Durée de l'épreuve : 3h.
Aucun document autorisé. Aucun matériel électronique autorisé : calculatrice non autorisée.
Le sujet est à traiter dans l'ordre. Les codes demandés dans les deux premières parties seront écrits en OCaml.
Les codes de la dernière partie en C.

Spéléo-logique

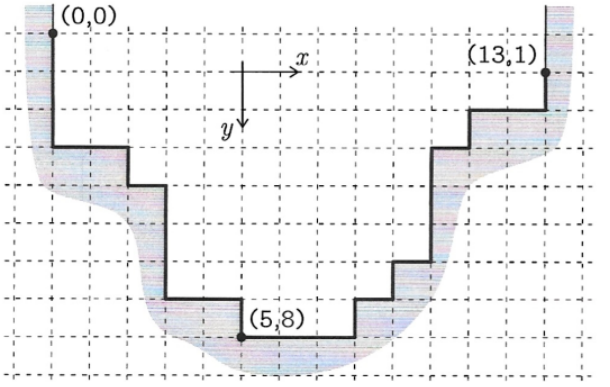
Sujet informatique commune, X-ENS 2022.

« *Une phrase courte et claire prend moins de temps à écrire que des pensées confuses...*
Utilisez du brouillon ![1] »

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. Pour les deux premières parties, traitées en Ocaml, on supposera défini le type `direction` suivant :

```
type direction = B | H | D | G
```

représentant les directions verticales (`B` : bas, `H` : haut) et horizontales (`G` : gauche, `D` : 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. Un profil sera représenté en OCaml par une liste de direction.
La figure 1 donne l'exemple d'une grotte et de son encodage.




```
[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` qui prend en argument une liste `g` décrivant un profil de grotte et renvoie `true` si et seulement si le profil est sans rebroussement, `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` qui prend en argument une liste `g` décrivant un profil de grotte et renvoie `true` si et seulement si le profil est celui d'une vallée, `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)`.

1. En-tête du sujet initial...

Question 3 Écrire une fonction `voisin` de type `int -> int -> direction -> (int * int)` qui prend en argument 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` qui prend en argument une liste `g` décrivant un profil et renvoie la liste des coordonnées $[(x_0, y_0), ..., (x_n, y_n)]$ 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` qui prend en argument la liste `g` décrivant le profil et renvoie `true` si et seulement si le profil est simple, `false` sinon. Justifier rapidement 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 a pour coordonnées `(5, 8)`.

Question 6 Écrire une fonction `fond` 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.

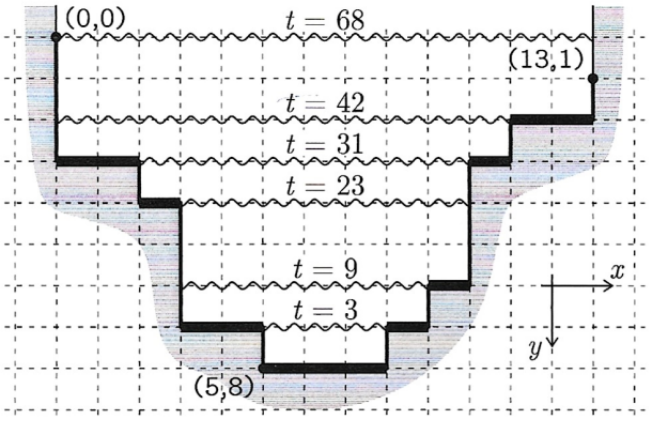


FIGURE 2 – Remplissage d'une vallée

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` de **complexité linéaire** en temps 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 ordonnéees 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 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 indique que le dernier rectangle est de hauteur infinie.

Question 8 Écrire une fonction `decomposition_en_rectangles` 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 une liste `v` donnée en argument.

Question 9 Écrire une fonction `hauteur_de_l_eau` qui prend en argument un nombre flottant `t` ≥ 0 et l'encodage d'une valléee `v` et renvoie la hauteur de l'eau (mesurée depuis le fond) dans la vallée au temps `t`. La hauteur de l'eau sera exprimée sous la forme d'un nombre flottant également.
En OCaml, il est possible de convertir un entier en nombre flottant en utilisant le fonction `float_of_int` de type `int -> float` (ou inversement : `int_of_float` de type `float -> int`).

Partie III : Grottes à ciel ouvert

