Ensembles et Séquences vs. Tableaux

L2 INF301 — Algorithmique et programmation impérative

Florent Bouchez Tichadou

19 juillet 2021

De nombreux problèmes informatiques (ou algorithmiques) nécessitent une grande quantité de données. C'est pour cette raison que les tableaux sont la structure de données la plus couramment utilisée : ils permettent de stocker efficacement dans la mémoire nos données.

Dans ce document, nous allons nous dégager de la vision du tableau pour définir ce dont nous avons réellement besoin algorithmiquement : gérer un grand nombre d'éléments, indépendamment de la façon dont ces éléments sont stockés en mémoire. Nous présenterons ainsi deux structures de données de *haut-niveau*—les ensembles et les séquences—ainsi qu'une proposition d'implantation *bas-niveau*, à base de tableaux.

1 Ensembles et Séquences : structures de données de haut niveau

1.1 Ensembles

Nous souhaitons avoir accès à des *ensembles* d'éléments, qui peuvent être entiers, flottants, chaînes de caractères, ou même des types construits comme un point ou un cercle (voir document « Langage Algorithmique »). On entend ici par ensemble un « regroupement » d'éléments, et non pas la notion ensembliste mathématique. En particulier on peut retrouver **plusieurs occurrences d'un même élément** dans un ensemble (par exemple, l'ensemble des notes des étudiants d'une même classe). Il n'y a pas d'ordre particulier entre les éléments d'un ensemble.

D'un point du vue algorithmique de haut-niveau, nous n'avons pas besoin de connaître le détail du stockage en mémoire de ces ensembles. À chacun de nos besoins (à gauche), nous avons des éléments de langage algorithmique de haut-niveau qui répondent à ce besoin (à droite).

Nous considérons ainsi que nous avons ces outils à notre disposition et pouvons ainsi nous concentrer sur la résolution de notre problème, sans se soucier de problèmes « bas-niveau » (indices, placement mémoire, etc.).

- création d'un ensemble ;
- taille (ou *longueur*) d'un ensemble ;
- accès à un élément ;
- ajout d'un élément ;
- suppression d'un élément ;
- modification d'un élément ;
- itération sur tous les éléments.

E ← { 12, 4, 5, 9 }
taille(E) *ou bien* longueur (E)
soit $e \in$ E
ajoute(E, e) *ou bien* E ← E∪{e}
supprime(E, e) *ou bien* E ← E\{e}
change(E, e, e')
pour chaque e ∈ E **faire**
  ...

Exemple

E ← { }
ajoute(E, 12)
ajoute(E, 8)
ajoute(E, 5)
ajoute(E, 3)
soit $e \in$ E */* *n'importe lequel* */*
supprime(E, e)
ajoute(E, e+4)
pour chaque e ∈ E **faire**
  afficher(e)

E	e
{}	
{12}	
{12, 8}	
{12, 5, 8}	
{3, 12, 5, 8}	
{3, 12, 5, 8}	5
{3, 8, 12}	5
{9, 3, 12, 8}	5

3 9 8 12

1.2 Séquences

Les séquences sont très similaires aux ensembles à un détail près : **l'ordre entre les éléments est important**. On retrouve les mêmes outils que pour les ensembles avec quelques modifications : le besoin d'échanger deux éléments (donc changer leur ordre dans la séquence), et le besoin d'insérer un élément à un endroit précis (par exemple, après la première occurence d'un autre élément). Notez que la suppression conserve l'ordre, et que l'itération garantit de parcourir les éléments du premier au dernier.

— création ordonnée ;
— échanger deux éléments ;
— ajouter après un élément ;
— ajouter au début ;
— ajouter à la fin.

