TP #14 — Stacks & queues; valores menores más cercanos; größtes Rechteck

Exercice 1. *Stacks & queues*

Pile fonctionnelle

On rappelle les spécifications de la structure de données de *pile fonctionnelle*. Une telle structure doit posséder :

- une valeur immuable de pile *vide* (ne possédant aucun élément) ;
- un constructeur *push* qui quand appelé avec une valeur x et une pile s construit une *nouvelle* pile où x a été *empilé* en sommet (début) de s ;
- une fonction *pop* qui quand appelée sur une pile s non vide, renvoie le couple constitué du sommet de s et d'une nouvelle pile égale à s dont on a supprimé le sommet. Autrement dit, une pile implémente une politique ¨*last in, first out*¨ (LIFO).

1. Implémentez une telle structure de données en OCaml, en utilisant des `'a list`. Vous devez *a minima* écrire deux fonctions :
   - `stack_push : 'a -> 'a list -> 'a list`
   - `stack_pop : 'a list -> 'a * 'a list`

   mais pouvez en ajouter d'autres (ou des variantes).
   Réfléchissez bien à comment traiter explicitement les éventuelles erreurs, notamment dans `stack_pop`.
2. Quels sont les coûts de vos fonctions (en fonction du nombre d'éléments présents dans la pile) ?
3. Testez. Vérifiez notamment l'immuabilité de votre implémentation.

File fonctionnelle

On donne les spécifications de la structure de données de *file fonctionnelle*. Une telle structure doit posséder :

- une valeur immuable de file *vide* (ne possédant aucun élément) ;
- un constructeur *push* qui quand appelé avec une valeurx x et une file q construit une *nouvelle* file où x a été *enfilé* en queue (fin) de q ;
- une fonction *pop* qui quand appelée sur une file q non vide, renvoie le couple constitué de la tête (début) de q et d'une nouvelle file égale à q dont on a supprimé la tête. Autrement dit, une file implémente une politique ¨*first in, first out*¨ (FIFO).

4. Implémentez (naïvement) une telle structure de données en OCaml, en utilisant des `'a list`. Vous devez *a minima* écrire deux fonctions :
   — `queue_push : 'a -> 'a list -> 'a list`
   — `queue_pop : 'a list -> 'a * 'a list`

   mais pouvez en ajouter d'autres (ou des variantes).
   Réfléchissez bien à comment traiter explicitement les éventuelles erreurs, notamment dans `queue_pop`.
5. Quels sont les coûts de vos fonctions (en fonction du nombre d'éléments présents dans la pile) ?
6. Testez. Vérifiez notamment l'immuabilité de votre implémentation, et la différence avec une pile.

Pile impérative

On donne les spécifications de la structure de données de *pile impérative*. Une telle structure doit posséder *a minima*:

— un constructeur de pile vide (éventuellement d'une certaine *capacité*, c'est à dire pouvant stocker au maximum un certain nombre d'éléments) ;
— une fonction *is_empty* qui teste si son argument est une pile vide ;
— un transformateur *push* qui quand appelé avec une valeur x et une pile s non pleine modifie s pour y ajouter x en son sommet ;
— un transformateur *pop* qui quand appelé avec une pile s non vide en supprime le sommet et le renvoie.

On souhaite implémenter en C une telle structure de donnée (dans une version à capacité fixe) en utilisant un tableau assorti d'un pointeur indiquant le sommet de la pile (ou plus précisément, l'indice dans le tableau où un éventuel prochain élément devra être empilé). Sommairement, une pile vide est un tableau vide avec un pointeur à zéro ; empiler consiste à ajouter l'élément à l'indice du tableau donné par le pointeur et incrémenter ce dernier ; dépiler consiste à décrémenter le pointeur et renvoyer l'élément se trouvant à cet indice.
On commence par définir le type :

```
struct stack
{
    size_t capacity;
    size_t top;
    int *data;
};
```

qui permettra de réaliser l'approche ci-dessus pour une pile contenant des `int`.

7. Écrivez une fonction C de signature :
   `struct stack *make(size_t capacity)`

qui crée une pile vide de capacité `capacity`, représentée par un objet de durée de stockage « allouée » dont elle renvoie l'adresse.

