Rappelons la définition vue dans le premier chapitre : la programmation impérative consiste à effectuer des opérations successives modifiant l'état de la mémoire. Les objets informatiques sont donc modifiables et l'ordre dans lequel les opérations sont effectuées a une importance.
Il est prouvé que tout algorithme réalisable en programmation impérative peut être écrit en programmation récursive avec la même complexité. La réciproque est également vraie. Ainsi, le paradigme de programmation est laissé au choix pour faciliter l'écriture et la compréhension des fonctions.
Ce chapitre a pour but d'approfondir les notions déjà étudiées, mais ne redéfinira pas les objets classiques comme tableaux, chaînes de caractères, références, boucles `for` et `while`, types mutables...

I Utilisation des tableaux

I.1 Fonctions avancées

Comme pour les listes, les tableaux disposent de fonctionnelles. Leur utilisation est la même que pour le type `list`, mais l'implémentation est différente. On retrouve les classiques :

- `Array.map : ('a -> 'b) -> 'a array -> 'b array`
- `Array.iter : ('a -> unit) -> 'a array -> unit`
- `Array.fold_right : ('b -> 'a -> 'a) -> 'b array -> 'a -> 'a`
- `Array.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a`

De plus, certaines fonctionnelles permettent de prendre en compte des fonctions qui agissent différemment selon l'indice d'un élément dans le tableau :

- `Array.mapi : (int -> 'a -> 'b) -> 'a array -> 'b array` prend en argument une fonction $f : [\![0; n-1]\!] \times A \longrightarrow B$ et un tableau $[|a_0, a_1, \dots, a_{n-1}|] \in A^n$ et renvoie le tableau $[|f(0, a_0), f(1, a_1), \dots, f(n-1, a_{n-1})|]$.
- La fonction `Array.iteri : (int -> 'a -> unit) -> 'a array -> unit` prend en argument une fonction f qui renvoie un type `unit`, c'est-à-dire qui modifie l'environnement, et l'applique sur chaque couple (i, a_i) du tableau.

Exercice 1

Implémenter les fonctions `map`, `iteri` et `fold_left` pour les tableaux.

Il est bon de signaler qu'il est possible de passer du type tableau au type liste, et réciproquement, en utilisant les fonctions :

- `Array.to_list : 'a array -> 'a list`
- `Array.of_list : 'a list -> 'a array`

Remarque A

Attention, il n'existe ni la fonction `List.to_array`, ni `List.of_array`.

I.2 Matrices

Les matrices peuvent être vues comme des tableaux à deux dimensions. La première idée qui nous vient à l'esprit pour créer une matrice est d'utiliser deux fois la fonction `Array.make` :

```
# let m = Array.make 3 (Array.make 4 1);;
val m : int array array = [|[|1; 1; 1; 1|]; [|1; 1; 1; 1|]; [|1; 1; 1; 1|]|]
```

La modification d'un élément se fait alors de la même façon que pour un tableau, en précisant l'indice de la ligne et de la colonne :

```
# m.(1).(2) <- 5;;
- : unit = ()
```

Cependant, après vérification, le résultat obtenu est étonnant :

```
# m;;
- : int array array = [|[|1; 1; 5; 1|]; [|1; 1; 5; 1|]; [|1; 1; 5; 1|]|]
```

Après réflexion, cela semble normale : lors de la création de la matrice, on a écrit que chaque ligne devait être la même, à savoir une colonne contenant quatre 1. Les lignes sont alors considérées comme le même objet et la modification de l'une entrainera nécessairement la modification des autres. Nous proposons une première solution pour éviter le problème :

```
let m = Array.make 3 [||];;
for i = 0 to 2 do
   m.(i) <- Array.make 4 1
done;;
m.(1).(2) <- 5;
m;;
- : int array array = [|[|1; 1; 1; 1|]; [|1; 1; 5; 1|]; [|1; 1; 1; 1|]|]
```

Ou plus simplement :

```
let m = Array.make_matrix 3 4 1;;
```

Exercice 2

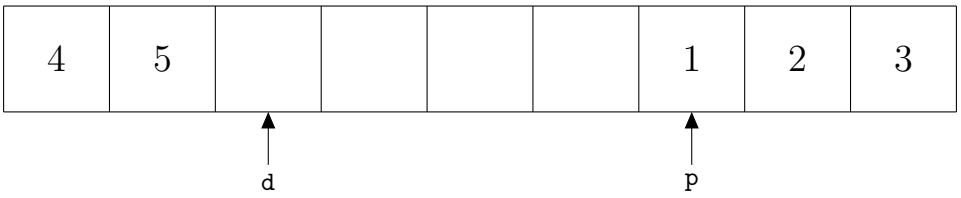
Écrire une fonction `transpose : 'a array array -> unit` qui transpose une matrice carrée.