S ← $\langle 3, 13, 4, 9 \rangle$
échange (S, x, y)
ajoute_après (S, e, x) *ou bien* ajoute e à S après x
ajoute_debut (S, e) *ou bien* $S \leftarrow \langle e, S \rangle$
ajoute_fin (S, e) *ou bien* $S \leftarrow \langle S, e \rangle$

Exemple

S ← $\langle 3, 4, 9 \rangle$
ajoute_debut (S, 12)
ajoute_fin(S, 8)
echange (S, 12, 9)
supprime(S, 4)
pour chaque e ∈ S **faire**
  afficher (e)

S
$\langle 3, 4, 9 \rangle$
$\langle 12, 3, 4, 9 \rangle$
$\langle 12, 3, 4, 9, 8 \rangle$
$\langle 9, 3, 4, 12, 8 \rangle$
$\langle 9, 3, 12, 8 \rangle$

9 3 12 8

1.3 Dictionnaires ou associations

Les *dictionnaires* ou *associations* sont des structures de données courantes qui permettent de retrouver une valeur à partir d'une « clef ». Par exemple on peut associer à un numéro de téléphone un nom. Algorithmiquement, cela revient à stocker un ensemble de couples `(clef, valeur)`. On peut ainsi rechercher dans l'ensemble le couple qui a la clef voulue.

Exemple

D ← $\{('a', 5), ('l', 2), ('g', 4)\}$
ajoute (D, $('x', 1)$)
supprime(D, $'l'$)
recherche(D, $'g'$)
pour chaque (clef, val) ∈ D **faire**
  afficher (clef, "→", val)

D
$\{('a', 5), ('l', 2), ('g', 4)\}$
$\{('a', 5), ('l', 2), ('g', 4), ('x', 1)\}$
$\{('a', 5), ('g', 4), ('x', 1)\}$
"Trouvé : 4"

"$a \to 5$ $g \to 4$ $x \to 1$"

2 Implantation bas-niveau à base de tableaux

D'un point de vue haut-niveau, on n'a pas besoin de savoir comment sont stockés en mémoire nos ensembles ou séquences. Il existe d'ailleurs plusieurs possibilités, et nous allons en voir une dans ce document, avec l'utilisation de tableaux.

Un tableau représente une zone mémoire contigüe où l'on peut stocker des informations. Les tableaux sont les structures de données bas-niveau les plus utilisées car ils possèdent en particulier deux propriétés intéressantes :

— stockage compact : un élément occupe exactement la place dont il a besoin ;
— accès facile : en temps constant à partir d'un indice.

L'inconvénient majeur des tableaux est que leur taille est fixe : une fois créé, un tableau ne peut pas (ou difficilement) changer de taille, car le tableau occupe à sa création une zone de la mémoire physique et les zones qui lui sont adjacentes seront en général utilisées pour d'autres variables, tableaux, ou autres structures de données.

Le but de cette section est de montrer que les tableaux peuvent être utilisés pour représenter des ensembles ou séquences, c'est-à-dire que toutes les actions de base (ajout, suppression, etc.) peuvent être réalisées.

2.1 Implantation d'un ensemble par un tableau avec longueur explicite

Un tableau peut stocker des éléments (par exemple des entiers) dans des « cases » accédées par leur indice. La principale difficulté va être de réconcilier le caractère « dynamique » d'un ensemble (ajout et suppression) avec la contrainte de taille fixe des tableaux. Pour cela nous allons par défaut créer des tableaux « trop grands » où seulement une partie servira à stocker l'ensemble initial, le reste servant en cas d'ajout d'éléments. Pour un ensemble donné, nous avons donc besoin de trois informations bas-niveau :

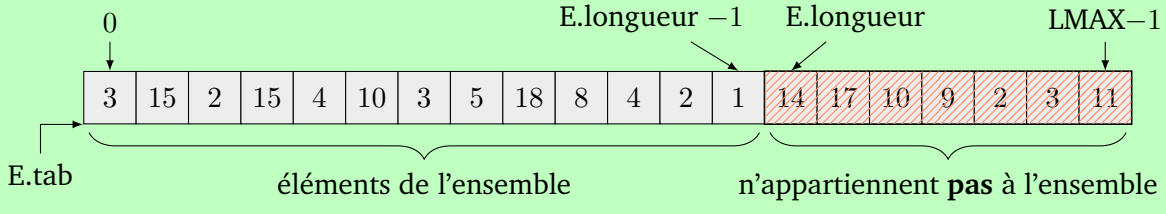
1. un tableau de taille fixe ;
2. la taille de ce tableau, qui est aussi la *longueur maximale* de l'ensemble ;
3. la *longueur* réelle de l'ensemble, i.e., le nombre de cases du tableau contenant réellement des éléments de l'ensemble. Les éléments seront stockés de la case d'indice zéro à la case d'indice longueur moins un.

