ECOLE POLYTECHNIQUE - ESPCI
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2022

JEUDI 28 AVRIL 2022
16h30 - 18h30

FILIERES MP-PC-PSI
Epreuve n° 8

INFORMATIQUE B (XELSR)

Durée : 2 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Spéléo-logique

Une phrase courte et claire prend moins de temps à écrire que des pensées confuses...
Utilisez du brouillon !

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve. Le langage de programmation sera **obligatoirement** `Python`.

Complexité. La complexité, ou le temps d'exécution, d'une fonction P est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc...) nécessaires à l'exécution de P. Lorsque cette complexité dépend de plusieurs paramètres n et m, on dira que P a une complexité en $\mathcal{O}(\phi(n,m))$ lorsqu'il existe trois constantes A, n_0 et m_0 telles que la complexité de P est inférieure ou égale à $A \cdot \phi(n,m)$, pour tout $n \geqslant n_0$ et $m \geqslant m_0$. Une fonction prenant une liste en argument sera dite de *complexité linéaire* si elle est de complexité $\mathcal{O}(n)$ où n désigne la longueur de la liste passée en argument.

Lorsqu'il est demandé de donner la complexité d'un programme, vous devrez justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Rappels concernant le langage `Python`. *L'utilisation de toute fonction* `Python` *sur les listes autre que celles mentionnées dans ce paragraphe est interdite.*

Si `a` désigne une liste en `Python` de longueur n :

- `len(a)` renvoie la longueur de cette liste, c'est-à-dire le nombre d'éléments qu'elle contient ; la complexité de `len` est en $\mathcal{O}(1)$.
- `a[i]` désigne le `i`-ème élément de la liste, où l'indice `i` est compris entre 0 et `len(a) - 1` ; la complexité de cette opération est en $\mathcal{O}(1)$.
- `a.append(e)` ajoute l'élément `e` à la fin de la liste `a` ; la complexité de cette opération est en $\mathcal{O}(1)$.
- `a.pop()` renvoie la valeur du dernier élément de la liste `a` et l'élimine de la liste `a` ; la complexité de cette opération est en $\mathcal{O}(1)$.
- `a.copy()` fait une copie de la liste `a` ; la complexité de cette opération est en $\mathcal{O}(n)$.

- Si `f` est une fonction (que l'on supposera de complexité $\mathcal{O}(1)$), la syntaxe `[f(x) for x in a]` permet de créer une nouvelle liste, similaire à la liste `b` résultant de l'exécution du code suivant :

```
b = []
for x in a:
    b.append(f(x))
```

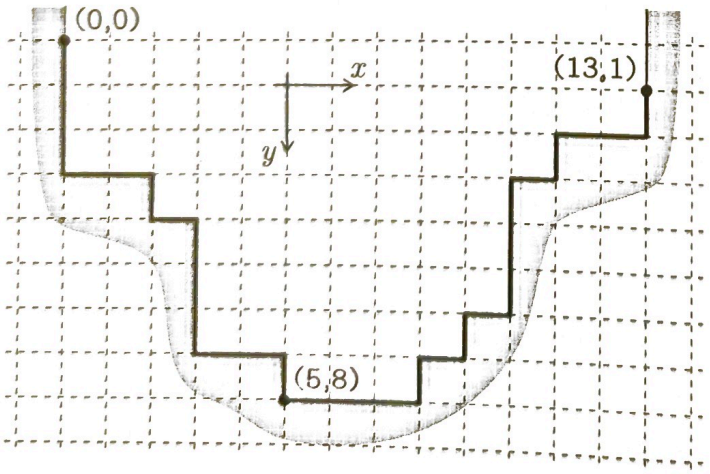
La complexité de cette création de liste est en $\mathcal{O}(n)$.

L'usage des structures de données d'ensemble `set` ou de dictionnaire `dict` **n'est pas autorisé.**

On fera attention à éviter les effets de bord : sauf lorsque cela est explicitement demandé dans l'énoncé de la question, les fonctions proposées ne devront pas modifier les paramètres qui lui sont passés en argument.