Exercice 3

Écrire une fonction `mult : int array array -> int array array -> int array array` qui calcule le produit de deux matrices. On supposera que les dimensions permettent de faire le calcul.

I.3 Files d'attente

Nous avons vu comment implémenter des files d'attente en utilisant des listes. Il est également possible de le faire en utilisant un tableau.
L'idée est de créer un tableau associé à deux curseurs qui indiquent l'indice du premier élément (à sortir) de la file, et celui de la première case vide. On décale alors ces curseurs selon l'ajout ou le retrait d'un élément :



Pour pouvoir différencier une file vide d'une file pleine, on peut alors imposer de toujours laisser une case vide dans la file pleine :

- Si la file est vide, alors $p = d$
- Si la file est pleine, alors $p = d + 1 \mod n$ où n est la taille du tableau.

Cette implémentation permet bien de rajouter et d'enlever un élément en temps $O(1)$, mais elle impose à la liste d'avoir une taille fixe.

Exercice 4

1. Créer un type de file selon ce modèle.
2. Écrire une fonction `creer : int -> 'a -> 'a file` qui crée une file vide de taille maximale donnée. Le deuxième argument n'est là que pour imposer le type des éléments (le tableau étant

non vide).

3. Écrire des fonctions `vide : 'a file -> bool` et `pleine : 'a file -> bool` qui effectuent ces tests.
4. Écrire des fonctions `enfile : 'a -> 'a file -> unit` et `defile : 'a file -> 'a`. Ces fonctions devront écrire un message d'erreur le cas échéant.

I.4 Dictionnaires

I.4.1 Présentation de la structure

Un **dictionnaire**, appelé également **tableau d'association**, est une structure de données permettant d'associer des **valeurs** à des **clés**. Pour faire le parallèle avec un dictionnaire papier, les valeurs correspondent aux définitions et les clés correspondent aux mots qu'elles définissent. Cette structure permet de réaliser de la mémoïsation en optimisant l'espace mémoire utilisé.
Les opérations effectuées dans une structure de dictionnaire sont généralement :

- Création.
- Ajout ou suppression d'une paire (c, v).
- Modification de la valeur v associée à la clé c.
- Lecture de la valeur v associée à la clé c. L'inverse n'est en général pas possible, ou alors avec une plus grande complexité temporelle.

Dans des cas très simples, on peut se contenter de l'utilisation d'un tableau (si les clés peuvent être assimilées à des indices), mais on peut rencontrer une contrainte supplémentaire : la plage de valeurs prises par les clés est beaucoup trop grande par rapport au nombre de clés utilisées (on utiliserait alors un tableau très grand mais quasi-vide). On peut alors penser à l'utilisation de listes :

```
type ('a, 'b) dico = {mutable liste : ('a * 'b) list};;
```

Exercice 5

Écrire des fonctions `creer : unit -> ('a, 'b) dico`, `ajouter : 'a * 'b -> ('a, 'b) dico -> unit`, `supprimer : 'a -> ('a, 'b) dico -> unit`, `modifier : 'a * 'b -> ('a, 'b) dico -> unit` et `lire : 'a -> ('a, 'b) dico -> 'b`.
Déterminer les complexités de ces fonctions selon la taille n du dictionnaire.

Comme vu dans l'exercice précédent, les complexités des fonctions agissant sur les listes sont trop grandes pour que la structure puissent être considérées comme efficace. Nous avons déjà vu une solution lorsque les clés sont des chaînes de caractères (arbres Patricia), et une autre solution utilisant des arbres binaires sera étudiée en deuxième année (arbres binaires de recherche).

