Chapitre 10

Tris
Exercices

Simon Dauguet
simon.dauguet@gmail.com

2 mai 2025

Exercice 1 :
Pour chacun des cas suivants, proposer une fonction qui renvoie une copie du tableau `tab` comportant des entiers naturels triés dans l'ordre décroissant. Préciser les variants, les invariants de boucles et la complexité temporelle.

1. `triComptageD(tab:list) -> list`
2. `triBullesD(tab:list) -> list`
3. `triInsertionD(tab:list) -> list`
4. `triSelectionD(tab:list) -> list`
5. `triFusionD(tab:list) -> list`

Exercice 2 :
Dans le cours, nous avons vu le tri à bulles qui permet de correctement positionner les plus grandes valeurs en premier. En ne comparant que des valeurs mitoyennes, que faut-il faire afin que se soit les plus petites valeurs qui se positionnent en premières ? Par exemple, `[4,8,0,5,1]` deviendrait après une étape `[0,4,8,1,5]`, puis `[0,1,4,8,5]`, et enfin `[0,1,4,5,8]`.

1. On le nommera le tri à Pierre, proposer une implémentation `triPierre(tab: list) -> list`.
2. Pour quel sorte de tableau, cet algorithme est-il le moins efficace ? Préciser la complexité temporelle dans le pire des cas.

Exercice 3 :
On souhaite écrire une fonction récursive du tri par insertion `triInsertionRec(tab: list) -> list`. Si le tableau ne contient qu'un élément ou vide, on le considérera trié, sinon on triera le sous-tableau de gauche pour ensuite insérer la dernière valeur dans ce sous-tableau dorénavant trié.

Par exemple, la pile d'exécution pour `tab = [6,1,0,5,8,1]` sera (On complète le tableau suivant en commençant par les 2 premières colonnes puis on complète en remontant les 3 dernières colonnes) :

appel	paramètre `tab`	`ttrie` + valeur	nombre de décalages pour l'insertion	sortie
1	`[6,1,0,5,8,1]`	`[0,1,5,6,8]+[1]`	3	`[0,1,1,5,6,8]`
2	`[6,1,0,5,8]`	`[0,1,5,6]+[8]`	0	`[0,1,5,6,8]`
3	`[6,1,0,5]`	`[0,1,6]+[5]`	1	`[0,1,5,6]`
4	`[6,1,0]`	`[1,6]+[0]`	2	`[0,1,6]`
5	`[6,1]`	`[6]+[1]`	1	`[1,6]`
6	`[6]`	`[]+[6]`	0	`[6]`

1. Simuler la pile d'exécution du tri par insertion récursif pour le tableau `[4,7,2,6,1,0]`.
2. Écrire alors la fonction `triInsertionRec(tab: list) -> list`.
3. Préciser les variants, invariants et la complexité temporelle.

Exercice 4 :

On souhaite écrire une fonction récursive du tri par sélection `triSelectionRec(tab: list) -> list`. Si le tableau ne contient qu'un élément ou vide, on le considérera trié, sinon on cherchera son minimum que l'on ajoute à gauche du sous-tableau, dont on exclut ce minimum, trié par appel récursif.

Par exemple, la pile d'exécution pour `tab = [6,1,0,5,8,1]` sera (On complète le tableau suivant en commençant par les 2 premières colonnes puis on complète en remontant les 3 dernières colonnes) :

appel	paramètre `tab`	minimum	tours de boucles pr déterminer le min	sortie
1	`[6,1,0,5,8,1]`	0	5	`[0]+[1,1,5,6,8]`
2	`[6,1,5,8,1]`	1	4	`[1] + [1,5,6,8]`
3	`[6,5,8,1]`	1	3	`[1]+[5,6,8]`
4	`[6,5,8]`	5	2	`[5]+[6,8]`
5	`[6,8]`	6	1	`[6] + [8]`
6	`[8]`	8	0	`[8]+[]`
7	`[]`	-	-	`[]`

1. Simuler la pile d'exécution du tri par sélection récursif pour le tableau `[4,7,2,6,1,0]`.
2. Écrire alors la fonction `triSelectionRec(tab : list) -> list`.
3. Préciser les variants, invariants et la complexité temporelle.