8. Écrivez une fonction C de signature :
   ```
   void destroy(struct stack *sp)
   ```
   qui détruit (libère la mémoire de) la pile (de durée de vie « allouée ») dont l'adresse est fournie en argument.
9. Testez vos fonctions (notamment pour l'absence d'accès illégaux et de fuite mémoire).
10. Écrivez une fonction C de signature :
    ```
    bool is_empty(struct stack *sp)
    ```
    qui renvoie `true` si la pile d'adresse fournie en argument est vide (ne contient aucun élément) et `false` sinon.
11. Écrivez une fonction C de signature :
    ```
    void push(struct stack *sp, int x)
    ```
    qui empile `x` dans la pile dont l'adresse est stockée dans `sp`. Vous pouvez traiter le cas d'une pile pleine à votre convenance.
12. De même pour une fonction
    ```
    int pop(struct stack *sp)
    ```
13. Quels sont les coûts de vos fonctions (en fonction du nombre d'éléments dans la pile) ?
14. Écrivez une fonction C de signature :
    ```
    void empty_and_print(struct stack *sp)
    ```
    qui appelle `pop` sur la pile d'adresse *sp* (et affiche l'élément ainsi supprimé) tant qu'elle n'est pas vide. Cette fonction devra uniquement utiliser les fonctions `is_empty` et `pop`, et ne pas manipuler la « représentation interne » de `struct stack`. (Ceci permettrait par exemple de la réutiliser *telle quelle* pour une *autre* implémentation de pile (utilisant les mêmes noms de type & de fonctions).)
15. Testez.

File impérative

On donne les spécifications de la structure de données de *file impérative*. Une telle structure doit posséder *a minima*:

— un constructeur de file vide (éventuellement d'une certaine *capacité*, c'est à dire pouvant stocker au maximum un certain nombre d'éléments) ;
— une fonction *is_empty* qui teste si son argument est une file vide ;
— un transformateur *push* qui quand appelé avec une valeur x et une file q non pleine modifie q pour y ajouter x en sa queue ;
— un transformateur *pop* qui quand appelé avec une file q non vide en supprime la tête et la renvoie.

On souhaite implémenter en C une telle structure de donnée (dans une version à *nombre total d'éléments enfilés* fixe) en utilisant une variante primitive de celle précédemment employée pour implémenter une pile : le tableau est maintenant assorti de *deux* pointeurs, l'un indiquant la queue de la file (ou plus précisément, l'indice dans le tableau où un éventuel prochain élément devra être enfilé) et l'autre la tête (ou plus précisément, l'indice dans le tableau d'un éventuel prochain élément à défiler (ou pour des raisons pratiques, ce même indice incrémenté de un)). Ainsi, enfiler un élément consiste à ajouter l'élément à l'indice donné par le pointeur de queue et incrémenter ce dernier, et défiler un élément consiste à renvoyer l'élément à l'indice donné par le pointeur de tête et incrémenter ce dernier.
On commence par définir le type :

```
struct queue
{
    size_t capacity;
    size_t head;
    size_t tail;
    int *data;
};
```

qui permettra de réaliser l'approche ci-dessus pour une file contenant des `int`.

16. Écrivez une fonction C de signature :

```
struct queue *make(size_t capacity)
```

qui crée une file vide de capacité (nombre total d'éléments pouvant être enfilés) `capacity`, représentée par un objet de durée de stockage « allouée » dont elle renvoie l'adresse.

17. Écrivez une fonction C de signature :

```
void destroy(struct queue *qp)
```

qui détruit (libère la mémoire de) la file (de durée de vie « allouée ») dont l'adresse est fournie en argument.

18. Testez vos fonctions (notamment pour l'absence d'accès illégaux et de fuite mémoire).

19. Écrivez une fonction C de signature :

```
bool is_empty(struct queue *qp)
```

qui renvoie `true` si la file d'adresse fournie en argument est vide (ne contient aucun élément) et `false` sinon.

20. Écrivez une fonction C de signature :

```
void push(struct queue *qp, int x)
```

qui enfile `x` dans la file dont l'adresse est stockée dans `qp`. Vous pouvez traiter le cas d'une file pleine à votre convenance.

21. De même pour une fonction

```
int pop(struct queue *qp)
```

22. Quels sont les coûts de vos fonctions (en fonction du nombre d'éléments dans la file) ?

23. Écrivez une fonction C de signature :

```
void empty_and_print(struct queue *qp)
```

qui appelle `pop` sur la file d'adresse *qp* (et affiche l'élément ainsi supprimé) tans qu'elle n'est pas vide. Cette fonction devra uniquement utiliser les fonctions `is_empty` et `pop`, et ne pas manipuler la « représentation interne » de `struct queue`.