I.4.2 Tables de hachage

La structure que nous allons présenter maintenant utilise des tableaux. Elle est appelée **table de hachage**. Posons C l'ensemble des clés et $n = |C|$. Comme dit précédemment, il n'est souvent pas raisonnable de créer un tableau de taille n. Choisissons alors $m \ll n$, m supérieur au nombre d'associations à ajouter dans le dictionnaire, et créons un tableau de taille m. Il est alors nécessaire de disposer d'une fonction $h : C \to [\![0; m-1]\!]$, appelée **fonction de hachage**, avec les propriétés suivantes :

- Elle est facile à calculer.
- Elle limite le nombre de collisions d'images de deux clés différentes, donc doit avoir une distribution d'apparence uniforme.
- Une perturbation minime de la clé doit engendrer une perturbation majeure de l'image (toujours pour limiter le nombre de collisions).

En supposant que les clés sont des entiers (quitte à effectuer une première transformation), il existe plusieurs types de fonctions de hachage dont on peut citer :

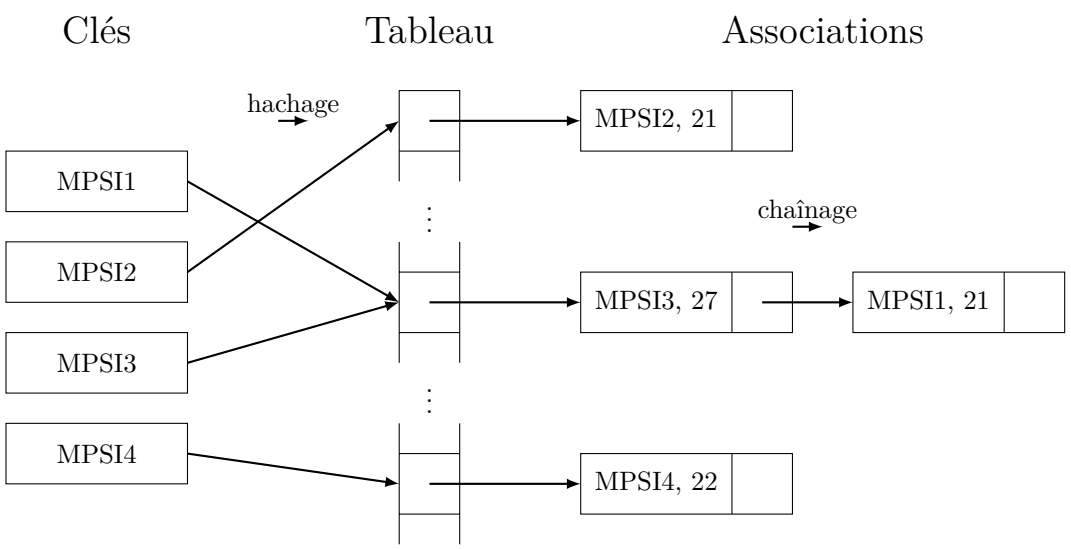
– Méthode par division : $h : c \mapsto c \mod m$. Il faut alors choisir m comme un nombre premier éloigné d'une puissance de 2. Cette méthode n'assure cependant pas la dernière des propriétés requises.
– Méthode par multiplication : $h : c \mapsto \lfloor mc\alpha \rfloor - m \lfloor c\alpha \rfloor$ où $\alpha \in]0, 1[$.

Cependant, malgré ces précautions, la fonction de hachage n'étant pas injective, des collisions peuvent se produire, et il faut prévoir des manières de les gérer.

Chaînage

Cette méthode utilise la structure de liste chaînée déjà étudiée dans l'exercice 5 pour gérer les conflits. Le nombre d'association menant à la même image par la fonction de hachage étant a priori faible, les complexités des différentes fonctions seront grandement réduites.

Exemple B



Adressage ouvert

En cas de conflit, plutôt que de laisser de nouvelles associations dans des cases du tableau déjà occupées, on se contente d'explorer le tableau jusqu'à trouver une case vide. Cette « exploration » peut se faire de différentes manières :

– On explore les cases adjacentes à la case i déjà occupées : $i+1 \mod m$, $i-1 \mod m$, $i+2 \mod m$, ...
– On utilise une nouvelle fonction de hachage h' et on cherche : $i + h'(c) \mod m$, $i + 2h'(c) \mod m$, ...

Remarque C

Dans chacune des solutions proposées, on peut être confronté à un phénomène d'agrégation : une sous-liste très grande dans une case du tableau (chaînage) ou un grand nombre de cases occupées avant de trouver une case libre (adressage ouvert). Ces cas particuliers font que la complexité dans le pire des cas pour les opérations sur les tables de hachage sont en $O(n)$ où n est le nombre d'éléments dans le dictionnaire. Cependant, ces opérations peuvent être réalisées en $O(1)$ en moyenne.