Exercice 5 :

Pour cet exercice, on rappelle l'existence de `chr()` et de `ord()`

1. Proposer une fonction `encodeMot(mot: str) -> int`, qui transforme un mot d'une ou deux lettres majuscules en code (tel que si `mot="let1 let2"`, on ait $n°let1 \times 26 + n°let2$). Par exemple, `encodeMot("A")` renvoie 1, `encodeMot("B")` renvoie 2, `encodeMot("Z")` renvoie 26, `encodeMot("AA")` renvoie $1*26+1 = 27$ ou `encodeMot("CE")` renvoie $3 * 26 + 5 = 83$.
2. Proposer une fonction `decodeMot(nb: int) -> str`, réciproque de `encodeMot()`. Par exemple, `decodeMot(31)` renverra `"AE"`.
3. Proposer une fonction `triComptage(tab: list) -> list`, qui renvoie une copie du tableau triée dans l'ordre croissant des codes des mots contenus dans `tab`. Par exemple, `triComptage(['A','AE','A','E','A'])` renverra `['A','A','A','E','AE']`.
4. Déterminer la complexité de chacune des fonctions dans le pire des cas.

Exercice 6 :
Proposer une fonction `triLigne(mat: list) -> list`, qui trie les lignes d'une matrice selon la somme des coefficients de chaque ligne (préciser le tri utilisé). Préciser les variants, invariants et la complexité temporelle.

Exercice 7 :
On considère une liste contenant des couples `(nom, année de naissance)`. Proposer une fonction `triFusionCouple(tab: list) -> list`, qui trie les couples du plus jeune au plus âgé. En cas d'égalité, on triera les noms par ordre alphabétique. Préciser les variants, invariants et la complexité temporelle.

Exercice 8 :
Afin de dissocier des informations capitales de patients, les états patients ont été séparées de leurs noms. Ainsi deux tableaux co-existent, `tnom =[nom0, nom1, ..., nomN]` comporte les noms, `tetats=[etat0, etat1, ..., etatN]` les états (où les `etati` est l'état du patient `nomi`). Plus l'état d'un patient est proche de 100 (%), moins il est vital de prendre en charge le patient. On souhaite trier les noms des patients selon leur état sans dévoiler leur pourcentage, ainsi on va trier un tableau d'entiers compris entre 0 et N selon le pourcentage puis transmettre ce tableau d'entiers pour retrouver le nom correspondant.

1. Proposer une fonction `triTabSeloneRef(tref: list) -> list`, qui renvoie le tableau d'entiers de 0 à N selon `tref`. Par exemple, `triTabSelonRef([77,23,85,57])` renverra `[1,3,0,2]`.
2. Proposer une fonction `nomTries(tnom: list, tab: list) -> list`, qui renvoie la liste des noms triés. Par exemple, `triTabSelonRef(['Ada','Ben','Carl','Dana'],[1,3,0,2])` renverra `['Ben','Dana', 'Ada','Carl']`.
3. Préciser les variants, invariants et les complexités temporelles de ces deux fonctions.

Exercice 9 :
Un brin d'ADN est une séquence de nucléotides : l'adénine (A) qui s'associe avec la thymine (T), ainsi que la cytosine (C) qui s'associe avec la guanine (G). Une séquence de tels nucléotides (par exemple, "AAGCA") permettra de coder un gène. On souhaite chercher puis classer certains gènes dans une séquence.

