

Ce devoir est composé de 6 exercices indépendants. Pour répondre à une question, il est permis de réutiliser une fonction demandée à une question précédente, même sans avoir réussi à coder cette fonction. Lorsque la complexité d'une fonction sera demandée, il s'agira toujours de la complexité temporelle dans le pire des cas, et on souhaitera établir un \mathcal{O} le plus petit possible.

Exercice 1 (Commandes et système de fichiers) :

En plus des commandes `chmod`, `cd`, `ls`, `mkdir`, `ln` vues en cours, on donne les commandes suivantes :

- `cp chemin1/fichier1 chemin2/fichier2` copie le premier fichier vers le deuxième ;
- `mv chemin1/fichier1 chemin2/fichier2` déplace le premier fichier vers le deuxième ;
- `rm chemin/fichier` supprime le fichier concerné.

On suppose disposer d'un répertoire **Exemple/** contenant l'arborescence ci-contre (les entrées terminant par un `/` sont des dossiers, les autres sont des fichiers).

```

Répertoire
Exemple/
|-- Autres/
|   |-- vieux_document.pdf
|-- Documents/
|   |-- Point_de_depart/
|       |-- donnees.txt
|-- MP2I/
|   |-- Info/
|       |-- Concours_Blanc/
|           |-- TP_exemple/
|               |-- tp.ml
|-- Maths/
|-- Physique/

```

- Dans cette question, on suppose que le dossier courant est le dossier **Point_de_depart/**.
 - Donner la commande permettant d'afficher les permissions accordées pour le fichier `donnees.txt`.
 - Donner la commande permettant de retirer les droits d'accès en écriture pour le fichier `donnees.txt`.
 - Donner la commande permettant de supprimer le fichier `vieux_document.pdf`.
 - Donner la commande permettant de se déplacer dans le dossier **Physique/**.
- Dans cette question, on suppose que le dossier courant est le dossier **Physique/**.
 - Donner la commande permettant de créer un dossier **Data/** dans le dossier courant.
 - Donner la commande permettant de faire une copie du fichier `donnees.txt` dans le dossier **Data/** (le nouveau fichier devra s'appeler `donnees_bis.txt`).
 - Donner la commande permettant de rajouter les droits d'accès en écriture pour le fichier `donnees_bis.txt`.
 - Donner la commande permettant de se déplacer dans le dossier **Concours_Blanc/**.
- Dans cette question, on suppose que le dossier courant est le dossier **Concours_Blanc/**.
 - Donner la commande permettant de déplacer le fichier `tp.ml` dans le dossier courant.
 - Donner la commande permettant de créer un lien symbolique du fichier `donnees_bis.txt` vers un nouveau fichier dans le dossier courant (ce nouveau fichier devra s'appeler `donnees.txt`).

Exercice 2 (Algorithme de tri) :

Notation : dans cet exercice, on introduit une notation pour faciliter nos raisonnements sur les tableaux : si `t` est un tableau, et si `i` et `j` sont deux entiers, on note `t[i:j]` le sous-tableau contenant les valeurs : `t.(i)`, `t.(i+1)`, ..., `t.(j-1)`. En particulier, si `t` est de longueur `n`, `t[0:n]` désigne `t`.

On considère la fonction `tri_gnome` ci-contre.

- Justifier que, pour toute entrée, la fonction `tri_gnome` ne rencontrera jamais d'erreur d'indice.

```

let tri_gnome t =
  let n = Array.length t in
  let i = ref 0 in
  while !i < n do
    if !i = 0 || t.(!i) >= t.(!i-1) then
      incr i
    else
      begin
        let tmp = t.(!i) in
        t.(!i) <- t.(!i-1) ;
        t.(!i-1) <- tmp ;
        decr i
      end
    end
  done