I.4.3 En `caml`

Les tables de hachages existent déjà en `caml`, dans le module `Hashtbl`. Voici les fonctions usuelles pour les manipuler :

– `Hashtbl.create : int -> ('a, 'b) Hashtbl.t` qui prend en argument le nombre maximal d'éléments supposé et crée une table vide.

- `Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` qui prend en argument une table, une clé et une valeur et rajoute une association. Si la clé est déjà utilisée, l'association précédente n'est pas effacée, mais « cachée » tant que cette nouvelle association n'est pas supprimée.
- `Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool` qui teste si une table contient une clé donnée.
- `Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b` qui prend en argument une table et une clé et renvoie la valeur associée.
- `Hashtbl.remove : ('a, 'b) Hashtbl.t -> 'a -> unit` qui supprime la dernière association dans une table de la clé donnée en argument.

II Analyse d'algorithme

II.1 Terminaison

La question de terminaison se pose quand plusieurs instructions sont répétées. Le principe de la boucle `for` est que le nombre de répétitions est connu à l'avance et, à moins que certaines opérations réalisées dans un passage de boucle ne terminent pas, la boucle `for` se terminera nécessairement.
Ainsi, en programmation impérative à l'aide de boucles itératives, seule la question de la terminaison de la boucle `while` se pose.
Comme pour la terminaison d'une fonction récursive, on utilise un ensemble bien fondé, ou plus simplement une « quantité qui diminue » à chaque passage dans la boucle :

Définition D

Un **variant de boucle** est une quantité v_n dépendant des variables d'un algorithme et du nombre de passages n dans la boucle, vérifiant :

- $v_0 \geq 0$ (positive avant l'exécution de la boucle)
- $\forall n > 0, v_n \geq 0$ (positive après chaque passage dans la boucle)
- $\forall n > 0, v_n < v_{n-1}$ (strictement décroissante)

Proposition E

Si une boucle `while` possède un variant de boucle, alors elle termine.

Exercice 6

On considère la fonction suivante :

```
let f p =
   let c = ref 0 and q = ref p in
   while !q > 0 do
      if !c = 0 then (q := !q - 2; incr c)
      else (incr q; decr c)
   done;
```

Exhiber un variant de boucle en fonction de q et c pour montrer la terminaison de la fonction.

II.2 Correction

La preuve d'une fonction itérative, qu'elle soit programmée avec une boucle `for` ou une boucle `while` s'inspire du raisonnement par récurrence.

Théorème F

Pour montrer qu'une boucle produit un certain résultat, il suffit de trouver une propriété vérifiant :

- Initialisation : La propriété est vraie avant la première itération de boucle.
- Conservation : Pour $n \in \mathbb{N}$, si la propriété est vraie après la n-ème itération, alors elle est vraie

après la $n+1$-ème itération.
- Terminaison : Si la propriété est vraie après être sorti de la boucle (`for` ou `while`), alors la boucle produit bien le résultat attendu.

La propriété est appelée **invariant de boucle**.

Exemple G

On considère la fonction suivante :

```
let fact n =
   let p = ref 1 and q = ref n in
   while !q > 0 do
      p := !p * !q; decr q
   done;
   !p;;
```

On pose p_k et q_k les valeurs de p et q **après** le k-ème passage de boucle. On pose $\mathcal{P}_k$ la propriété : $p_k = \dfrac{n!}{q_k!}$.

- Initialisation : Avant le premier passage, on a $p_0 = 1 = \dfrac{n!}{q_0!}$ donc $\mathcal{P}_k$ est vraie.
- Conservation : Supposons $\mathcal{P}_k$ vraie pour $k \in \mathbb{N}$. On a $p_{k+1} = p_k \times q_k$ et $q_{k+1} = q_k - 1$. Dès lors, $p_{k+1} = \dfrac{n!}{q_k!} \times q_k = \dfrac{n!}{(q_k-1)!} = \dfrac{n!}{q_{k+1}!}$ donc $\mathcal{P}_{k+1}$ est vraie.
- Terminaison : après être sorti de la boucle, on a $q_k = 0$. On en déduit que $p_k = n!$ et donc que la fonction calcule bien $n!$.

