

# Projet — Spéléo-logique

Adapté d'un sujet informatique commune, X-ENS 2022.

**Consignes.** Ce devoir est un projet de programmation à réaliser en autonomie cet été pour préparer au mieux votre rentrée en classe de MPI/MPI\*. L'objectif est de travailler la programmation dans les deux langages OCaml et C, tout en révisant certaines notions de MP2I, comme les piles et les arbres.

Toutes les questions sont des questions de programmation, à réaliser sur machine. Du code est fourni et doit être complété. Le devoir est à rendre avant le jour de la rentrée, par mail à l'adresse suivante :

remi.hutin@ac-dijon.fr

(et si par ailleurs vous découvrez ce qui vous semble être une erreur d'énoncé, faites m'en part à cette adresse).

Ce projet comporte deux grandes parties.

- La **partie I** est à réaliser en OCaml. Pour chaque question de cette partie, on fournit un jeu de test permettant de tester votre proposition : si le test échoue, c'est que votre implémentation n'est pas correcte ; et s'il réussit, vous pouvez passer à la question suivante. Vous pouvez ainsi progresser en autonomie sur cette partie.
- La **partie II** est à réaliser en C. Dans cette partie, les tests seront à réaliser par vos soins, afin de vous assurer de la validité de vos réponses.

★ ★ ★

## I. Grottes et vallées

Cette partie est à traiter en OCaml. Toutes les fonctions à écrire sont à compléter dans le fichier « `vallee.ml` ». À chaque fois que vous avez une fonction à écrire, vous trouverez une instruction « `failwith "à compléter"` », qu'il vous faudra supprimer, pour la remplacer par votre solution.

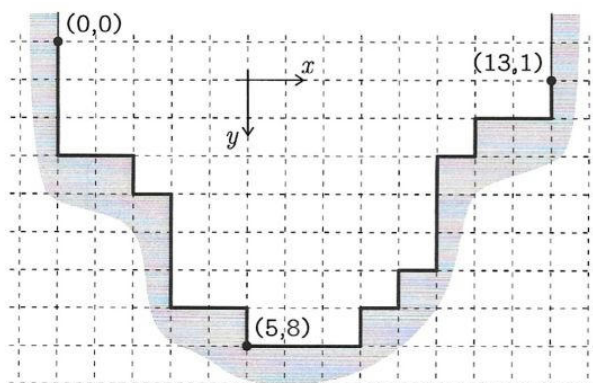
**Le problème.** Nous allons déterminer le remplissage d'une grotte lors d'une inondation alimentée par une source d'eau localisée quelque part dans la grotte. La grotte considérée sera bi-dimensionnelle et décrite par le profil de son fond. Pour les deux premières parties, traitées en OCaml, on définit 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 ( $B^\infty$  et  $H^\infty$ ). 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 Fig. 1 ci-dessous 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]

Fig. 1. — Une grotte et son profil.

## I. A. 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.** Compléter la fonction `est_sans_rebroussement` qui prend en paramètre une liste `g` décrivant un profil de grotte et renvoie `true` si le profil est sans rebroussement, `false` sinon. **Valider le test avant de passer à la suite.**

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 Fig. 1 est une vallée.

**Question 2.** Compléter la fonction `est_une_vallee` qui prend en paramètre une liste `g` décrivant un profil de grotte et renvoie `true` si le profil est celui d'une vallée, `false` sinon. **Valider le test avant de passer à la suite.**

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.** Compléter la fonction `voisin : (int * int) → direction → (int * int)` qui prend en paramètre un couple de coordonnées OCaml  $(x, y)$  et une direction  $d \in \{H, B, G, D\}$  et renvoie le couple de coordonnées du voisin du point  $(x, y)$  dans la direction  $d$ . (*Pas de test pour cette question.*)

**Question 4.** Compléter la fonction `liste des points` qui prend en paramètre une liste `g` décrivant un profil et renvoie la liste des coordonnées  $[(x_0, y_0), \dots, (x_n, y_n)]$  des points de l'origine à l'arrivée du profil. **Valider le test avant de passer à la suite.**