Par convention, nous décidons de fixer la longueur maximale à une constante `LMAX` (par exemple 10, 100, 1234,... selon les besoins). Nous construisons un nouveau type pour stocker les deux autres informations :

```
type Ensemble : { tab : tableau de LMAX éléments, longueur : entier }
E : Ensemble /* une instance de ce type ensemble                    */
```

Exemple

Pour LMAX=20 et 13 éléments.



2.2 Implantation des opérations sur ensembles

Nous devons maintenant montrer que les opérations usuelles sur les ensembles sont possibles avec la représentation bas-niveau choisie.

2.2.1 Création d'ensemble, taille, et accès/modification des éléments

Pour la création, nous devons allouer un tableau et initialiser la longueur à zéro. On considère ici que la déclaration de la variable E crée directement le tableau.

Il n'est pas possible de manière bas-niveau de créer un ensemble contenant initialement des éléments. Pour cela, il suffit d'*ajouter* les éléments initiaux de l'ensemble un par un (voir section suivante).

La taille de l'ensemble est directement accessible *via* le champ `longueur` de la structure.

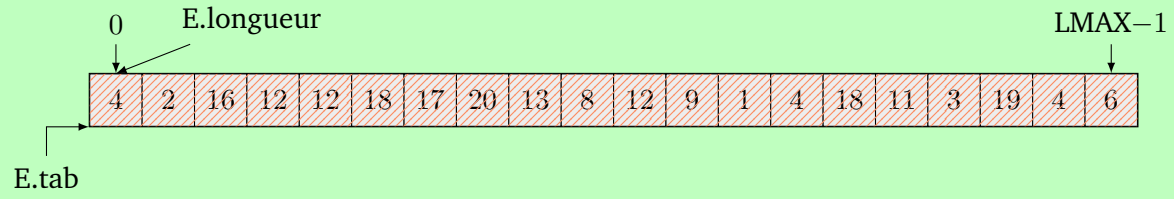
Si la longueur est non nulle, on peut accéder ou modifier (sans erreur) les éléments dans le tableau par leur indice.

```
E : Ensemble
E.longueur ← 0
```

```
afficher ("L'ensemble contient ", E.longueur, " éléments.")
E.tab[4] ← 12 /* Si longueur ⩾ 5                                  */
```

Exemple

Création d'un ensemble.



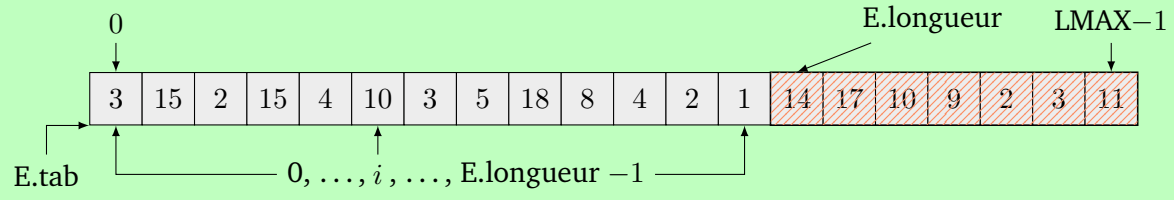
2.2.2 Itération sur tous les éléments

Parcourir un ensemble revient dans notre cas à parcourir tous les indices du tableau qui contiennent un élément : ceux de zéro à longueur moins un.

```
pour i de 0 à E.longueur −1 faire
    afficher ("Indice ", i, " : ", E.tab[i])
```

Exemple

État d'un ensemble durant un parcours.