1. Proposer une fonction `existeGene(seq: str, i: int, g: str) -> int`, qui renvoie -1 si le gène `g` n'a pas été trouvé dans la sous-séquence `seq[i:]`, ou l'indice du début de séquence. Exemple : `existeGene('AAGCA', 0, 'GC')` renverra 2 mais `existeGene('AAGCA', 3, 'GC')` renverra -1.
   Préciser variant, invariant et complexité temporelle de votre fonction.
2. Proposer une fonction `compteGene(seq: str, g: str) -> list`, qui renvoie un tableau contenant tous les indices où commence le gène `g` dans la séquence `seq`. Le tableau renvoyé sera vide si `g` n'est jamais rencontré.
   Préciser variant, invariant et complexité temporelle de votre fonction.
3. On suppose que l'on possède une séquence d'ADN `seq` et une liste de gènes `lgenes`. Proposer une fonction `triGene(seq:str, lgenes:list) -> list` qui trie les gènes contenus dans `lgenes` du plus présent au moins présent dans la séquence. À égalité d'apparition, on prendra l'ordre alphabétique du gène pour différencier deux gènes distincts. Préciser le type de tri utilisé ainsi que sa complexité temporelle (un O suffira).

Exercice 10 (*(adapté d'un TD de* LPinault*)*) :
Victor souhaite avoir une vision fine des connivences musicales dans son groupe d'amis. Pour cela il demande à chacun de ses amis de classer par ordre de préférence une liste d'artistes.
Ci-suit, son classement ainsi que celui de son amie Uma (0 étant le préféré).

- Victor : Johnny Hallyday, Power Wolf, Metallica, Daniel Balavoine, Rammstein ; (noté `[0,1,2,3,4]`).
- Uma : Daniel Balavoine, Johnny Hallyday, Metallica, Power Wolf, Rammstein ; (noté `[3,0,2,1,4]`).

On définit l'*inversion* d'une paire d'artiste {I, J} lorsque Victor préfère I à J tandis qu'Uma préfère J à I. De là on dira que moins il y a d'*inversions* entre les classements de deux personnes, plus ces personnes ont des goûts musicaux proches. Par exemple, Victor préfère J.H. à P.W. tout comme Uma, il n'y a pas *inversion* pour cette paire d'artistes.

1. Compter le nombre d'*inversions* entre les classements d'Uma et de Victor.
2. Proposer une fonction `nbInversion(classU: list) -> int`, qui renvoie le nombre d'*inversions*, nombre calculé de manière naïve.

   Pour cela, on supposera que `classV` est le classement de Victor toujours ordonné, *i.e.* `classV=[0,1,..., n-1]` (on ne l'écrit donc pas). Cependant `classU` est un tableau composé de nombres entiers correspondant au classement d'Uma dont les valeurs correspondent aux indices des artistes classés par Victor.
3. Proposer les variants/invariants de la boucle principale et la complexité temporelle de cette fonction.
4. Après la lecture ci-dessous, préciser avec des mots, les étapes *diviser*, *assembler* et déterminer l'étape *régner*.

   On souhaite compter le nombre d'*inversions* entre deux listes comportant les mêmes artistes, mais de taille quelconque, en mettant en œuvre un algorithme du type "*diviser pour régner*".

   Pour cela, on coupe le classement `classU` en deux sous-classements de même taille : celui des artistes préférés `classTop` et des autres artistes `classBof`.

   Les *inversions* (j, i) (avec $i < j$) peuvent être de deux sortes :
   - soit j et i apparaissent dans le même sous-classement (`classTop` ou `classBof`), nommé *inversion* interne.
   - soit j et i apparaissent dans deux sous-classements différents ($j \in classTop$ et $i \in classBof$), nommé *inversion* mixte.

   On compte alors par appels récursifs les *inversions* de chaque sous-classement `classTop` et `classBof` (*inversions* internes), et l'on récupère chaque sous-classement trié suite à ces appels.

   On ajoute au compte d'*inversions* internes, les *inversions* mixtes grâce à une fonction `fusionMixte(classTop: list, classBof: list) -> (int, list)`. Cette fonction renvoie le nombre d'*inversions* mixtes et fusionne les deux tableaux triés en un nouveau tableau trié dans l'ordre croissant.
5. On suppose que la fonction `fusionMixte(classTop: list, classBof: list) -> (int,list)` existe.