```
typedef enum
{
  B, H, G, D
} direction;
```

Il est possible en langage C de définir comme ci-contre un type **énumération**. Ici les valeurs de `B`, `H`, `G`, `D` sont de type `direction`. Ces valeurs seront utilisées pour représenter les directions des profils des grottes dans cette partie.

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 de 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 non normalisé (à gauche) et l'autre normalisé (à droite).

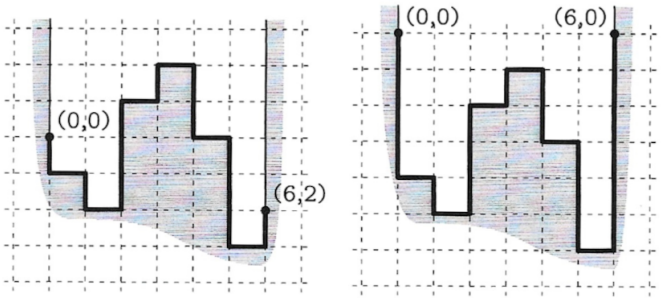


FIGURE 3 – Deux profils, non-normalisé (à gauche) et normalisé (à droite), d'une même grotte à ciel ouvert

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 montre la structure hiérarchique parent-enfant pour les 12 rectangles qui composent la grotte à ciel ouvert décrite par le profil normalisé représenté en C par le tableau `profil` ci-dessous :

```
direction profil[] = {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 };
```

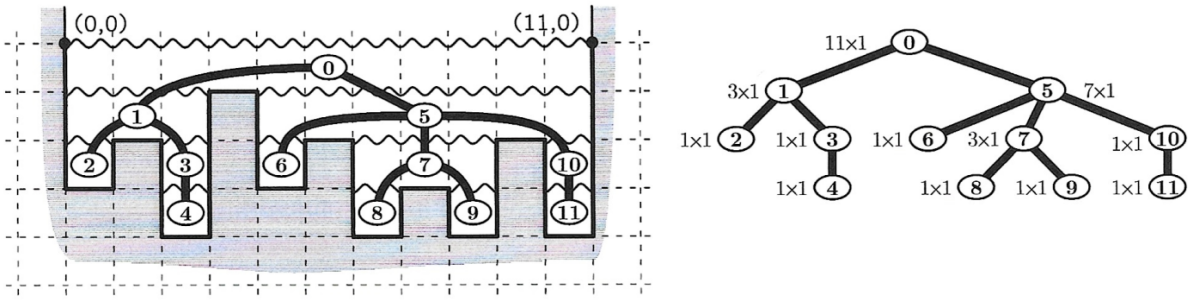


FIGURE 4 – La structure hiérarchique des rectangles

On se dote d'une structure de données `tableau`, de tableau redimensionnable permettant de stocker des entiers et qui supporte les opérations indiquées dans la figure 5, page 4. Il n'est pas demandé d'écrire le code de ces fonctions !

Spécification fonctionnelle de la structure de tableau redimensionnable

```
/* Crée un nouveau tableau redimensionnable et renvoie un pointeur
 * sur la structure associée.  */
tableau* creer_tableau();

/* Ajoute un nouvel élément dans le tableau.
 * Entrée : t  un pointeur vers un tableau redimensionnable
 *          v  valeur du nouvel élément
 * Sortie : pas de valeur de retour, le tableau contient
 *          le nouvel élément.  */
void push_back(tableau* t, int v);

/* Renvoie la valeur d'un élément à l'indice donné.
 * Entrée : t  un pointeur vers un tableau redimensionnable
 *          i  la valeur d'un indice valide
 * Sortie : la valeur de l'élément du tableau à l'indice donné */
int get(tableau* t, int i);

/* Modifie la valeur de l'élément à l'indice donné.
 * Entrée : t  un pointeur vers un tableau redimensionnable
 *          i  l'indice d'un élément dans le tableau
 *          v  la nouvelle valeur de l'élément
 * Sortie : pas de valeur de retour, la valeur de l'élément a été
 *          modifiée dans le tableau.  */
void set(tableau* t, int i, int v);

/* Renvoie le nombre d'éléments du tableau.
 * Entrée : t  un pointeur vers un tableau redimensionnable
 * Sortie : le nombre d'éléments du tableau.  */
int taille(tableau* t);