Aucune justification d'un algorithme ou de sa complexité ne devrait excéder 10 lignes.

Le problème. Nous allons déterminer le remplissage d'une grotte lors d'une inondation alimentée par une source d'eau localisée quelque part dans la grotte. La grotte considérée sera bi-dimensionnelle et décrite par le profil de son fond. On supposera définies quatre constantes globales `H = "H"`, `B = "B"`, `G = "G"` et `D = "D"`, représentant les quatre directions verticales (Haut et Bas) et horizontales (Gauche et Droite). Le profil de la grotte sera donné sous la forme d'une suite de pas horizontaux ou verticaux de longueur 1, encodée sous la forme d'une liste composée des constantes `H`, `B`, `G` et `D`, pour les quatre directions Haut, Bas, Gauche et Droite. L'origine du profil sera toujours le point `(0,0)`. On considérera toujours que le profil de la grotte se prolonge à gauche et à droite par deux murs verticaux infinis ($\texttt{B}^\infty$ et $\texttt{H}^\infty$). Dans tout le sujet, on supposera également que le profil contient toujours au moins un pas `D` vers la Droite. La figure 1 donne l'exemple d'une grotte et de son encodage par une liste :




```
[ B,B,B,D,D,B,D,B,B,B,D,D,B,D,
  D,D,H,D,H,D,H,H,H,D,H,D,D,H ]
```

FIGURE 1 – Une grotte et son profil.

Partie I : Validité d'un profil

On dira qu'un profil est *sans rebroussement* s'il ne contient pas de pas qui revienne immédiatement sur le pas précédent, par exemple pas de `...,G,D,...`. Étant donné les conditions aux bords, le profil d'une grotte sans rebroussement ne commence pas par `H` et ne finit pas par `B`.

Question 1. Écrire une fonction `est_sans_rebroussement(g)` qui prend en argument la liste `g` décrivant un profil et renvoie `True` si et seulement si le profil est sans rebroussement, et `False` sinon.

Une *vallée* est une grotte dont le profil est sans rebroussement et commence par descendre en ne faisant que des pas vers le Bas ou la Droite, puis remonte en ne faisant que des pas vers le Haut ou la Droite jusqu'à son point d'arrivée (la direction Gauche est en particulier interdite). La grotte de la figure 1 est une vallée.

Question 2. Écrire une fonction `est_une_vallee(g)` qui prend en argument la liste `g` décrivant un profil et renvoie `True` si et seulement si le profil est une vallée, et `False` sinon.

On considère désormais que les axes des x et des y pointent respectivement vers la Droite et le Bas. On rappelle que le profil d'une grotte a pour origine la position `(0,0)`.

Question 3. Écrire une fonction `voisin(x,y,d)` qui prend en arguments deux entiers `x`, `y` et une direction `d` $\in \{$`H,B,G,D`$\}$, et renvoie le couple de coordonnées du voisin du point `(x,y)` dans la direction `d`.

Question 4. Écrire une fonction `liste_des_points(g)` qui prend en argument la liste `g` décrivant un profil et renvoie la liste des coordonnées `[`$(x_0,y_0),\ldots,(x_n,y_n)$`]` des points de l'origine à l'arrivée du profil.

On dira qu'un profil est *simple* s'il ne repasse pas par le même point.

Question 5. Écrire une fonction `est_simple(g)` qui prend en argument la liste `g` décrivant un profil et renvoie `True` si et seulement si le profil est simple. Expliciter sa complexité.

Partie II : Vallée

Dans cette partie, nous considérerons que les profils sont toujours de type "vallée".

Le *fond* d'une vallée est son point le plus à gauche parmi ses points les plus bas. Le fond de la vallée de la figure 2 page suivante a pour coordonnées `(5,8)`.