   Proposer la fonction `nbInversionDiviserRegner(classU: list) -> (int,list)`, qui renvoie le nombre d'*inversions* du tableau `classU` ainsi que le tableau `classU` trié.
6. On souhaite définir la fonction `fusionMixte(classTop: list, classBof: list) -> (int,list)`, qui à partir de deux tableaux triés dans l'ordre croissant, renvoie un entier comptant le nombre d'*inversions* ainsi que la fusion triée des tableaux.

   Expliquer pourquoi `classTop[iT] > classBof[iB]` entraîne `len(classTop)-iT` inversions mixtes, puis écrire la fonction `fusionMixte()`.

Exercice 11 (Trier des listes partiellement triées (*X-ENS 2016 partie IV*)) :

On souhaite un algorithme de tri, qui est d'autant plus efficace que la liste donnée en entrée est déjà partiellement triée. On choisi une version simplifiée du tri utilisé par Python (qui s'appelle `TimSort`). On nommera α-tri cette version simplifiée.

Ce tri est basé sur un découpage de la liste à trier en séquences croissantes maximales d'éléments consécutifs (appelées *scm*). Ces séquences sont croissantes au sens large. Il consiste à effectuer une succession de fusions de *scm* consécutives jusqu'à n'avoir plus qu'une seule *scm*. Fusionner deux *scm* consécutives consiste à réordonner leurs éléments pour ne former qu'une seule *scm*, comme dans le tri fusion. On notera $|x|$ la longueur d'une *scm* `x`.

On rappelle qu'une *pile* `p`, outre son initialisation `p=[]`, possède deux opérations : l'ajout en fin de liste d'un élément `x` en utilisant `p.append(x)`, et la suppression du dernier élément en utilisant `p.pop()`.

L'algorithme α-tri se déroule en deux temps. On commence par partitionner la liste en *scm* consécutives, en identifiant leurs indices de début et de fin dans la liste. Dans un second temps, on effectue les fusions.

Partionnement en *scm*

Si `s` est une liste d'entiers de longueur $n \geqslant 1$, son partitionnement en *scm* est l'unique séquence de longueur $k \geqslant 1$ de couples d'entiers (d_0, f_0), (d_1, f_1), ... (d_{k-1}, f_{k-1}) telle que :

- $d_0 = 0$ et $f_{k-1} = n - 1$
- $\forall i \in [\![0, k-2]\!]$, $d_{i+1} = f_i + 1$ et $s[f_i] > s[d_{i+1}]$.
- $\forall i \in [\![0, k-1]\!]$, $d_i \leq f_i$ et la suite $s[d_i]$, $s[d_{i+1}]$, ... $s[f_i]$ est croissante au sens large

Exemple : si l'on considère la séquence `s=[3,4,8,11,1,5,2,7,9,0,10,0]`, on obtient $k = 4$ et la décomposition :

$$\underbrace{3 \leq 4 \leq 8 \leq 11}_{(d_0,f_0)=(0,3)} > \underbrace{1 \leq 5}_{(d_1,f_1)=(4,5)} > \underbrace{2 \leq 7 \leq 9}_{(d_2,f_2)=(6,8)} > \underbrace{0 \leq 10}_{(d_3,f_3)=(9,10)} > \underbrace{0}_{(d_4,f_4)=(11,11)}$$

1. Écrire une fonction `scm(s: list) -> list`, qui renvoie la liste ordonnée des couples d'indices correspondant au partitionnement en *scm* de `s`.

   Par exemple, avec comme paramètre la liste `s = [3,4,8,11,1,5,2,7,9,0,10,0]`, l'appel à `scm(s)` renverra la liste `[(0,3), (4,5), (6,8), (9,10), (11,11)]`.

Fusions de deux *scm* consécutives

Les fusions effectuées par α-tri concernent toujours deux *scm* consécutives. Nous aurons donc besoin d'une procédure pour réaliser une telle fusion.