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

**Question 5.** Compléter la fonction `est simple` qui prend en paramètre la liste `g` décrivant le profil et renvoie `true` si le profil est simple, `false` sinon. Justifier rapidement sa complexité en commentaire de la fonction. **Valider le test avant de passer à la suite.**

## I. B. 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 Fig. 2 a pour coordonnées  $(5, 8)$ .

**Question 6.** Compléter la fonction `fond` qui renvoie les coordonnées  $(x, y)$  du fond de la vallée encodée par la liste des directions `v`. **Valider le test avant de passer à la suite.**

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 Fig. 2 indique le niveau de l'eau à différentes dates  $t$  dans la vallée de la Fig. 1.

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 son ordonnée. La vallée de la Fig. 2 possède exactement 8 plateaux, indiqués en gras sur la figure :  $(0, 2, 3)$ ,  $(2, 3, 4)$ ,  $(3, 5, 7)$ ,  $(5, 8, 8)$ ,  $(8, 9, 7)$ ,  $(9, 10, 6)$ ,  $(10, 11, 3)$  et  $(11, 13, 2)$ .

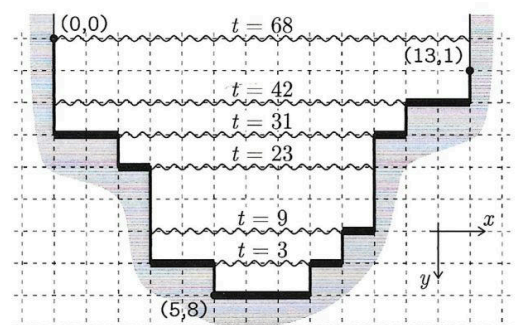


Fig. 2. – Remplissage d'une vallée.

**Question 7.** Compléter la fonction `plateaux` de complexité linéaire en temps qui renvoie la liste des triplets correspondant aux plateaux de la vallée encodée par la liste `v`. **Valider le test avant de passer à la suite.**

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 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.** Compléter la fonction `decomposition_en_rectangles` 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. **Valider le test avant de passer à la suite.**

**Bonus :** Écrire cette fonction avec une complexité linéaire en la taille de `v`.

*Indication :* On rappelle qu'en OCaml, il est possible de convertir un entier en nombre flottant en utilisant la fonction `float_of_int` : `int`  $\rightarrow$  `float` (et inversement avec `int_of_float` : `float`  $\rightarrow$  `int`).

**Question 9.** Compléter la fonction `hauteur_de_l_eau` qui prend en argument un nombre flottant  $t \geq 0$  et l'encodage d'une vallée `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. **Valider le test pour conclure cette partie.**

## II. Grottes à ciel ouvert

Cette partie est à traiter en C. Toutes les fonctions à écrire sont à compléter dans le fichier « `grotte.c` ». Toutes les définitions de fonction sont déjà données, et vous trouverez des commentaires `/** A compléter */`, que vous devrez à chaque fois remplacer par votre solution. Vous trouverez également des fonctions d'affichage fournies au début du fichier. Pour cette partie, il y a peu de tests fournis : à vous donc de les écrire pour tester toutes vos fonctions.

**Rappel :** Pour compiler votre programme, utilisez la commande suivante :

```
gcc -Wall -Wextra grotte.c -o grotte
```

Des avertissements apparaîtront tant que les fonctions n'auront pas été complétées : c'est normal.

En langage C, il est possible de définir un type *enumération* comme ci-contre. Grâce à cette définition, on dispose (comme pour la partie précédente en OCaml) de quatre valeurs `B`, `H`, `D` et `G`, de type « `direction` ». On utilisera ces valeurs pour représenter les directions des profils de grotte dans cette partie.