2.2.3 Ajout d'un élément

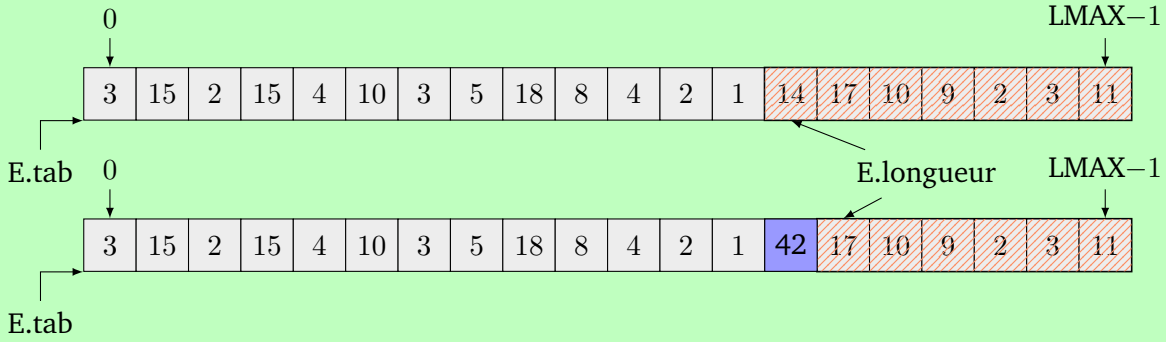
Pour ajouter un élément, il faut tout d'abord qu'il y ait encore assez de place dans le tableau. Si ce n'est pas le cas, nous considérons que c'est une erreur et l'élément n'est pas ajouté. Une autre possibilité serait par exemple de créer un nouveau tableau deux fois plus grand et recopier tous les éléments dans le nouveau tableau.

S'il y a assez de place, il suffit de placer le nouvel élément à l'indice `E.longueur`, puis incrémenter `E.longueur` pour qu'elle continue de refléter correctement la taille de l'ensemble.

```
ajoute (E : Ensemble, x : élément)
    si E.longueur = LMAX alors
        Erreur("Plus de place")
    sinon
        E.tab[E.longueur ] ← x
        E.longueur ← E.longueur +1
```

Exemple

Ajout de 42 à un ensemble



2.2.4 Suppression d'un élément

Pour la suppression, nous devons choisir une convention sur l'argument : est-ce l'élément à supprimer ou l'indice de l'élément à supprimer ? Dans le deuxième cas, nous savons directement où est dans le tableau l'élément à supprimer, dans le premier il faut d'abord effectuer une recherche.

Effectuer une recherche d'appartenance à un ensemble est de toutes façons une fonction importante. Aussi nous considérons qu'une telle fonction existe et renvoie l'indice de la première occurrence de l'élément recherché s'il appartient à l'ensemble (et -1 sinon) (voir section Exercices). Ainsi, nous pouvons réaliser l'algorithme bas-niveau de la suppression d'un élément stocké à un indice donné.

Il est important de comprendre qu'un tableau représente une zone mémoire, et qu'on ne peut pas « supprimer » le contenu d'une case mémoire : chaque case contient des bits qui sont à zéro ou à un, et donc représentent **toujours** une valeur (de la même manière, les cases entre E.longueur et LMAX−1 contiennent également des valeurs).

De plus, E.longueur doit constamment être valide, i.e., indiquer le nombre d'éléments de l'ensemble. Nous devons donc décrémenter la longueur, mais sans perdre le dernier élément. Pour cela, copions-le à la place de l'élément à supprimer (l'ordre n'est pas important ici).

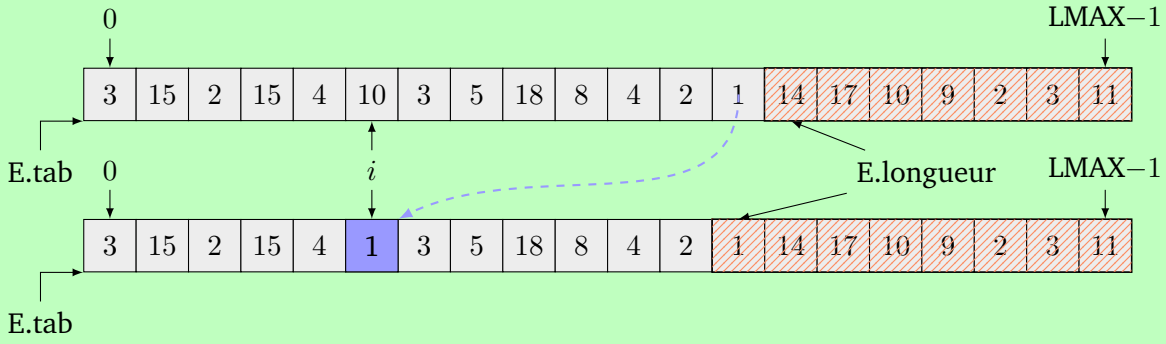
```
supprime (E : Ensemble, i : indice)
    si i < 0 ou i ⩾ LMAX alors
        Erreur("Indice incorrect")
    sinon
        E.tab[i] ← E.tab[E.longueur-1]
        E.longueur ← E.longueur-1
```

```
supprime_elt (E : Ensemble, x : élément)
    i ← cherche (E, x)
    si i < 0 alors
        Erreur("Élément introuvable")
    sinon
        E.tab[i] ← E.tab[E.longueur-1]
        E.longueur ← E.longueur-1
```

Ci-dessous, 1 est copié à la place du 10 ; il reste également à l'indice 12 où il était précédemment, mais E.longueur est décrémentée et cette case n'appartient plus à la séquence.