Question 6. Écrire une fonction `fond(v)` qui renvoie les coordonnées (x,y) du fond de la vallée encodée par la liste des directions `v`.

On considère à présent qu'au temps $t = 0$, une source d'eau *située au fond de la vallée*, commence à couler avec un *débit constant* et à remplir la vallée. L'objectif de cette partie est de calculer quelle sera la hauteur de l'eau dans la vallée à chaque instant t. On considérera que le *débit de la source est unitaire*, c'est-à-dire d'une unité de surface (un carreau) par unité de temps. La figure 2 indique le niveau de l'eau à différentes dates t dans la vallée de la figure 1.

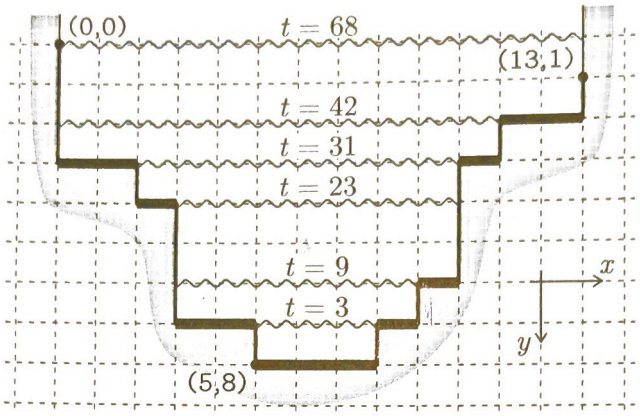


FIGURE 2 – Remplissage d'une vallée.

On appelle *plateau* tout segment horizontal *maximal* du profil de la vallée. Un plateau est défini par le triplet (x_0, x_1, y) où $x_0 < x_1$ sont les abscisses de ses deux extrémités et y est leur ordonnée. La vallée de la figure 2 possède exactement 8 plateaux, indiqués en gras sur la figure : `(0,2,3)`, `(2,3,4)`, `(3,5,7)`, `(5,8,8)`, `(8,9,7)`, `(9,10,6)`, `(10,11,3)`, `(11,13,2)`.

Question 7. Écrire une fonction `plateaux(v)` de complexité linéaire qui renvoie la liste des triplets correspondant aux plateaux de la vallée encodée par la liste `v`.

Remarquons que si l'on trie les plateaux d'une vallée du plus profond au moins profond (par y décroissants), on obtient une décomposition du volume intérieur de la vallée en rectangles. Ces rectangles sont délimités verticalement par les ordonnées consécutives des plateaux et horizontalement par les abscisses des extrémités des plateaux. La vitesse de montée des eaux est constante à l'intérieur de chaque rectangle et vaut exactement $1/w$ où w est la largeur du rectangle. L'eau met donc un temps hw à remplir chaque rectangle de taille $w \times h$. Dans le cas de la vallée illustrée ci-dessus la liste des tailles (w, h) des rectangles ainsi obtenus est, de bas en haut : `[(3,1), (6,1), (7,2), (8,1), (11,1), (13,-1)]` où la valeur `-1` de la dernière hauteur signifie que ce dernier rectangle est de hauteur infinie.

Question 8. Écrire une fonction `decomposition_en_rectangles(v)` de complexité linéaire qui renvoie la liste des tailles des rectangles, triés de bas en haut, décomposant le volume intérieur d'une vallée encodée par la liste `v`. Justifier le bon fonctionnement de votre algorithme.

Question 9. Écrire une fonction `hauteur_de_l_eau(t,v)` qui pour tout nombre flottant `t` $\geqslant$ `0.0`, renvoie la hauteur de l'eau (mesurée depuis le fond) dans une vallée encodée par la liste `v`.

Partie III : Grottes à ciel ouvert

Une grotte est dite *à ciel ouvert* si son profil est simple et ne contient aucun pas vers la Gauche.

Nous dirons que le profil d'une grotte à ciel ouvert est *normalisé* si le point à la fin du profil est situé à la même profondeur que l'origine, 0, et si tous les autres points du profil sont à une profondeur au moins égale à 1.