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

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

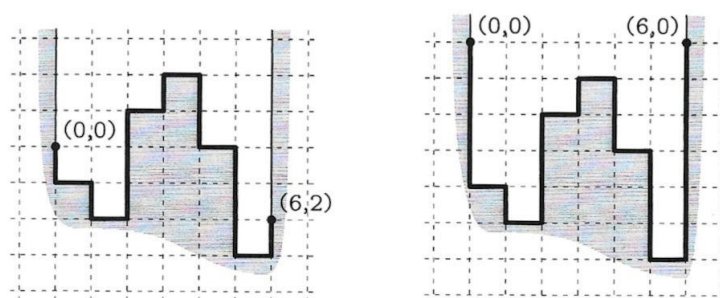


Fig. 3. – Deux profils, non-normalisé (à gauche) et normalisé (à droite), d'une même grotte à ciel ouvert

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 };
```

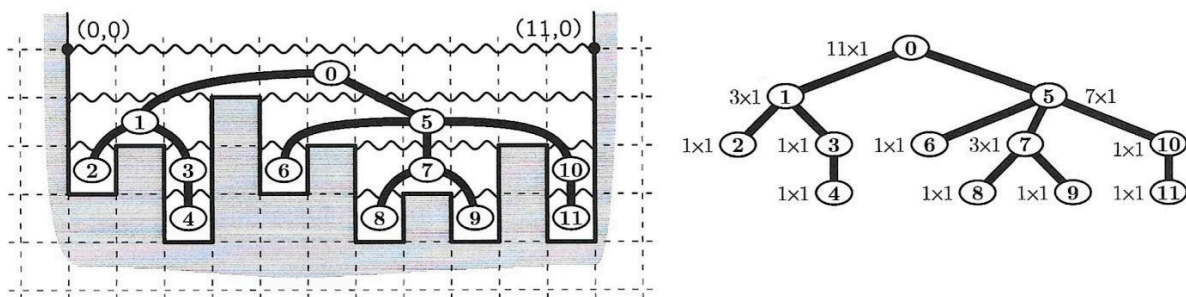


Fig. 4. – La structure hiérarchique des rectangles

La structure hiérarchique sera encodée en C par 5 tableaux : **origine\_x**, **origine\_y**, **largeur**, **parent** et par un tableau de tableaux enfants. Cet encodage sera exprimé de la façon suivante :

- ▶ les  $n$  rectangles seront numérotés de 0 à  $n - 1$  ;
- ▶ **origine\_x[i]** contiendra l'abscisse du coin inférieur gauche du rectangle d'indice  $i$  ;
- ▶ **origine\_y[i]** contiendra l'ordonnée du coin inférieur gauche du rectangle d'indice  $i$  ;
- ▶ **largeur[i]** contiendra la largeur du rectangle d'indice  $i$  ;
- ▶ **parent[i]** contiendra l'indice du rectangle parent du rectangle d'indice  $i$ , ou  $-1$  s'il s'agit du rectangle au sommet de la hiérarchie ;
- ▶ **nb\_enfants[i]** contiendra le nombre d'enfants du rectangle d'indice  $i$  (0 si un rectangle n'a pas d'enfants) ;
- ▶ **enfants[i]** contiendra un tableau, dont les valeurs seront les indices des rectangles enfants du rectangle d'indice  $i$ . Comme on ne sait pas *a priori* combien d'enfants aura chaque rectangle, les tableaux **enfants[i]** seront tous de taille  $n$ , et seules les **nb\_enfants[i]** premières cases seront utilisées.

La grille ci-dessous donne les valeurs de ces tableaux pour la grotte de la Fig. 3 :

	0	1	2	3	4	5	6	7	8	9	10	11
<b>origine_x</b>	0	0	0	2	2	4	4	6	6	8	10	10
<b>origine_y</b>	1	2	3	3	4	2	3	3	4	4	3	4
<b>largeur</b>	11	3	1	1	1	7	1	3	1	1	1	1
<b>parent</b>	-1	0	1	1	3	0	5	5	7	7	5	10
<b>nb_enfants</b>	2	2	0	1	0	3	0	2	0	0	1	0
<b>enfants</b>	[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.

Pour représenter cette hiérarchie, on fixe le type **hierarchie** ci-contre. Ce type contient tous les tableaux décrits précédemment, auxquels on a ajouté un champ **taille** représentant le nombre de rectangles dans la hiérarchie, et qui correspond donc à la taille de chacun des tableaux.