Exercice 7

On considère les fonctions :

```
let echange t i j =
   let x = t.(i) in
   t.(i) <- t.(j); t.(j) <- x;;

let partition t =
   let p = t.(0) and i = ref 0 in
   for j = 1 to Array.length t - 1 do
      if t.(j) < p then
         (incr i; echange t !i j)
   done;
   echange t 0 !i;;
```

En posant $\mathcal{P}_j$ la propriété : $\begin{cases} \forall k \in [\![1;i]\!], t[k] < p \\ \forall k \in [\![i+1;j]\!], t[k] \geq p \end{cases}$, montrer que la fonction `partition` repositionne les éléments d'un tableau $[t_0, t_1, \ldots, t_{n-1}]$ en mettant les éléments plus petits que t_0 au début du tableau, ceux plus grands que t_0 à la fin du tableau et t_0 entre ces deux parties.

II.3 Complexité

Dans la majeure partie des cas, le calcul de la complexité temporelle d'une fonction itérative revient à déterminer la complexité d'un passage dans la boucle et à la multiplier par le nombre d'itérations.

Exemple H

La fonction suivante a une complexité linéaire, car chaque passage dans la boucle est en temps constant, et la boucle est de la taille du tableau moins 1 :

```
let maximum t =
   let m = ref t.(0) in
   for i = 1 to Array.length t - 1 do
      m := max !m t.(i)
   done;
   !m;;
```

Cependant, la complexité peut ne pas être la même à chaque passage dans une boucle. Il faut alors faire le calcul rigoureusement (par exemple lorsque deux boucles sont imbriquées).

Exercice 8

On dispose d'une fonction `f : int -> int` qui vaut 0 pour tout entier, sauf pour un unique entier k. Écrire une fonction permettant de trouver l'entier k en un temps linéaire en k (on suppose que les images par la fonction f se calculent en temps constant).

Exercice 9

On dispose de k œufs dans un immeuble de n étages, et on souhaite déterminer l'étage critique, à savoir l'étage le plus bas duquel on peut lâcher un œuf sans qu'il se casse selon le principe :

- Si après un lâcher, l'œuf ne s'est pas cassé, on peut le réutiliser.
- Si par contre il se casse, il faut utiliser un autre œuf.

L'objectif étant de minimiser le nombre de lancers.

1. Déterminer une stratégie optimale pour $k = 1$, puis pour $k = n$ et déterminer leurs complexité.
2. Déterminer une stratégie réalisant $O(\sqrt{n})$ lâchers pour $k = 2$.

III Programmation dynamique

La programmation dynamique consiste à calculer des solutions à des petits problèmes pour pouvoir répondre à un gros problème. Elle est à la programmation itérative ce que la mémoïsation est à la programmation récursive. La différence principale entre ces deux modes de programmation est que la programmation dynamique est **bottom-up** (on travaille sur des problèmes de plus en plus grands) alors que la mémoïsation est **top-down** (on ne calcule la solution d'un problème que quand on en a besoin, à commencer par le plus grand).

Dans la suite de cette partie, nous étudierons différents problèmes résolubles à l'aide de programmation dynamique.

III.1 Plus longue sous-séquence croissante

Définition I

Soit $a = a_1, a_2, \ldots, a_n$ une suite d'entiers. Une **sous-séquence** de a est une séquence de la forme $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ où $1 \leq i_1 < i_2 < \ldots < i_k < n$. Cette sous-séquence est dite **croissante** si de plus $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$.

Comme le nom de la partie l'indique, nous souhaitons écrire un algorithme en programmation dynamique permettant de déterminer la plus longue sous-séquence croissante d'une suite. Par exemple, la plus longue sous-séquence croissante de $2; 8; 3; 1; 2; 8; 9; 7; 2; 9$ est $2; 3; 8; 9$. La sous-séquence $1; 2; 7; 9$ convient également.