La figure 3 présente deux profils d'une même grotte à ciel ouvert : l'un normalisé (à droite) et l'autre non (à gauche).

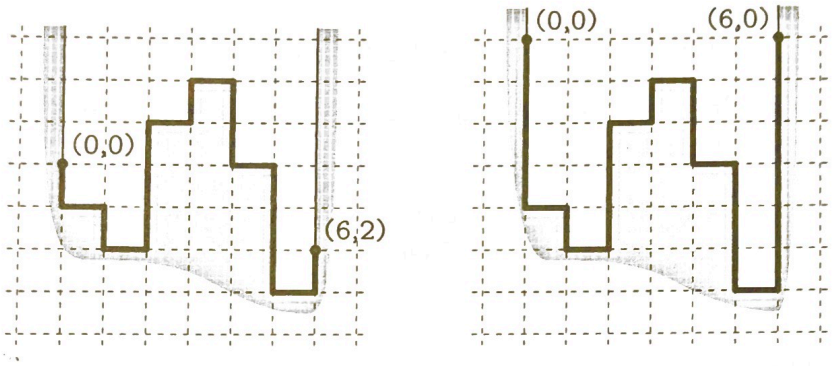


FIGURE 3 – Deux profils, non-normalisé (à gauche) et normalisé (à droite), d'une même grotte à ciel ouvert.

On suppose désormais que les profils seront tous à ciel ouvert et normalisés jusqu'à la fin de cette partie. Remarquons qu'un profil normalisé contient exactement le même nombre de pas `B` que de pas `H`. Cette propriété sera utile pour le bon déroulement des algorithmes ci-dessous.

Pour déterminer l'ordre de remplissage de la grotte, nous allons procéder comme précédemment en la découpant en rectangles, sauf que cette fois-ci, pour simplifier, *tous les rectangles de la décomposition seront de hauteur* 1 (sauf le dernier qui est de hauteur infinie).

Dans le cas d'une grotte à ciel ouvert, les rectangles qui se remplissent les uns après les autres ne sont plus les uns au-dessus des autres mais organisés hiérarchiquement : chaque rectangle qui n'est pas au fond de la grotte est le "parent" d'un ou plusieurs rectangles "enfants" au-dessous de lui que l'on liste de gauche à droite. La figure 4 page suivante montre la structure hiérarchique parent-enfant pour les 12 rectangles composant la grotte à ciel ouvert décrite par un profil normalisé.

Cette structure hiérarchique sera encodée par 4 listes **`origine`**, **`largeur`**, **`parent`**, **`enfants`**, de la façon suivante :

- les n rectangles seront numérotés de 0 à $n-1$;
- `origine[i]` contiendra le couple d'entiers correspondant aux coordonnées du coin inférieur gauche du rectangle n°`i` ;

```
[ B,B,B,D,H,D,B,B,D,H,H,H,D,B,B,D,H,D,B,B,D,H,D,B,D,H,H,D,B,B,D,H,H,H,H ]
```