```
typedef struct {
    int taille;
    int* origine_x;
    int* origine_y;
    int* largeur;
    int* parent;
    int* nb_enfants;
    int** enfants;
} hierarchie;
```

Pour déterminer la taille de la hiérarchie (c'est-à-dire le nombre de rectangles) d'un profil normalisé, on peut se contenter de compter le nombre de pas vers le Bas dans le profil.

**Question 10.** Compléter la fonction `int nombre_B(direction profil[], int taille_profil)` qui compte le nombre de B dans le tableau de direction `profil` de taille `taille_profil`.

*Indication : on sera vigilant à ne pas confondre la taille du profil et la taille de la hiérarchie dans la suite du sujet.*

**Question 11.** Compléter la fonction `hierarchie init_hierarchie(int taille)` qui initialise et renvoie une hiérarchie de `taille` donnée. On veillera à allouer toute la mémoire nécessaire, y compris pour les tableaux de tableaux.

**Question 12.** Compléter la fonction `void liberer_hierarchie(hierarchie h)` qui libère toute la mémoire allouée par la fonction `init_hierarchie`.

## II. A. 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 connaît 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.

- **Initialement** : la *pile* est vide, la *position* actuelle est  $(0, 0)$ .
- **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`, donc on met la largeur temporairement à  $-1$  (car on ne la connaît pas encore), donc on initialise le nombre d'enfants à  $0$ , et donc le parent est le sommet de la *pile* (ou  $-1$  si la *pile* est vide).
    - On ajoute ce rectangle aux enfants de son père, sans oublier d'augmenter le nombre d'enfants du père.
    - On ajoute l'indice de ce rectangle à la *pile*.
  - **Sinon** 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* à partir de la *position* actuelle et de l'*origine* de ce rectangle.
    - Puis on retire l'indice de ce rectangle de la *pile*.

La Fig. 5 (page suivante), représente l'exécution cet algorithme pas à pas sur un exemple, en montrant l'évolution des tableaux `origine` (combinant `origine_x` et `origine_y` pour cette figure uniquement), `largeur`, `parent` et `enfants` et de la *pile*.



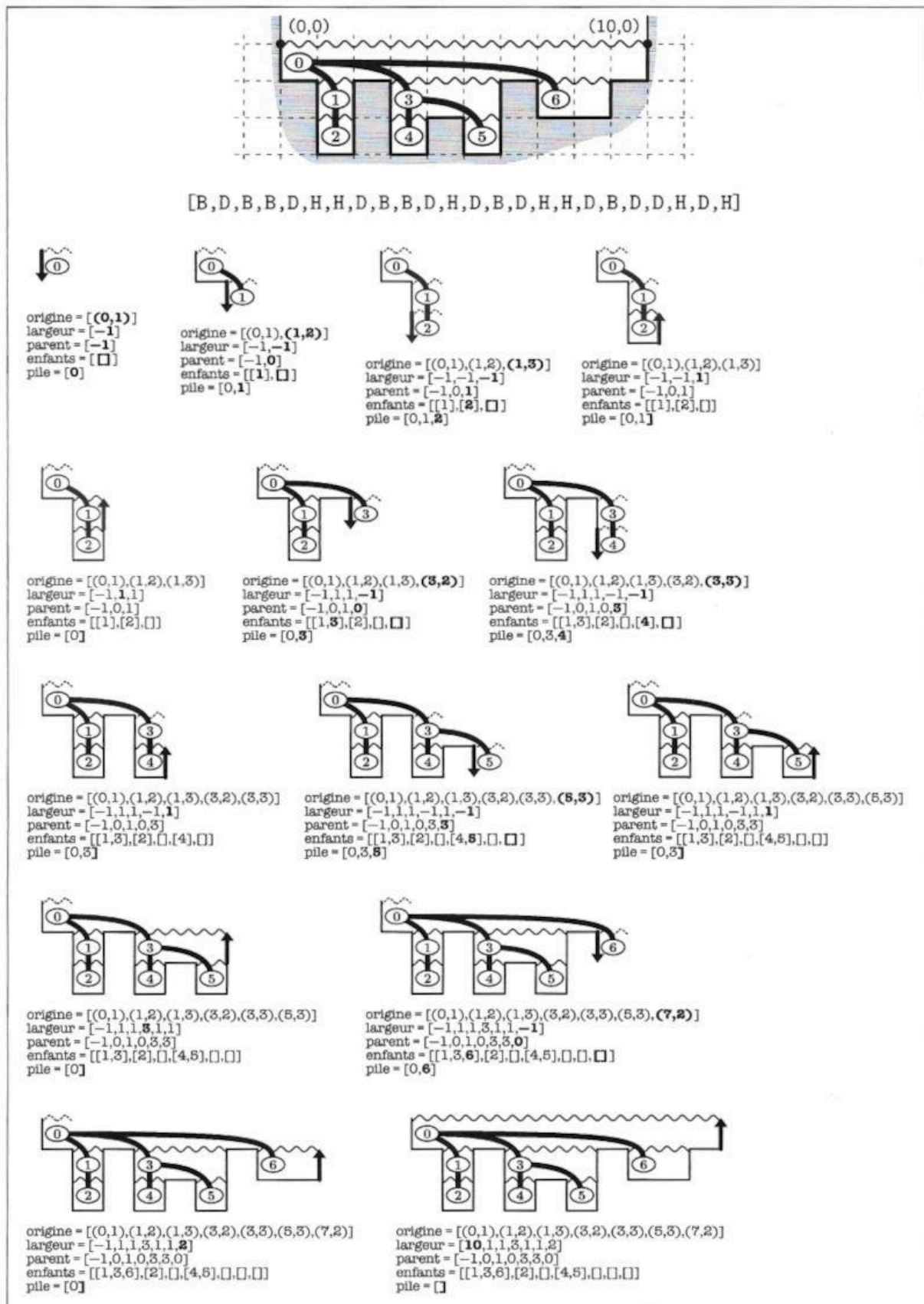


Fig. 5. – Une exécution de l’algorithme de décomposition en 7 rectangles de la grotte la ciel ouvert en haut : on peut suivre les modifications **en gras** de la représentation de chaque tableau et de la pile.

**Implémentation d'une structure de pile.** Pour pouvoir implémenter l'algorithme décrit ci-dessus, nous allons avoir besoin d'une structure de pile. Le code fourni contient un début d'implémentation de pile. On représente un pile avec la structure suivante :