Exemple

Suppression de l'élément à l'indice 5.



2.3 Implantation d'une séquence par un tableau avec longueur explicite

Les séquences sont simplement des ensembles dans lequel l'ordre des éléments est important. Nous pouvons donc utiliser la même structure (tableau plus longueur). Nous allons voir ici les différences par rapport aux ensembles.

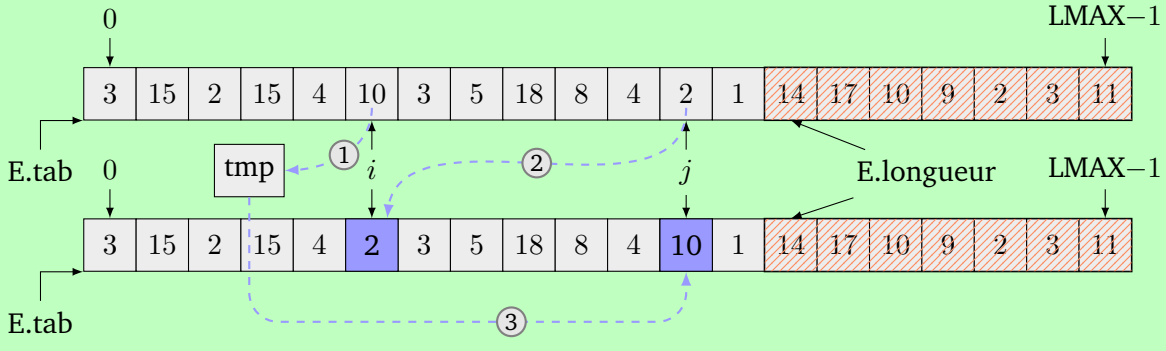
2.3.1 Échange de deux éléments

L'échange d'éléments (donnés par leurs indices) n'a pas de sens dans un ensemble puisqu'il n'y a pas d'ordre entre les éléments. C'est en revanche important pour une séquence. Pour alléger le code, nous considérons ici que les indices sont corrects (compris entre zéro et longueur moins un). Notez que l'algorithme reste correct même si $i = j$.

```
échange (S : Séquence, i, j : indices)
    tmp ← S.tab[i]
    S.tab[i] ← S.tab[j]
    S.tab[j] ← tmp
```

Exemple

Échange des indices 5 et 11.



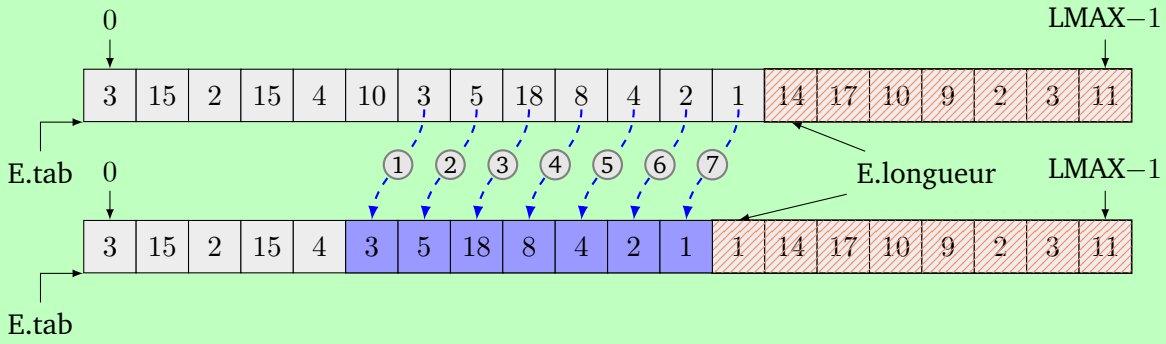
2.3.2 Suppression d'un élément

Cette fois, il nous faut conserver l'ordre des éléments. On ne peut donc juste déplacer le dernier élément à la place de l'élément à supprimer : il faut décaler tous les éléments !

```
supprimer (S : Séquence, i : indice)
    pour j de i à S.longueur −2 faire
        S.tab[j] ← S.tab[j+1]
    S.longueur ← S.longueur −1
```

Exemple

Suppression de l'élément à l'indice 5.