24. Testez.

Exercice 2. *Valores menores más cercanos*

En OCaml pour commencer

Dans tout ce qui suit, pour toute `'a list` ℓ, on utilise la notation $\ell[i]$ pour désigner le *i-ème* élément de ℓ en partant de sa tête (s'il existe). (Bien entendu, cela ne veut pas dire qu'une telle syntaxe a été magiquement ajoutée en OCaml…)

Soit une liste d'entiers ℓ (que l'on représentera par une `int list`), on définit pour tout indice i d'un élément de ℓ les ensembles $s_i := \{j \mid 0 < j < i \wedge \ell[j] < \ell[i]\}$. On souhaite alors résoudre le problème suivant : on veut construire une liste ℓ' d'options sur des entiers (que l'on représentera par une `int option list`) telle que $\ell'[i]$ vaut `Some k` avec $k := \ell[\max\ s_i]$ si $s_i \neq \emptyset$, et `None` sinon. Autrement dit, ℓ' contient à chaque position i la valeur du plus proche élément précédent $\ell[i]$ qui lui est inférieur, si un tel élément existe.
Par exemple, on a :

- ℓ = `[4; 3; 2; 1]` ; ℓ' = `[None; None; None; None]`
- ℓ = `[1; 2; 3; 4]` ; ℓ' = `[None; Some 1; Some 2; Some 3]`
- ℓ = `[0; 2; 4; 1; 5; 7; 6; 5; 2]` ;
  ℓ' =
  ```
  [None; Some 0; Some 2; Some 0; Some 1; Some 5; Some 5;
   Some 1; Some 1]
  ```

1. Sans l'implémenter, décrivez un algorithme naïf permettant de résoudre ce problème, et donnez son coût en fonction de la longueur de son argument.

Ce problème peut être résolu plus efficacement que par une approche naïve en utilisant une pile : on parcourt ℓ une unique fois (de la tête vers la queue) en maintenant une pile des valeurs déjà rencontrées ; pour obtenir la valeur de $\ell'[i]$ il suffit de dépiler la pile jusqu'à y trouver une éventuelle valeur qui lui est inférieur, puis on empile $\ell[i]$ avant de passer à l'indice suivant.

2. Écrivez une fonction OCaml `vmmc : 'a list -> 'a option list` qui utilise l'algorithme décrit ci-dessus pour résoudre le problème posé.

Il pourra être utile de définir une ou plusieurs fonctions intermédiaires, par exemple une fonction `drop_while` telle que celle disponible dans le module `List`. N'oubliez pas non plus l'existence d'une fonction `List.rev`

3. Analysez le coût de votre fonction en fonction de la longueur de son argument. S'il n'est pas linéaire, cela veut dire que votre implémentation est trop inefficace ou votre analyse trop imprécise (dans ce dernier cas, il peut être utile d'essayer de borner *globalement* le nombre des opérations effectuées sur la pile).
4. Testez.

On considère maintenant une façon alternative d'encoder le résultat de l'algorithme ; avec les mêmes notations que précédemment, on a :

$$\ell''[i] = \min_{0 \leq j \leq i} j,\ \forall k \in [\![j, i]\!], \ell[k] \geq \ell[i]$$

Autrement dit $\ell''[i]$ est le plus petit indice tel que tous les éléments de ℓ entre cet indice et i sont supérieurs ou égaux à $\ell[i]$, ou de façon équivalente c'est un plus l'indice de l'élément donnant sa valeur à $\ell'[i]$ dans la version précédente du problème (en considérant qu'un élément n'apparaissant pas dans ℓ a -1 pour indice).
Par exemple, on a :

— $\ell =$ `[4; 3; 2; 1]` ; $\ell'' =$ `[0; 0; 0; 0]`
— $\ell =$ `[1; 2; 3; 4]` ; $\ell'' =$ `[0; 1; 2; 3]`
— $\ell =$ `[0; 2; 4; 1; 5; 7; 6; 5; 2]` ;
$\ell'' =$
`[0; 1; 2; 1; 4; 5; 5; 4; 4]`

5. Écrivez une fonction OCaml `vmmc2 : 'a list -> int list` qui résout cette nouvelle variante du problème.
6. Testez.

En C maintenant

On souhaite maintenant résoudre le même problème (dans sa seconde version) lorsque les données sont représentées par un tableau d'`int` en C. Nous pouvons alors exploiter le fait qu'il est désormais efficace d'accéder à un élément du tableau d'indice quelconque afin de représenter implicitement la pile utilisée par l'algorithme. (Un peu) plus précisément, soit a le tableau en entrée et a' un tableau dans lequel l'algorithme écrit son résultat, l'état de la pile juste avant de traiter l'élément de a d'indice i est donné (en partant du sommet) par l'élément de a d'indice $i-1$, puis celui d'indice $a'[i-1]$, puis celui d'indice $a'[a'[i-1]]$ etc.