2. Écrire une procédure `fusionner(s: list, r1: 'scm', r2: 'scm') -> None`, qui prend une liste `s` en paramètre ainsi que deux *scm* consécutives encodées par leurs indices de début et de fin, et les fusionne en une seule *scm* : si `r1 = (d1, f1)\` et `r2 = (d2, f2)`, alors après l'appel à la procédure, la partie de `s` située entre les indices `d1` et `f2` dans `s` doit être triée. Cette procédure ne crée pas une nouvelle liste, elle modifie la liste `s`.

   Remarque : Il n'est pas demandé de vérifier que les *scm* sont consécutives.

Algorithme α-tri

Les fusions des *scm* sont effectuées en deux temps. Tout d'abord, on utilise une pile initialement vide, dans laquelle les *scm* sont ajoutées une par une, dans l'ordre. À chaque fois qu'une *scm* est ajoutée, on compare les longueurs (leurs nombres d'éléments) de la dernière *scm* `z` de la pile et de l'avant-dernière `y` (si elle existe). Si $|y| < 2|z|$, on retire `y` et `z` de la pile, on les fusionne et on ajoute la *scm* fusionnée dans la pile. On continue à effectuer des fusions tant que la condition sur les longueurs des deux dernières *scm* est vérifiée. Quand on arrive à une pile avec un seul élément, ou telle que $|y| \geqslant 2|z|$, on ajoute la *scm* suivante dans la pile et on recommence les fusions éventuelles.

Dans un deuxième temps, lorsque toutes les *scm* initiales ont été ajoutées à la pile, on effectue une dernière passe en fusionnant itérativement les deux dernières *scm* de la pile, jusqu'à n'avoir plus qu'une seule *scm*. Cette *scm* est bien la liste initiale triée.

Exemple d'exécution de la phase de fusion pour `[3,4,8,11,1,5,2,7,9,0,10,0]` :

```
## Découpage en scm ##
Liste à trier : [3, 4, 8, 11, 1, 5, 2, 7, 9, 0, 10, 0]
Liste des scm : [(0, 3), (4, 5), (6, 8), (9, 10), (11, 11)]
## Première phase ##
État de la pile : [(0, 3)]
État de la pile : [(0, 3), (4, 5)]
État de la pile : [(0, 3), (4, 5), (6, 8)]
Fusion des scm : (4, 5) et (6, 8). État de la liste : [3, 4, 8, 11, 1, 2, 5, 7, 9, 0, 10, 0]
État de la pile : [(0, 3), (4, 8)]
Fusion des scm : (0, 3) et (4, 8). État de la liste : [1, 2, 3, 4, 5, 7, 8, 9, 11, 0, 10, 0]
État de la pile : [(0, 8)]
État de la pile : [(0, 8), (9, 10)]
État de la pile : [(0, 8), (9, 10), (11, 11)]
## Deuxième phase ##
Fusion des scm : (9, 10) et (11, 11). État de la liste : [1, 2, 3, 4, 5, 7, 8, 9, 11, 0, 0, 10]
État de la pile : [(0, 8), (9, 11)]
Fusion des scm : (0, 8) et (9, 11). État de la liste : [0, 0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11]
État de la pile : [(0, 11)]
La liste triée : [0, 0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11]
```

3. À l'aide de la procédure précédente **`fusionner()`**, écrire une procédure **`depileFusionneRemplace(s: list, pile`** qui prend en paramètre une liste **`s`** ainsi qu'une pile de *scm* (sous la forme de couples d'indices de début et de fin). Cette procédure devra retirer les deux *scm* au sommet de la pile, les fusionner dans la liste **`s`** et replacer les indices de la **`scm`** fusionnée au sommet de la pile.

   Remarque : On supposera que la pile contient au moins deux *scm*.

4. En utilisant les questions précédentes, écrire une procédure **`alphaTri(s: list) -> None`**, qui trie la liste **`s`** en utilisant l'algorithme α-tri décrit ci-dessus (*cf.* l'exemple).Attention, cette procédure ne crée pas une nouvelle liste, elle modifie la liste passée en paramètre.