```

2. Montrer que la boucle **while** de la fonction `tri_gnome` admet l'invariant de boucle suivant :
- Si i_{\max} (resp. i) désigne la valeur maximale (resp. courante) stockée dans la référence i , alors $t[0:i]$ et $t[i:i_{\max}]$ sont deux sous-tableaux de t triés dans l'ordre croissant, et les éléments de $t[0:i+1]$ sont plus petits que les éléments de $t[i+1:i_{\max}]$.
3. En déduire :
- que la fonction `tri_gnome` termine sur toute entrée;
 - la complexité de `tri_gnome t` en fonction de la taille n de t ;
 - que la fonction `tri_gnome` est correcte : pour tout tableau t , `tri_gnome t` modifie le tableau t pour qu'il contienne les mêmes éléments qu'au départ mais triés dans l'ordre croissant.

Exercice 3 (Fonction mystère) :

On considère la fonction ci-contre.

- Démontrer la terminaison de f .
- Pour $n \in \mathbb{N}$, conjecturer une relation entre $f\ n$ et n .
En déduire une spécification de f .
- Démontrer la correction de f relativement à cette spécification.
- Déterminer la complexité de f .

```
let f n =
  let s = ref 1 in
  let c = ref 0 in
  while !s <= n do
    incr c ;
    s := !s + 2 * !c + 1
  done ;
  !c
```

Exercice 4 (Fonction de Hofstadter) :

On considère la fonction de Hofstadter ci-dessous, dont on souhaite étudier la terminaison.

- Vérifier que h termine sur 0, 1, 2, 3, 4 et donner les valeurs renvoyées.
- Démontrer que h termine sur tout entier naturel.

```
let rec h n = match n with
| 0 -> 0
| _ -> n - h(h(n-1))
```

Indication : on pourra montrer que $\forall n \in \mathbb{N}$, $h\ n$ termine et de plus renvoie une valeur appartenant à un intervalle dépendant de n à déterminer.

Exercice 5 (Problème de l'arrêt) :

Si f est une fonction, et x un argument de f , on dit que f **termine** sur x si l'appel $f\ x$ renvoie un résultat au bout d'un certain temps, ou est interrompu à la suite d'une erreur (qui va lever une exception), par exemple une division par zéro ou une erreur d'indice dans un tableau.

```
let rec f1 n = match n with
| 0 -> 0
| _ -> n + f1 (n-2)
```

```
let f2 n =
  let i = ref n in
  while !i <> 10 do
    incr i
  done
```

- Déterminer les entiers $n \in \mathbb{Z}$ pour lesquels la fonction $f1$ ci-dessus termine, et démontrer la terminaison de cette fonction sur cet ensemble d'entiers.
- Même question pour la fonction $f2$ ci-dessus.
- Écrire une fonction `infini` qui ne termine sur aucun entier.
- L'objectif de cette question est d'examiner la possibilité de décider automatiquement (via un algorithme) si une fonction f termine sur une entrée x , en analysant le code de f .

On suppose l'existence d'une fonction curryfiée `arret` prenant 2 arguments :

- une chaîne de caractère `code_f` correspondant au code d'une fonction f ;
- un objet de type quelconque x ;

et qui renvoie `true` si f termine sur x , et `false` sinon.

- Donner le type de la fonction `arret`.
- Écrire une fonction `bizarre` prenant en argument une chaîne de caractère `code_f` correspondant au code d'une fonction f , ainsi qu'un objet de type quelconque x , telle que `bizarre code_f x` termine si et seulement si f **ne** termine **pas** sur x .
Le résultat renvoyé par `bizarre` lorsqu'elle termine n'a aucune importance.