```
typedef struct {
    int* contenu; /* tableau contenant les éléments */
    int capacite; /* le nombre d'éléments maximal que la pile peut stocker */
    int taille; /* le nombre d'éléments dans la pile : initialement 0 */
} pile;
```

- Le champ **capacite** représente le nombre maximal d'éléments que la pile peut contenir. Si on tente d'empiler à une pile pleine, une erreur est provoquée.
- Le champ **contenu** contient les éléments de la pile. Il s'agit d'un tableau de taille **capacite**.
- Le champ **taille** représente le nombre d'éléments actuellement dans la pile. Ce champ vaut 0 initialement, et ne peut jamais dépasser **capacite**. Seules les **taille** premières cases du tableau **contenu** représentent les éléments de la pile.

On souhaite maintenant implémenter les opérations suivantes sur la structure de pile :

- `pile* creer_pile (int capacite);`
- `void liberer_pile(pile* p)`
- `bool est_vide(pile* p);`
- `void empiler(pile* p, int v);`
- `int depiler(pile* p);`
- `int sommet(pile* p);`

**Question 13.** Certaines de ces opérations sont déjà implémentées, d'autres sont à compléter. Compléter le code aux endroits indiqués. **Tester votre implémentation à l'aide de la fonction `test_pile()` fournie avant de passer à la suite.**

**Construction de la hiérarchie.** On peut maintenant implémenter l'algorithme de décomposition.

**Question 14.** Compléter la fonction :

`hierarchie construire_hierarchie(direction profil[], int taille_profil)` qui implémente l'algorithme de décomposition à partir du **profil** de taille **taille\_profil**, et renvoie la hiérarchie ainsi construite.

## II. B. Remplissage de la grotte

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 Fig. 6, la source (symbolisée par le symbole ▲) est placée soit à l'origine (figure de gauche), soit à l'origine du rectangle au milieu (figure de droite).

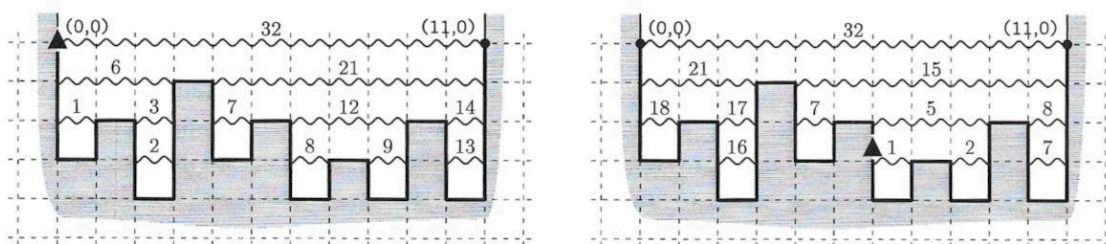


Fig. 6. – Les dates et ordre de remplissage dépendent de la position de la source (symbolisée par ▲)

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 Fig. 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 15.** Compléter la fonction `int* ordre_remplissage_depuis_origine(hierarchie h)` qui prend en paramètre une structure hiérarchique `h`, et qui initialise et renvoie un tableau `ordre` contenant le numéro des rectangles dans l'ordre dans lequel ils se remplissent. Autrement dit, `ordre[0]` contient le premier rectangle à remplir, `ordre[1]` le second, etc.

*Indication : cette question consiste à implémenter un parcours d'arbre particulier, que l'on pourra réaliser à l'aide d'une fonction auxiliaire.*

**Question 16.** Compléter la fonction `float* hauteurs_eau_depuis_origine(hierarchie h, float t)` qui prend en paramètres une structure hiérarchique `h` et un temps `t` exprimé par un nombre flottant. Cette fonction initialise et renvoie un tableau de flottant `hauteur` de manière à ce que `hauteur[i]` soit la hauteur d'eau dans le rectangle d'indice `i` à l'instant `t`. La hauteur sera donc un flottant entre `0.0` et `1.0` sauf pour le dernier rectangle qui est infini et peut donc être rempli à une hauteur arbitrairement grande.

**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 Fig. 6 à droite entre les dates  $t = 5$  et  $t = 7$ .

**Question 17.** Expliquer (en commentaires) pourquoi jamais plus de deux rectangles ne se rempliront simultanément. Votre réponse ne devrait pas excéder quelques phrases.

**Question 18.** Compléter la fonction `int* volumes_totaux(hierarchie h)` qui prend en paramètre une structure hiérarchique `h`, et qui initialise et renvoie un tableau `volume` tel que `volume[i]` est la somme des volumes des rectangles descendants du rectangle d'indice `i`, `i` inclus.

**Question 19 : Bonus.** Cette question est difficile et est facultative. Elle n'est de plus pas très guidée, vous pouvez suivre l'approche que vous souhaitez.

Écrire une fonction `float* hauteurs_eau_depuis_source(hierarchie h, float t, int source)` qui prend en paramètres une structure hiérarchique `h`, un temps `t` et l'indice du rectangle où est située la source. Comme à la question 17, cette fonction doit renvoyer un tableau `hauteur` de manière à ce que `hauteur[i]` soit la hauteur d'eau dans le rectangle d'indice `i` à l'instant `t`.

*Indication : la fonction `volumes_totaux` pourrait être utile !*