7. Écrivez une fonction C de signature :
```
size_t *vmmc2(size_t n, int a[n])
```

qui implémente l'algorithme esquissé ci-dessus pour résoudre le problème posé. Cette fonction ne devra user d'aucune récursion, et le résultat devra être renvoyé sous la forme d'un tableau de même longueur que l'argument `a`, de durée de stockage « allouée ».

8. Testez.

Exercise 3. *Größtes Rechteck* (adapté de XENS Info. B 2014)
Une application de l'algorithme de l'exercice précédent est à la résolution efficace du problème du *plus grand rectangle* (inscrit). Soit un tableau bidimensionnel de valeurs 0 ou 1 (représentant par exemple une image en noir et blanc), ce problème consiste à rechercher le plus grand rectangle inscrit (ou sous-rectangle) tout à 0. Ce problème (pour un tableau initial carré) était celui que se proposait d'étudier l'épreuve de composition d'informatique B (PC & MP hors spécialité informatique) du concours d'entrée 2014 à l'École polytechnique / ENS Cachan / ESPCI. On en redonne l'essentiel ci-dessous (à peu près tel que rédigé à l'origine), d'où l'on a juste supprimé les questions relatives à l'algorithme de l'exercice précédent (déjà traitées ci-dessus, ou qui seront à traiter en TD). (Bien entendu, vous devez également tester toutes vos fonctions, directement ou non.)

Recherche unidimensionnelle

Dans cette partie, nous allons étudier le problème de reconnaissance de forme sur des tableaux unidimensionnels. Nous supposerons donnés un entier n et un tableau *tab* de taille n représenté par un tableau d'`int` dont les cases contiennent 0 ou 1. Le but de cette partie est de trouver le nombre maximal de 0 contigus (c'est à dire figurant dans des cases consécutives) dans le tableau.
Dans cette partie nous utiliserons le tableau suivant comme exemple :

```
//    i      : 0  1  2  3  4  5  6  7  8  9  A  B  C
int tab[13] = {0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1};
```

1. Écrire une fonction C de signature :
   `size_t nombreZerosDroite(size_t n, int tab[n], size_t i)`
   prenant en paramètres le tableau *tab* de taille n ainsi qu'un indice i compris entre 0 et $n-1$. Cette fonction devra renvoyer le nombre de cases contiguës du tableau contenant 0 à droite de la case d'indice i, cette case étant comprise. D'un point de vue formel il s'agit de renvoyer 0 si la case d'indice i contient un 1 et sinon de renvoyer :

   $$\max\{j-i+1 \text{ tel que } i \leq j \leq n \text{ et } \forall k \in [\![i,j]\!],\ tab[k]=0\}$$

   Sur le tableau exemple précédent, en prenant comme indice i $=3$ nous obtenons 0 et pour l'indice i $=5$ votre fonction devra retourner 2. Notez que si la case i contient 1, alors votre fonction devra retourner 0.

Il est maintenant possible de réaliser un algorithme de complexité optimale utilisant la fonction précédente. En voici une brève description. Parcourons le tableau de gauche à droite et à chaque début de plage de 0 contigus, nous calculons la taille de cette plage puis repartons juste après elle. Ayant ainsi calculé la taille de toutes les plages de 0 il suffit de retenir celle de taille maximale. Sur l'exemple précédent, on part de l'indice 0 qui est un début de plage de 0 de taille 2. Nous continuons donc à chercher le prochain début de plage à partir de l'indice $0+2=2$. On examine ensuite l'indice 3 qui contient encore un 1 puis arrivons à l'indice 4 qui est le début d'une plage de 0 de taille 3. Nous allons donc chercher le prochaine début de plage à partir de l'indice $4+3=7$ etc.

2. Écrire une fonction C de signature :
`size_t nombreZerosMaximum(size_t n, int tab[n])`
qui renvoie le nombre maximal de 0 contigus en utilisant l'algorithme précédent. On attachera une attention particulière à ce que l'algorithme produit soit de complexité linéaire c'est à dire en temps $O(n)$.