Pour $i \in [\![1;n]\!]$, notons m_i la taille de la plus longue sous-séquence croissante terminant par a_i. On a :

$$m_i = 1 + \max\{m_j | j < i \text{ et } a_j < a_i\}$$

Il suffit alors de calculer les m_i pour $i \in [\![1;n]\!]$, puis de renvoyer la plus grande valeur. Nous proposons ci-dessous une fonction permettant de répondre au problème.

```
let plssc t =
   let n = Array.length t in
   let m = Array.make n 1 and prec = Array.make n 0 in
   let k = ref 0 and fin = ref 0 in
   for i = 0 to n - 1 do
      for j = 0 to i - 1 do
         if t.(j) < t.(i) && 1 + m.(j) > m.(i) then begin
            m.(i) <- 1 + m.(j);
            prec.(i) <- j
         end
      done;
      if m.(i) > !k then begin
         k := m.(i);
         fin := i
      end
   done;
   let ssc = Array.make !k t.(!fin) in
   while !k > 1 do
      decr k;
      fin := prec.(!fin);
      ssc.(!k - 1) <- t.(!fin)
   done;
   ssc;;
```

Dans cette fonction, le tableau `prec` permet de tracer l'élément précédent un autre dans la plus longue sous-séquence croissante. Les références `k` et `fin` permettent de garder en mémoire la taille et le dernier élément de celle-là. La dernière boucle `while` permet de la reconstruire à partir du tableau `prec` et des références.

La fonction a une complexité temporelle en $O(n^2)$ (Deux boucles `for` imbriquées, de complexité constante. La boucle `while` est de complexité $O(n)$).

III.2 Multiplication chaînée de matrices

Exercice 10

Supposons que l'on souhaite calculer le produit de quatre matrices A, B, C et D, de tailles respectives 50×20, 20×1, 1×10 et 10×100.

1. Déterminer le nombre de multiplications effectuées pour multiplier une matrice $m \times n$ par une matrice $n \times p$.
2. En déduire le nombre de multiplications effectuées avec le parenthésage $A \times ((B \times C) \times D)$ puis $(A \times B) \times (C \times D)$.

On cherche à écrire un algorithme qui minimise le nombre de multiplications de n matrices A_1, A_2, …, A_n de dimensions respectives $m_0 \times m_1$, $m_1 \times m_2$, …, $m_{n-1} \times m_n$. Pour cela, pour $1 \leq i \leq j \leq n$, on écrit $C(i,j)$ le nombre minimum de multiplications effectuées pour calculer $A_i \times A_{i+1} \ldots \times A_j$.

3. En imaginant $A_i \times A_{i+1} \ldots \times A_j$ comme un produit de deux matrices, déterminer la valeur de $C(i,j)$ en fonction des $C(i,k)$ et $C(k+1,j)$ pour $k \in [\![i; j-1]\!]$.
4. En déduire une fonction `chaine : int vect -> int` qui prend en argument le tableau des dimensions $[m_0, \ldots, m_n]$ et renvoie le nombre minimum de multiplications à effectuer pour calculer le produit de matrices.

5. Déterminer la complexité temporelle de la fonction précédente.
6. Comment modifier la fonction précédente pour renvoyer également le parenthésage à effectuer ?

III.3 Problème du sac à dos (Knapsack)

Exercice 11

On considère le problème suivant : on dispose de n objets de poids respectifs w_1, w_2, …, w_n et de valeurs respectives v_1, …, v_n. On a de plus un sac à dos de poids maximal W. On cherche à choisir des objets à mettre dans le sac à dos, sans dépasser son poids maximal, et en maximisant la valeur du sac.

En considérant $K(w, i)$ comme étant la valeur maximale atteignable avec les objets $1, 2, \ldots, i$ dans un sac de poids maximal w, proposer une fonction calculant la valeur maximale à mettre dans le sac à dos.

III.4 Distance d'édition

Les correcteurs orthographiques étudient la proximité d'écriture entre deux mots pour proposer une correction lorsque c'est possible.

Définition J : Distance de Levenshtein

On appelle **distance d'édition** entre deux mots u et v le nombre minimal d'éditions, c'est-à-dire insertions, suppressions ou substitutions de lettres, pour passer de u à v.

Exemple K

Voici deux « alignements » possibles de JANVIER et FEVRIER. Un tiret sur la première ligne indique une insertion, un tiret sur la deuxième ligne indique une suppression, deux lettres différentes indique une substitution.

J	A	N	V	-	I	E	R
F	E	-	V	R	I	E	R

Distance : 4

-	J	A	-	N	V	I	E	R
F	E	-	V	R	I	-	E	R

Distance : 7

Exercice 12

En posant $u = u_1 \ldots u_m$, $v = v_1 \ldots v_n$ et $E(i, j)$ la distance d'édition entre $u_1 \ldots u_i$ et $v_1 \ldots v_j$, écrire une fonction `levenshtein : string -> string -> int` qui détermine la distance d'édition entre deux mots.