/* Libère l'espace associé à un tableau redimensionnable.
 * Entree : t  un pointeur vers un tableau redimensionnable
 * Sortie : pas de valeur de retour, la mémoire allouée dans le tas
 *          par la structure est libérée, le pointeur t ne doit plus
 *          être utilisé.  */
void liberer_tableau(tableau* t);
```

FIGURE 5 – Spécification de la structure de tableau redimensionnable fournie.

La structure hiérarchique sera encodée **en C** par 4 tableaux redimensionnables `origine_x`, `origine_y`, `largeur`, `parent` et par un tableau de tableaux redimensionnables `enfants`. Cet encodage sera exprimé de la manière suivante :

- les n rectangles seront numérotés de 0 à $n-1$,
- `origine_x[i]` contiendra l'abscisse des coordonnées du coin inférieur gauche du rectangle numéro i,
- `origine_y[i]` contiendra l'ordonnée des coordonnées du coin inférieur gauche du rectangle numéro i,
- `largeur[i]` contiendra la largeur du rectangle numéro i,
- `parent[i]` contiendra le numéro du rectangle parent du rectangle numéro i, ou -1 s'il s'agit du rectangle au sommet de la hiérarchie,
- `enfants[i]` contiendra un tableau redimensionnable contenant les numéros *de gauche à droite* des rectangles enfants du rectangle numéro i, le tableau contenant les numéros des enfants du rectangle numéro i sera de taille nulle si le rectangle n'a pas d'enfants.

Voici une **représentation** des valeurs de ces cinq tableaux pour la grotte de la figure 4 :

```
origine_x : [0, 0, 0, 2, 2, 4, 4, 6, 6, 8, 10, 10]
origine_y : [1, 2, 3, 3, 4, 2, 3, 3, 4, 4, 3, 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 `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. On maintient également les coordonnées `(x, y)` du point en cours de traitement sur le profil.

tous les tableaux redimensionnables `origine_x`, `origine_y`, `largeur` et `parent` sont vides
le tableau `enfants` est initialisé et les sous-tableaux sont vides
la pile `pile` est vide
la position actuelle est l'origine
pour chaque pas du profil **faire**
  mettre à jour la position actuelle
  si le pas est `B` **alors**
    on crée un nouveau rectangle dont on stocke les coordonnées de l'origine dans `origine_x` et `origine_y`, dont on met la largeur temporairement à `-1` (car on ne la connait pas encore), dont on initialise la liste des enfants à vide [], et dont le parent est le numéro du rectangle au sommet de la `pile` (ou `-1` si la pile est vide) ; on l'ajoute à la liste des enfants de son père, puis ajoute le numéro de ce rectangle nouvellement « ouvert » au sommet de la pile.
  fin
  si le pas est `H` **alors**
    on « ferme » le rectangle qui se trouve au sommet de la `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_x` et `origine_y` puis on retire son numéro de la `pile`.
  fin
fin

La figure 8, page 8, représente l'exécution cet algorithme pas à pas sur un exemple, en montrant l'évolution des tableaux `origine`, `largeur`, `parent` et `enfants` et de la pile `pile`.

Attention : dans la figure 8, l'origine de chaque rectangle est représentée par un couple de coordonnées entières alors que dans ce sujet, les abscisses seront stockées dans le tableau `origine_x` et les ordonnées dans le tableau

`origine_y`.

Question 10 Expliquer rapidement ce qui garantit le bon fonctionnement de cet algorithme.

On considère que le profil est représenté par un tableau `g` dont la taille est donnée par la variable `taille_profil`.

Question 11 Écrire les initialisations à effectuer pour préparer `origine_x`, `origine_y`, `largeur`, `parent` et `enfants`. Expliquer les choix effectués le cas échéant.

On se dote également d'une structure de `pile` avec la spécification suivante (figure 6, page 6).
Rappel : il n'est pas demandé d'écrire le code de ces fonctions !