De la 1D vers la 2D

Cette partie consiste à développer un algorithme efficace pour détecter un rectangle d'aire maximale rempli de 0 dans un tableau bidimensionnel. Par mesure de simplification, nous travaillerons sur des tableaux carrés et nous supposerons donné un tableau *tab2* de taille $n \times n$. Un tel tableau bidimensionnel sera représenté par un tableau d'`int` à deux dimensions, avec `tab2[i][j]` l'élément de ligne i et colonne j.
Nous utiliserons le tableau suivant comme exemple :

```
//              j : 0  1  2  3  4  5  6  7      i
int tab2[8][8] = {{0, 0, 1, 1, 0, 0, 0, 1}, // 0
                  {0, 0, 0, 0, 1, 0, 0, 1}, // 1
                  {1, 0, 0, 0, 0, 0, 0, 1}, // 2
                  {0, 1, 0, 0, 0, 0, 0, 1}, // 3
                  {0, 0, 1, 1, 0, 0, 0, 1}, // 4
                  {0, 0, 1, 0, 0, 0, 0, 1}, // 5
                  {0, 1, 1, 1, 0, 1, 0, 1}, // 6
                  {0, 0, 1, 1, 0, 0, 0, 1}};// 7
```

Méthode naïve. La première méthode proposée consiste à prendre une cellule (i, j) et à trouver un rectangle d'aire maximale dont le coin inférieur gauche est cette cellule. Remarquez que suivant l'orientation donnée dans le tableau exemple ci-dessus, un rectangle d'aire maximale de coin inférieur gauche la cellule (i, j) aura comme coin supérieur droit la cellule (i', j') avec $i' \leq i$ et $j' \geq j$. Par exemple,

un rectangle d'aire maximale de coin inférieur gauche la cellule $(i, j) = (3, 2)$ aura comme coin supérieur droit la cellule de coordonnées $(2, 6)$.

3. Écrire une fonction C de signature
   ```
   size_t
   rectangleHautDroit
   (size_t n, int tab2[n][n], size_t i, size_t j)
   ```
   qui renvoie l'aire d'un rectangle d'aire maximale rempli de 0 et de coin inférieur gauche (i, j). **On cherchera la solution la plus efficace possible.** Pour $i = 3$ et $j = 2$ sur le tableau exemple ci-dessus, la fonction devra renvoyer la valeur 10.
4. Expliquer comment trouver naïvement un rectangle d'aire maximale rempli de 0 en utilisant la fonction `rectangleHautDroit`. Quelle serait la complexité de cette approche en fonction de n ?

Un peu de précalcul. Notre algorithme parcourt de nombreuses fois les même cases afin de vérifier la présence d'un 0 ou d'un 1. Nous allons donc effectuer avant notre algorithme un précalcul de certaines valeurs ce qui nous permettra, dans la prochaine partie, d'accélérer les fonctions précédentes.
Pour chaque cellule (i, j), nous allons calculer le nombre $c_{i,j}$ de cellules contiguës contenant 0 au dessus de la cellule (i, j), cette dernière étant comprise. Ce nombre est tel que les cellules (i, j), $(i-1, j)$, …, $(i-c_{i,j}+1, j)$ contiennent 0 et la cellule $(i-c_{i,j}, j)$ contient 1 ou bien cette cellule n'existe pas. Ces valeurs $c_{i,j}$ seront rangées dans un tableau *col* représenté par un pointeur de pointeur de `size_t`. Le tableau col suivant est obtenu à partir du tableau exemple (où l'on utilise ici une représentation de `col` sous la forme d'un tableau à deux dimensions, ce qui n'est **pas** ce que votre fonction doit renvoyer).

```
//                  j : 0  1  2  3  4  5  6  7      i
size_t col[8][8]  = {{1, 1, 0, 0, 1, 1, 1, 0}, // 0
                     {2, 2, 1, 1, 0, 2, 2, 0}, // 1
                     {0, 3, 2, 2, 1, 3, 3, 0}, // 2
                     {1, 0, 3, 3, 2, 4, 4, 0}, // 3
                     {2, 1, 0, 0, 3, 5, 5, 0}, // 4
                     {3, 2, 0, 1, 4, 6, 6, 0}, // 5
                     {4, 0, 0, 0, 5, 0, 7, 0}, // 6
                     {5, 1, 0, 0, 6, 1, 8, 0}};// 7
```

5. Écrire une fonction C de signature :
   ```
   size_t **colonneZeros(size_t n, int tab2[n][n])
   ```
   qui renvoie un pointeur de pointeur de `size_t` contenant l'adresse d'un « tableau » bidimensionnel de durée de stockage « allouée » de mêmes dimensions que `tab2`, et tel que `col[i][j]` donne le nombre de cellules

contiguës contenant 0 au dessus de (i, j), cette cellule étant comprise. **On apportera un soin particulier à programmer la fonction la plus rapide possible.**