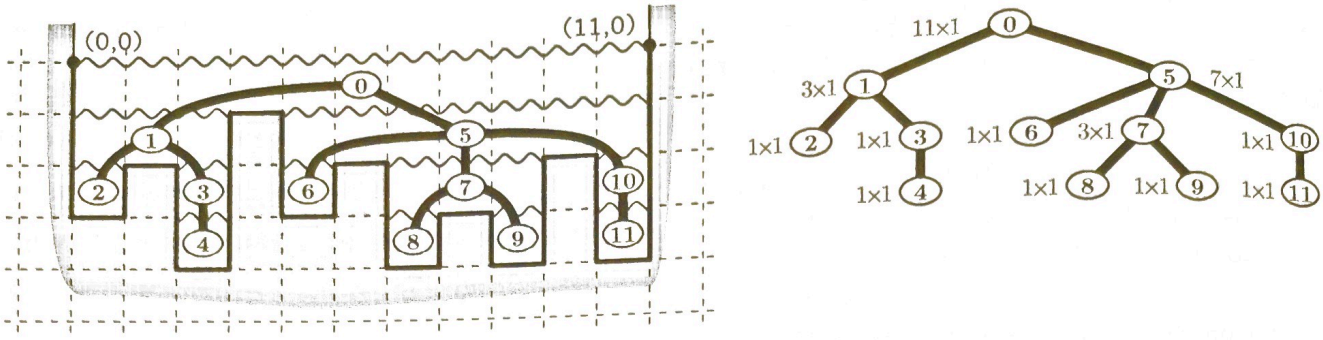


FIGURE 4 – La structure hiérarchique des rectangles.

- `largeur[i]` contiendra la largeur du rectangle n°`i` ;
- `parent[i]` contiendra le numéro du rectangle parent du rectangle n°`i` (ou `-1` si c'est le rectangle au sommet de la hiérarchie) ;
- `enfants[i]` contiendra la liste des numéros *de gauche à droite* des rectangles enfants du rectangle n°`i` (cette liste sera vide, `[]`, si ce rectangle n'a pas d'enfants).

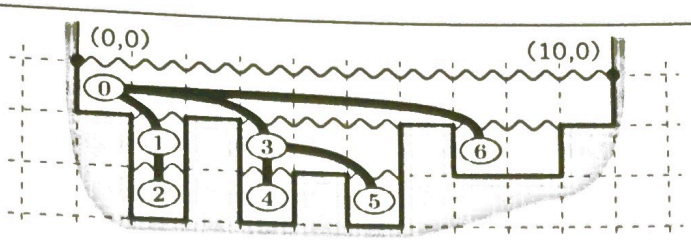
Voici les valeurs de ces quatre listes pour la grotte de la figure 4 :

```
origine = [(0,1),(0,2),(0,3),(2,3),(2,4),(4,2),(4,3),(6,3),(6,4),(8,4),(10,3),(10,4)]
largeur = [11, 3, 1, 1, 1, 7, 1, 3, 1, 1, 1, 1]
parent = [-1, 0, 1, 1, 3, 0, 5, 5, 7, 7, 5, 10]
enfants = [[1,5], [2,3], [], [4], [], [6,7,10], [], [8,9], [], [], [11], []]
```

Nous allons dans un premier temps construire cette structure hiérarchique, puis nous l'utiliserons pour calculer le niveau de l'eau dans les différentes parties en fonction de la position de la source et du temps.

Algorithme de décomposition en rectangles. L'algorithme procède en parcourant le profil (normalisé !) de la grotte une seule fois en partant de l'origine. Tout au long de l'algorithme, on maintient une liste `pile` qui contient les numéros des rectangles *ouverts* dont on connait l'origine mais pas encore la largeur et qui peuvent donc avoir des enfants :

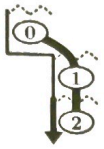
- Au départ : toutes les listes `pile`, `origine`, `largeur`, `parent`, `enfants` sont vides.
- Tout au long de l'algorithme, on maintient les coordonnées `(x,y)` du point où nous en sommes sur le profil.
- À chaque fois que le pas du profil est `B` : on crée un nouveau rectangle dont on stocke l'origine dans `origine`, dont on met la largeur temporairement à `-1` (car on ne la connaît pas encore), dont on initialise la liste des enfants à vide `[]`, et dont le parent est le numéro



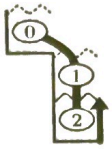
[B,D,B,B,D,H,H,D,B,B,D,H,D,B,D,H,H,D,B,D,D,H,D,H]

origine = [**(0,1)**]
largeur = [**−1**]
parent = [**−1**]
enfants = [**[]**]
pile = [**0**]

origine = [(0,1),**(1,2)**]
largeur = [−1,**−1**]
parent = [−1,**0**]
enfants = [[**1**],**[]**]
pile = [0,**1**]