Spécification fonctionnelle d'une structure de pile

```
/* Crée une pile.
 * Entrée : c   capacité max de la pile
 * Sortie : un pointeur vers une pile pouvant empiler successivement
 * c éléments au maximum.  */
pile* creer_pile(int c);

/* Détermine si une pile est vide.
 * Entrée : p    un pointeur vers une structure de pile
 * Sortie : true si la pile est vide, false sinon.  */
bool est_vide(pile* p);

/* Empile un élément.
 * Entrée : p   un pointeur vers une structure de pile
 *          v   une valeur à empiler
 * Sortie : pas de valeur de retour, si la capacité était suffisante,
 *          la valeur v a été ajoutée au sommet de la pile.  */
void empiler(pile* p, int v);

/* Dépile un élément.
 * Entrée : p   un pointeur vers une structure de pile
 * Sortie : la valeur de l'élément dépilé, de plus cet élément
 *          n'est plus au sommet de la pile.  */
int depiler(pile* p);

/* Permet de consulter la valeur du sommet de la pile.
 * Entrée : p    un pointeur vers une structure de pile
 * Sortie : la valeur de l'élément au sommet de la pile, la pile
 *          n'est pas modifiée.  */
int sommet(pile* p);

/* Libère la mémoire utilisée par une pile.
 * Entrée : p   un pointeur vers une structure de pile.
 * Sortie : pas de valeur de retour, toute la mémoire allouée
 *          sur le tas pour la pile est libérée :
 *          le pointeur ne doit plus être utilisé.  */
void liberer_pile(pile* p);
```

FIGURE 6 – Spécification de la structure de pile fournie.

Question 12 Écrire une fonction `hierarchie_rectangles` qui met à jour les tableaux `origine`, `largeur`, `parent`, `enfants` décrivant la hiérarchie de rectangles correspondant au profil *normalisé* `g` en utilisant l'algorithme décrit précédemment.
Justifier 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 7 page 7 la source (symbolisée par ▲) est placée soit à l'origine (figure de gauche), soit à l'origine du rectangle au milieu (figure de droite).

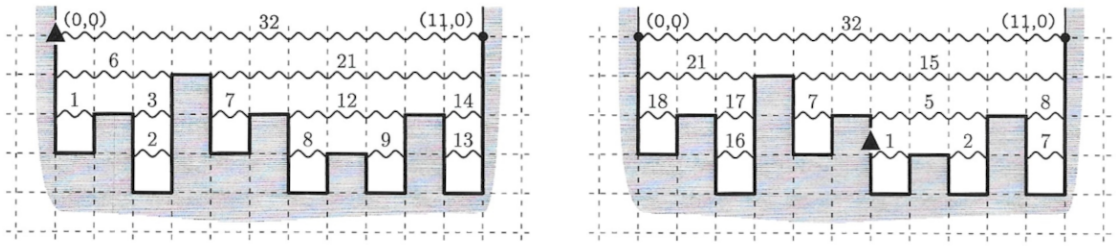


FIGURE 7 – Les dates et ordre de remplissage dépendent de la position de la source (symbolisée par ▲)

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 7). 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 13 Écrire une fonction `ordre_remplissage_depuis_origine` qui prend en entrée les informations utiles de la structure hiérarchique des rectangles et met à jour un tableau redimensionnable `ordre` contenant le numéro des rectangles dans l'ordre dans lequel ils se remplissent. Spécifier la fonction selon les choix effectués.
Justifier sa complexité.

Question 14 Écrire une fonction `hauteurs_eau_depuis_origine` qui prend en entrée les informations utiles de la structure, un temps `t` exprimé par un nombre flottant, un tableau de flottants `hauteur`, et met à jour `hauteur` de manière à ce que `hauteur[i]` soit la hauteur d'eau dans le rectangle numéro 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).
Justifier 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 7 à droite entre les dates $t = 5$ et $t = 7$.

Question 15 Expliquez pourquoi jamais plus de deux rectangles ne se rempliront simultanément. Votre réponse ne devrait pas excéder 5 lignes.

Question 16 Écrire une fonction `volumes_totaux` qui prend en entrée certains des éléments décrivant la hiérarchie des rectangles, et qui renvoie ou met à jour un tableau `volume` telle que `volume[i]` est la somme des volumes des rectangles descendants du rectangle numéro i, i inclus.

Question 17 Décrire un *algorithme* qui prend en entrée le numéro `source` du rectangle où est située la source, un temps `t` exprimé par un nombre flottant, les éléments de la hiérarchie des rectangles de la grotte qui vous paraîtront utiles, et qui permet de renvoyer un tableau `hauteur` telle que `hauteur[i]` soit la hauteur d'eau présente dans le rectangle numéro 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.