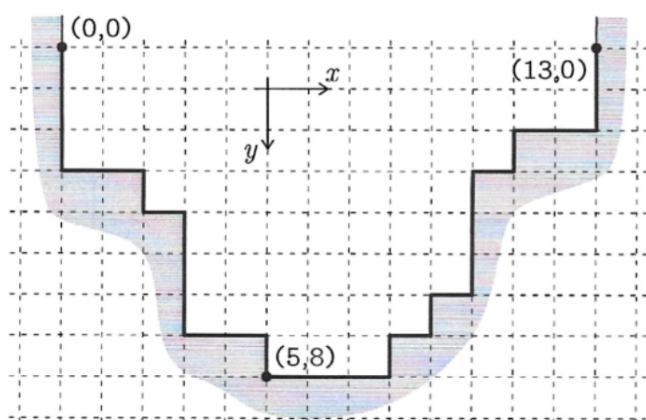
- (c) Écrire une fonction `paradoxe` prenant en argument une chaîne de caractère `code_f` correspondant au code d'une fonction `f`, telle que `paradoxe code_f` termine si et seulement si `f` ne termine pas sur `code_f`.
- (d) En utilisant `code_paradoxe`, le code de la fonction `paradoxe`, exhiber une contradiction et conclure.

Exercice 6 (Modélisation de l'inondation d'une grotte) :

Le but de cet exercice est de déterminer le remplissage d'une grotte lors d'une inondation alimentée par une source d'eau localisée au point le plus profond de la grotte. Le fond de la grotte sera décrit par le profil de sa coupe, que l'on peut décomposer sous la forme d'une suite de déplacements horizontaux ou verticaux de longueur 1, que l'on peut modéliser en OCaml par les types suivants :

```
type direction = Bas | Haut | Gauche | Droite
type profil = direction list
```

Par exemple, le profil de la grotte dessinée ci-dessous sera représenté comme suit :



Exemple

```
let ex = [Bas; Bas; Bas; Droite; Droite;
          Bas; Droite;
          Bas; Bas; Bas; Droite; Droite;
          Bas; Droite; Droite; Droite;
          Haut; Droite;
          Haut; Droite;
          Haut; Haut; Haut; Droite;
          Haut; Droite; Droite;
          Haut; Haut]
```

L'origine du profil d'une grotte sera toujours le point de coordonnées (0,0), et les coordonnées (x,y) d'un point se déplaçant le long du profil évoluent selon des axes qui pointent vers la **Droite** et vers le **Bas**.

1. Écrire une fonction `verif_profondeur` : `profil -> bool` prenant en argument le profil d'une grotte et renvoyant **True** si la fin du profil se situe à la même profondeur que son origine (et **False** sinon).

Dans la suite, on supposera (sans le vérifier) que l'on travaille uniquement avec des grottes dont la fin du profil se situe à la même profondeur que l'origine du profil.

On dit qu'un profil est **sans rebroussement** s'il ne commence pas par **Haut**, ne termine pas par **Bas**, et ne contient jamais deux pas successifs qui "s'annulent" (par ex. **Droite** suivi directement de **Gauche**).

2. Écrire une fonction `verif_sans_rebroussement` : `profil -> bool` prenant en argument le profil d'une grotte et renvoyant **True** s'il 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 arriver à la fin de son profil (en particulier, la direction **Gauche** est interdite).

Par exemple, la grotte dessinée au début de l'exercice est une vallée.

3. Écrire une fonction `verif_vallee` : `profil -> bool` prenant en argument le profil d'une grotte supposée sans rebroussement et renvoyant **True** si c'est une vallée (et **False** sinon).

Dans la suite de cet exercice, on supposera (sans le vérifier) qu'on travaille uniquement avec des vallées. Le **fond** d'une vallée est son point le plus à gauche parmi ses points les plus bas.

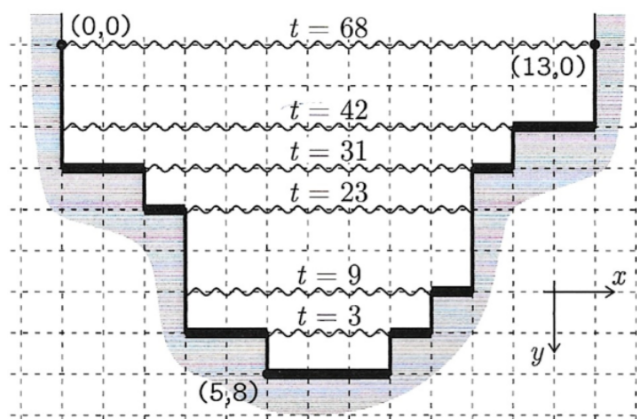
Par exemple, le fond de la vallée dessinée au début de l'exercice a pour coordonnées (5,8).