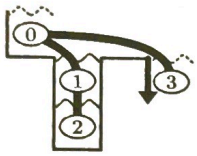


origine = [(0,1),(1,2),**(1,3)**]
largeur = [−1,−1,**−1**]
parent = [−1,0,**1**]
enfants = [[1],[**2**],**[]**]
pile = [0,1,**2**]

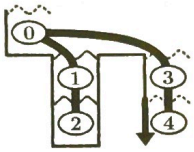


origine = [(0,1),(1,2),(1,3)]
largeur = [−1,−1,**1**]
parent = [−1,0,1]
enfants = [[1],[2],[]]
pile = [0,**1**]

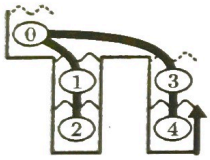
origine = [(0,1),(1,2),(1,3)]
largeur = [−1,**1**,1]
parent = [−1,0,1]
enfants = [[1],[2],[]]
pile = [**0**]



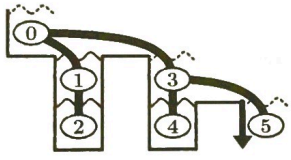
origine = [(0,1),(1,2),(1,3),**(3,2)**]
largeur = [−1,1,1,**−1**]
parent = [−1,0,1,**0**]
enfants = [[1,**3**],[2],[],**[]**]
pile = [0,**3**]



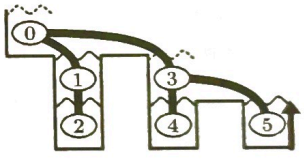
origine = [(0,1),(1,2),(1,3),(3,2),**(3,3)**]
largeur = [−1,1,1,−1,**−1**]
parent = [−1,0,1,0,**3**]
enfants = [[1,3],[2],[],[**4**],**[]**]
pile = [0,3,**4**]



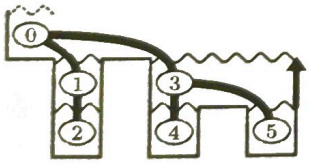
origine = [(0,1),(1,2),(1,3),(3,2),(3,3)]
largeur = [−1,1,1,−1,**1**]
parent = [−1,0,1,0,3]
enfants = [[1,3],[2],[],[4],[]]
pile = [0,**3**]



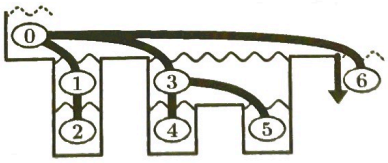
origine = [(0,1),(1,2),(1,3),(3,2),(3,3),**(5,3)**]
largeur = [−1,1,1,−1,1,**−1**]
parent = [−1,0,1,0,3,**3**]
enfants = [[1,3],[2],[],[4,**5**],[],**[]**]
pile = [0,3,**5**]



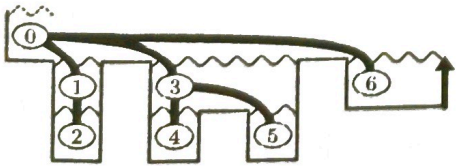
origine = [(0,1),(1,2),(1,3),(3,2),(3,3),(5,3)]
largeur = [−1,1,1,−1,1,**1**]
parent = [−1,0,1,0,3,3]
enfants = [[1,3],[2],[],[4,5],[],[]]
pile = [0,3]



