droite des rectangles déjà remplis, soit un rectangle à gauche s'il est situé à gauche. Le rectangle père peut être commun.

Ainsi par récurrence, au plus deux rectangles peuvent être remplis simultanément.

Question 14. On utilise la même idée pour le parcours de graphe. Là encore la complexité est linéaire en le nombre de rectangle.

```
def volume_totaux(largeur, parent, enfants):
    volume = largeur.copy()
    def traiter(s):
        for e in enfants[s]:
            traiter(e)
            volume[s] += volume[e]
    traiter(0)
    return volume
```

Question 15. Il faut dans un premier temps créer l'ordre de remplissage des rectangles un peu comme pour la fonction 11 sauf que dans la liste retour il faudra des couples de rectangles, `(r1, r2)` avec comme convention si seul le rectangle `r1` est rempli, alors `r2` vaut `-1`. La complexité sera certainement linéaire.

Ensuite on procédera comme à la question 12, avec une complexité qui sera encore linéaire.