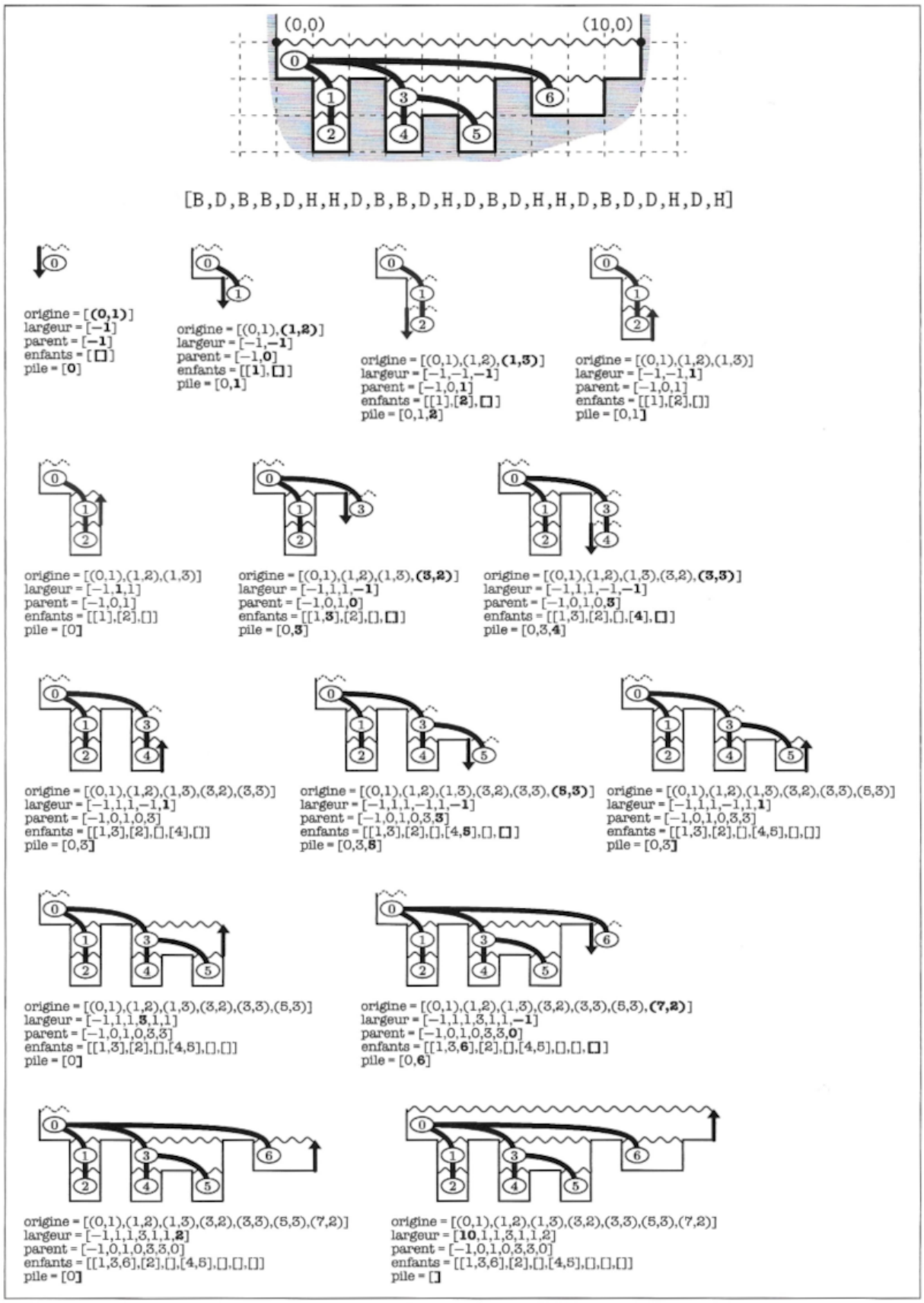


FIGURE 8 – Une exécution de l'algorithme de décomposition en 7 rectangles de la grotte à ciel ouvert en haut : on peut suivre les modifications **en gras** de la représentation de chaque tableau et de la pile.