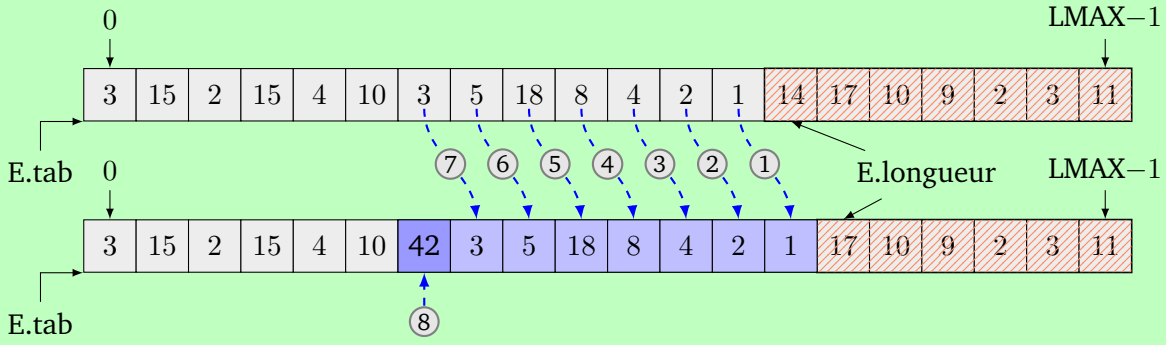
2.3.3 Ajout d'un élément

L'ordre est important, il nous faut donc proposer une fonction d'ajout à une position précise. Voici un algorithme qui ajoute un élément à un indice donné, en décalant les éléments suivants de une case. Cet algorithme permet également d'ajouter en début (indice zéro) ou fin (indice `longueur`) de séquence, ou après un élément en particulier si son indice est connu (par la recherche par exemple). Il est important de commencer à décaler les éléments en partant de la fin (donc boucle d'indice décroissant) faute de quoi écraserait toutes les valeurs entre l'indice et le dernier élément.

```
ajoute_à (S : Séquence, x : élément, i : indice)
    si S.longueur = LMAX alors
        Erreur("Plus de place")
    pour j de S.longueur −1 à i faire
        S.tab[j+1] ← S.tab[j]
    S.tab[i] ← x
    S.longueur ← S.longueur +1
```

Exemple

Ajout de 42 à de l'indice 6.



2.4 Implantation d'un dictionnaire par un tableau avec longueur explicite

Il serait possible de d'utiliser deux tableaux distincts : l'un pour les clés et l'autre pour les valeurs. La correspondance se ferait alors simplement grâce à l'indice : la valeur correspondant à `clefs[`i`]` serait `valeurs[`i`]`. Cependant, c'est souvent source de bugs de séparer ainsi des éléments qui algorithmiquement sont en fait proches (les clés de leurs valeurs). Nous conseillons donc de créer une structure correspondant à ce couple, et d'utiliser un seul tableau contenant ces structures.

Voici ci-dessous une implantation possible pour l'exemple donné précédemment (section 1.3).

```
type couple : { cle : char, valeur : entier }
type Dico : { tab : tableau de LMAX couples, longueur : entier }

recherche (D, maclef)
    trouvé ← Faux
    pour i de 0 à D.longueur −1 faire
        si D.tab[i].cle = maclef alors
            trouvé ← Vrai
            afficher ("Trouvé : ", D.tab[i].valeur)
    si non trouvé alors
        afficher ("Clé non trouvée", maclef)
```

Exercice 1 (Affichage)

Écrire un algorithme haut-niveau puis un bas-niveau qui affiche tous les éléments d'un ensemble.