6. Quelle est la complexité de votre fonction `colonneZeros` ?

Algorithme optimisé

Rectangle d'aire maximale dans un histogramme. Une nouvelle étape dans notre algorithme réside dans la résolution du problème du calcul d'un rectangle d'aire maximale inscrit dans un histogramme. Nous supposerons donné un histogramme c'est à dire un tableau *histo* de taille n contenant des entiers. Chaque valeur représentera la hauteur d'une barre de l'histogramme. Par exemple le tableau suivant est associé à l'histogramme dessiné ci-dessous.

```
//              i : 0  1  2  3  4  5  6  7
size_t histo[8] = {3, 1, 2, 4, 2, 2, 0, 1};
```

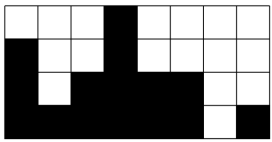

FIGURE 1 – Représentation graphique de l'histogramme `histo`

L'idée est de calculer un indice `L[i]` pour chaque colonne i, tel que `L[i]` est le plus petit entier j satisfaisant : $0 \leq j \leq i$ et `histo[k]` $\geq$ `histo[i]`, pour tout k tel que $j \leq k \leq i$.
Notons que $0 \leq$ `L[i]` $\leq i$. Dans notre exemple nous aurons `L[2] = 2` car la colonne 1 est strictement plus petite que la colonne 2. Nous aurons par ailleurs `L[5] = 2` car `histo[2]` $\geq$ `histo[5]`, `histo[3]` $\geq$ `histo[5]`, `histo[4]` $\geq$ `histo[5]` mais `histo[1]` $<$ `histo[5]`.
Nous pourrions calculer les valeurs `L[i]` de manière naïve mais cela ne permettrait pas d'améliorer notre algorithme. Nous allons donc procéder autrement. [*Cf. exercice précédent.*]

7. Écrivez une fonction C de signature :
   ```
   size_t *calculeL(size_t n, size_t histo[n])
   ```
   qui renvoie en temps $O(n)$ (l'adresse d'un objet de durée de stockage « allouée » représentant) le tableau `L` de taille n tel que `L[i]` est l'indice défini précédemment. Écrivez de même une fonction C de signature :
   ```
   size_t *calculeR(size_t n, size_t histo[n])
   ```
   qui renvoie en temps $O(n)$ (l'adresse d'un objet de durée de stockage « allouée » représentant) un tableau `R` de valeurs `R[i]` $\geq$ i qui de manière analogue est

l'indice maximal tel que `histo[k]` $\geq$ `histo[i]` pour tout k tel que $i \leq k \leq$ `R[i]`.

8. Justifier que pour tout i, le rectangle commençant à l'indice `L[i]`, terminant à l'indice `R[i]` et de hauteur `histo[i]` est inscrit dans l'histogramme.
9. Soit un rectangle d'aire maximale inscrit dans l'histogramme. Supposons que ce rectangle commence à l'indice ℓ, termine à l'indice r, et a pour hauteur h. Montrer qu'il existe $i \in [\![\ell, r]\!]$ tel que `histo[i]` $= h$, `L[i]` $= \ell$, et `R[i]` $= r$.
10. En déduire une fonction C de signature :

    ```
    size_t
    plusGrandRectangleHistogramme(size_t n, size_t histo[n])
    ```

    qui calcule et renvoie l'aire d'un plus grand rectangle inscrit dans l'histogramme. Donner sa complexité.

Conclusion

En revenant au problème initial du calcul d'un rectangle de 0 d'aire maximale dans une matrice en deux dimensions, remarquer que chaque ligne du tableau *col* calculé par la fonction `colonneZeros` de la question 5 peut être interprétée comme un histogramme.
En utilisant cette analogie, il est possible de proposer une méthode efficace de résolution du problème.

11. Écrire la fonction C de signature :

    ```
    size_t rectangleToutZero(size_t n, int tab2[n][n])
    ```

    qui calcule un rectangle d'aire maximale rempli de 0 dans le tableau `tab2` et en renvoie son aire.
12. Quelle est la complexité de votre fonction ?
13. Modifiez votre fonction afin qu'elle renvoie les coordonnées (i, j) d'un coin d'un rectangle atteignant l'aire maximale.