4. Écrire une fonction `fond` : `profil -> int * int` prenant en argument le profil d'une vallée et renvoyant les coordonnées de son fond.

On considère à présent qu'au temps $t = 0$, une source située au fond de la vallée commence à couler et à remplir la vallée. On cherche alors à calculer quelle sera la hauteur de l'eau dans la vallée à chaque instant t . On considère que le débit de la source est constant et unitaire, c'est-à-dire d'une unité de surface (un carreau) par unité de temps. Pour cela, on décompose le volume intérieur de la vallée en rectangles.

La figure ci-contre indique le niveau de l'eau à différents instants t dans la vallée donnée en exemple au début de l'exercice.

Dans le cas de la vallée ci-dessus, la liste des tailles (largeur, hauteur) des rectangles obtenus est, de bas en haut : $[(3, 1); (6, 1); (7, 2); (8, 1); (11, 1); (13, 2)]$.



Une **pile** est une structure de données abstraite fondée sur le principe “dernier arrivé, premier sorti”, ou LIFO (“last in, first out”). Autrement dit, lorsqu'on souhaite sortir un élément de la pile, on sort l'élément le plus récent (cet élément est au **sommet de la pile**).

De même, une **file** est une structure de données abstraite fondée sur le principe “premier arrivé, premier sorti”, ou FIFO (“first in, first out”). Autrement dit, lorsqu'on souhaite sortir un élément de la file, on sort l'élément le plus ancien (cet élément est le **prochain dans la file**).

On suppose que ces structures de données sont déjà implémentées par des types 'a pile et 'a file, et munies des fonctions suivantes dont la complexité est toujours en $\mathcal{O}(1)$:

Fonctions du type 'a pile

```

creer_pile : unit -> 'a pile
pile_vide : 'a pile -> bool
empiler : 'a pile -> 'a -> unit
sommet : 'a pile -> 'a
depiler : 'a pile -> 'a

```

Fonctions du type 'a file

```

creer_file : unit -> 'a file
file_vide : 'a file -> bool
enfiler : 'a file -> 'a -> unit
prochain : 'a file -> 'a
defiler : 'a file -> 'a

```

On décrit maintenant un algorithme permettant de décomposer une vallée en de tels rectangles.

- L'algorithme procède en parcourant le profil de la vallée une seule fois en partant de l'origine.
 - On commence par stocker dans une **pile** la position la plus à gauche pour chaque profondeur > 0 de la grotte jusqu'à atteindre le fond de la grotte.
 - ↔ La position au sommet de la pile est alors la position de la profondeur maximale.
 - On stocke alors dans une **file** la position la plus à droite pour chaque profondeur > 0 de la grotte, jusqu'à atteindre la fin du profil.
 - ↔ La première position de la file est alors la position de la profondeur maximale.
 - En dépiler et défiler en même temps la pile et la file ainsi construites, on détermine les rectangles successifs de la décomposition, que l'on stocke dans une nouvelle liste.
5. (a) Écrire une fonction `decomposition_en_rectangles : profil -> (int * int) list` de **complexité linéaire** prenant en argument le profil d'une vallée et renvoyant la liste des rectangles qui la composent, triés de bas en haut.
- (b) Justifier que votre fonction `decomposition_en_rectangles` est bien de complexité linéaire en la taille du profil pris en argument.
6. Écrire une fonction `hauteur_eau : profil -> int -> float` prenant en argument le profil d'une vallée et un entier t , et renvoyant la hauteur de l'eau dans la vallée à l'instant t .