Exercice 2 (Recherche)

Écrire un algorithme haut-niveau puis un bas-niveau qui recherche si un élément appartient à un ensemble. L'algorithme haut-niveau renvoie un booléen, tandis que le bas-niveau pourra renvoyer l'indice de l'élément s'il a été trouvé, ou une valeur particulière sinon.

Exercice 3 (Maximum)

Écrire un algorithme haut-niveau puis un bas-niveau qui recherche et renvoie le maximum d'une séquence. Prendre soin des cas particuliers.

Exercice 4 (Moyenne)

Écrire un algorithme haut-niveau puis un bas-niveau qui calcule la moyenne (flottante) d'un ensemble d'entiers.

Exercice 5 (Supprimer les négatifs)

Écrire un algorithme haut-niveau puis un bas-niveau qui supprime tous les éléments négatifs d'un ensemble.

Exercice 6 (Trois plus grands)

Écrire un algorithme haut-niveau puis un bas-niveau qui renvoie sous forme de triplet les trois plus grands éléments d'un ensemble.

Exercice 7 (Plus de x que de y)

Écrire un algorithme haut-niveau puis un bas-niveau qui, étant donnés un ensemble et deux valeurs x et y, renvoie Vrai s'il y a plus de x que de y dans l'ensemble et Faux sinon.

Exercice 8 (Températures)

Soit une séquence contenant les températures minute par minute depuis les dernières 24 heures.[1] Écrire un algorithme haut-niveau puis un bas-niveau qui permet de trouver la température la plus proche de $0°$C. En cas d'égalité entre une positive et une négative, renvoyer la température négative.

Exercice 9 (Somme des chiffres)

Écrire un algorithme haut-niveau puis un bas-niveau qui convertit un entier en la séquence des chiffres qui le composent (en base dix), et renvoie un couple composé de la séquence et de la somme des chiffres. Par exemple, sur l'entier 423, l'algorithme doit renvoyer $(\langle 4, 2, 3\rangle, 9)$.

Exercice 10 (Palindrome)

Un palindrome est un mot ou une phrase qui peut se lire aussi bien dans les deux sens (gauche-droite ou droite-gauche), en ne tenant compte que les lettres (sans les accents et en ignorant les espaces). Par exemple « elle » et « Élu par cette crapule. » sont des palindromes

Écrire un algorithme haut-niveau puis un bas-niveau qui vérifie si une phrase est un palindrome.

Quel est le mot palindrome le plus long de la langue française ?

1. Testez votre code Python ou C sur https://www.codingame.com/training/easy/temperatures !

Exercice 11 (Mystère 1)

Soit l'algorithme suivant qui opère sur une séquence S.

```
secret ← 0
résultat ← 0
pour chaque s ∈ S faire
    suivant s et secret faire
        cas où secret > s faire rien
        cas où secret = s faire résultat ← résultat+1
        cas où secret < s faire secret ← s ; résultat ← 1
si résultat > 0 alors
    afficher ("Le nombre secret est : ", secret)
afficher ("Le résultat est : ", résultat)
```

Question 11.1 Tracez l'exécution de l'algorithme pour $S = \langle -2, 3, 1, -1, 3, 6, 2, 6, 6, 4, 0, 6 \rangle$.

Question 11.2 Expliquez l'effet de l'algorithme pour des données quelconques. Exhibez pour cela un invariant de boucle (propriété sur les variables et la séquence qui reste vraie d'une itération à la suivante).

Exercice 12 (Intersection)

Pour cet exercice, on a unicité des éléments dans un ensemble. Écrire un algorithme haut-niveau puis un bas-niveau qui réalise l'intersection de deux ensembles.

Exercice 13 (Tri minimum)

Écrire un algorithme haut-niveau puis un bas-niveau qui réalise le tri d'une séquence par recherches successives du minimum.

Exercice 14 (Dictionnaire)

On définit un dictionnaire comme un ensemble de mots, et un mot comme une séquence de lettres.

Écrire un algorithme haut-niveau puis un bas-niveau qui recherche si un mot appartient à un dictionnaire.

Exercice 15 (Partition de dictionnaire)

Soit un dictionnaire défini comme au précédent exercice. Écrire un algorithme haut-niveau puis un bas-niveau qui partitionne le dictionnaire en fonction de la première lettre de chaque mot, i.e., en un ensemble de couples `(car, dict)` où tous les mots de `dict` commencent par la lettre `car`.