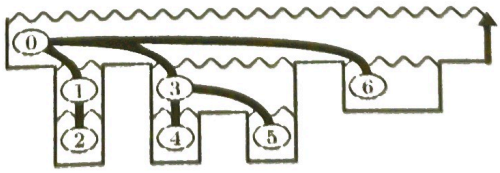
origine = [(0,1),(1,2),(1,3),(3,2),(3,3),(5,3)]
largeur = [−1,1,1,**3**,1,1]
parent = [−1,0,1,0,3,3]
enfants = [[1,3],[2],[],[4,5],[],[]]
pile = [**0**]



origine = [(0,1),(1,2),(1,3),(3,2),(3,3),(5,3),**(7,2)**]
largeur = [−1,1,1,3,1,1,**−1**]
parent = [−1,0,1,0,3,3,**0**]
enfants = [[1,3,**6**],[2],[],[4,5],[],[],**[]**]
pile = [0,**6**]



origine = [(0,1),(1,2),(1,3),(3,2),(3,3),(5,3),(7,2)]
largeur = [−1,1,1,3,1,1,**2**]
parent = [−1,0,1,0,3,3,0]
enfants = [[1,3,6],[2],[],[4,5],[],[],[]]
pile = [**0**]



origine = [(0,1),(1,2),(1,3),(3,2),(3,3),(5,3),(7,2)]
largeur = [**10**,1,1,3,1,1,2]
parent = [−1,0,1,0,3,3,0]
enfants = [[1,3,6],[2],[],[4,5],[],[],[]]
pile = [**]**]

FIGURE 5 – Une exécution de l'algorithme de décomposition en 7 rectangles de la grotte à ciel ouvert en haut : on peut suivre les modifications **en gras** de chacune des listes `pile`, `origine`, `largeur`, `parent` et `enfants` à chaque étape du parcours du profil de la grotte.

du rectangle au bout de la liste `pile` (ou `-1` si `pile` est vide) ; on l'ajoute à la liste des enfants de son père, puis on rajoute le numéro de ce rectangle nouvellement "ouvert" au bout de la liste `pile` des rectangles ouverts.

- À chaque fois que le pas du profil est `H` : on "ferme" le rectangle qui se trouve au bout de liste `pile` (qui contient les rectangles actuellement ouverts). Pour cela, on met à jour sa largeur en se basant sur la position actuelle et sur son origine stockée dans `origine` ; puis on retire son numéro de la liste `pile`.

La figure 5 page précédente exécute cet algorithme pas à pas sur un exemple, en montrant l'évolution des listes `pile`, `origine`, `largeur`, `parent` et `enfants` à chaque étape.

Le bon fonctionnement de cet algorithme est garanti par le fait que le profil est normalisé : à chaque ouverture d'un rectangle en suivant un pas `B` (correspondant à son bord gauche), correspond un pas `H` (son bord droit) pour sa fermeture.

Question 10. Écrire une fonction `hierarchie_rectangles(g)` qui renvoie le quadruplet des quatre listes `(origine, largeur, parent, enfants)` décrivant la hiérarchie de rectangles correspondant au profil *normalisé* `g`. Donner sa complexité.

Nous allons désormais exploiter cette structure hiérarchique pour calculer l'ordre de remplissage des rectangles de la grotte. Commençons par observer que cet ordre dépend de la position de la source. Sur la figure 6, la source (symbolisée par ▲) est placée soit à l'origine (figure de gauche), soit à l'origine du rectangle au milieu (figure de droite).

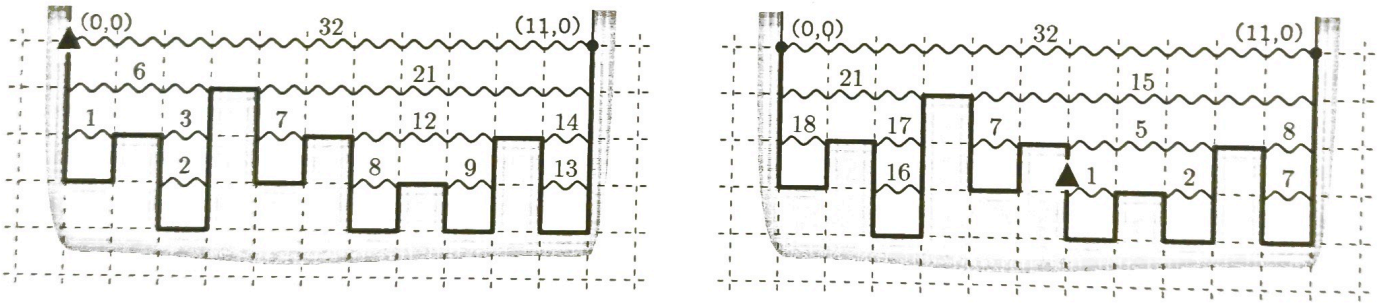


FIGURE 6 – Les dates et ordre de remplissage dépendent de la position de la source (symbolisée par ▲).

Les dates de remplissage des différents rectangles sont marquées au-dessus de leur bord supérieur. On constate que ces dates sont non seulement différentes, mais aussi que, lorsque la source est "au milieu" de la grotte, alors, plusieurs rectangles peuvent se remplir simultanément, comme c'est le cas des rectangles remplis entre $t = 5$ et $t = 7$ dans la figure de droite. Cette situation n'est cependant pas possible quand la source est située tout à gauche de la grotte, à la position $(0, 0)$ (admis).

Le cas de la source située à l'origine. On supposera que l'eau s'écoule instantanément verticalement et prend donc un temps nul à dévaler les pentes (comme cela a été supposé dans les deux chronologies de la figure 6). On se place dans le cas où la source est située à l'origine.

On admet alors que l'eau remplit les rectangles de gauche à droite, un seul à la fois. On admettra également que chaque rectangle commencera à se remplir une fois que l'ensemble de ses rectangle-enfants seront remplis et que ceux-ci se remplissent l'un après l'autre de gauche à droite.

Question 11. Écrire une fonction `ordre_remplissage_depuis_origine(parent,enfants)` qui prend en entrée les deux listes `parent` et `enfants` décrivant la hiérarchie des rectangles et renvoie la liste des numéros des rectangles dans l'ordre dans lequel ils se remplissent. Donner sa complexité.

Question 12. Écrire une fonction `hauteurs_eau_depuis_origine(t, largeur, parent, enfants)` qui prend en entrée un flottant `t`, et les trois listes `largeur`, `parent` et `enfants`, décrivant la hiérarchie des rectangles d'une grotte à ciel ouvert, et renvoie une liste `hauteur` où `hauteur[`i`]` est la hauteur d'eau dans le rectangle n°i à l'instant `t` (la hauteur sera donc un flottant entre 0 et 1 sauf pour le dernier rectangle qui est infini et peut donc être rempli à une hauteur arbitrairement grande). Expliciter sa complexité.

Le cas d'une source à une position arbitraire. Comme nous l'avons vu précédemment, lorsque la source est à une position arbitraire, il est possible que plusieurs rectangles se remplissent simultanément : quand un bassin est plein, l'eau s'écoule alors équitablement des deux côtés, comme illustré sur la figure 6 à droite entre les dates $t = 5$ et $t = 7$.

Question 13. Expliquez pourquoi jamais plus de deux rectangles ne se rempliront simultanément. Votre réponse ne devrait pas excéder 5 lignes.

Question 14. Écrire une fonction `volumes_totaux(largeur, parent, enfants)` qui prend en entrée les trois listes `largeur`, `parent` et `enfants` décrivant la hiérarchie des rectangles, et qui renvoie une liste `volume` telle que `volume[`i`]` est la somme des volumes des rectangles descendants du rectangle n°i, i inclus.

Question 15. Décrire un *algorithme* qui prend en entrée le numéro `source` du rectangle à l'origine duquel est située la source, les trois listes `largeur`, `parent` et `enfants` décrivant la hiérarchie des rectangles, et qui renvoie une liste `hauteur` telle que `hauteur[`i`]` est la hauteur d'eau présente dans le rectangle n°i à l'instant `t`. On pourra utiliser les procédures définies ci-dessus. On ne demande pas l'écriture d'un programme mais la présentation argumentée d'une solution algorithmique à ce problème.

Justifier le fonctionnement de votre algorithme. Expliciter sa complexité.

★
★ ★
★