Exercices d'algorithmique

Version 2019.08.19

Pierre-Antoine Champin

août 19, 2019

Table des matières

Les exercices plus difficiles sont indiqués par une, deux ou trois étoiles $\star$.

Des indices sont parfois données en note.

Remerciements

Ce support de cours a été initié par Pierre-Antoine Champin.

Un grand merci à
- Amélie Cordier
- Samba Ndojh Ndiaye
- Christine Solnon

pour leur contribution à ce manuel d'exercices, ainsi qu'à tous ceux qui ont participé à son amélioration par leurs remarques et commentaires.

Un autre merci aux dévelopeurs de Brython, qui a rendu possible la mise en place de l'auto-évaluation.

CHAPITRE 1

Enchaînements d'instructions

1.1 Géométrie

1. Calculer le diamètre, le périmètre et la surface d'un cercle à partir de son rayon :

```
def cercle(r: float) -> (float, float, float):
    """
    :entrée r: float
    :pré-cond: r ≥ 0
    :sortie d: float
    :sortie p: float
    :sortie s: float
    :post-conf: d contient le diamètre d'un cercle de centre r,
                p contient le périèmtre d'un cercle de centre r,
                s contient la surface d'un cercle de centre r
    """
```

2. Calculer les coefficients d'une droite à partir de deux points :

```
def coef_droite(x1: float, y1: float, x2: float, y2: float) -> (float, float):
    """
    :entrée x1: float
    :entrée y1: float
    :entrée x2: float
    :entrée y2: float
    :pré-cond: les deux points de coordonnées (x1, y1) et (x2, y2)
               sont distincts et ne sont pas alignés verticalement,
               autrement dit x1 ≠ x2
    :sortie a: float
    :sortie b: float
    :post-cond: a et b sont les coefficients de la droite passant par
                les deux points de coordonnées (x1, y1) et (x2, y2),
                autrement dit y1 = a×x1 + b et y2 = a×x2 + b

    """
```

1.2 Conditions et calcul

1. Calculer le plus petit parmi trois nombres :

```
def plus_petit(a: int, b: int, c: int) -> int:
    """
    :entree a: int
    :entree b: int
    :entree c: int
    :pré-cond: (aucune)
    :sortie pp: int
    :post-cond: pp = le plus petit nombre de l'ensemble {a, b, c}
    """
```

2. Calculer si elles existent les racines d'une équation du second degré :

```
def racines_2nd_degre(a: float, b: float, c: float) -> (float, float):
    """
    :entrée a: float
    :entrée b: float
    :entrée c: float
    :pré-cond: a ≠ 0
    :sortie r1: float ou None
    :sortie r2: float ou None
    :post-cond: si l'équation ax²+bx + c n'a pas de racine réelle,
                r1 = r2 = None,
                sinon, si elle a exactement une racine réelle,
                r1 a pour valeur cette racine et r2 = None,
                sinon (l'équation a deux racine réelles),
                r1 et r2 ont pour valeur ces deux racines,
                avec r1 > r2
    """
```

NB : pour calculer les racines, il est nécessaire de calculer une racine carrée. On pourra pour cela utiliser la fonction *racine_carree* (page 11) ci-dessous ou la fonction `sqrt` fournie par Python (inclure la ligne `from math import sqrt` en haut de votre programme).

1.3 Durées et dates

1. Convertir une durée en secondes :

```
def duree_en_secondes(j: int, h: int, m: int, s: int) -> float:
    """
    :entrée j: int
    :entrée h: int
    :entrée m: int
    :entrée s: float
    :pré-cond: j > 0 , 0 ≤ h < 24 , 0 ≤ m < 60, 0 ≤ s < 60
    :sortie ds: float
    :post-cond: ds est le nombre de secondes correspond à
                une durée de j jours, h heures, m minutes et s secondes.
    """
```

2. Convertir un nombre de secondes en durée :

```
def secondes_en_duree(sec: int) -> (int, int, int, int):
    """
    :entrée sec: int
    :pré-cond: sec ≥ 0
    :sortie j: int
```

(suite sur la page suivante)

(suite de la page précédente)

```
    :sortie h: int
    :sortie m: int
    :sortie s: int
    :post-cond: sec est le nombre de secondes correspond à
                une durée de j jours, h heures, m minutes et s secondes
                avec j > 0 , 0 ≤ h < 24 , 0 ≤ m < 60, 0 ≤ s < 60 .
    """
```

3. ⋆ Déterminer l'ordre entre deux heures de la journée :

```
def ordre_heures(h1: int, m1: int, s1: int, h2: int, m2: int, s2: int) -> int:
    """
    :entrée h1: int
    :entrée m1: int
    :entrée s1: int
    :entrée h2: int
    :entrée m2: int
    :entrée s2: int
    :pré-cond: 0 ≤ h1 < 24 , 0 ≤ m1 < 60 , 0 ≤ s1 < 60 ,
               0 ≤ h2 < 24 , 0 ≤ m2 < 60 , 0 ≤ s2 < 60
    :sortie o: int
    :post-cond: o est un entier positif si h1:m1:s1 est après h2:m2:s2 ,
                un entier négatif si h1:m1:s2 est avant h2:m2:s2 ,
                0 si les deux heures sont identiques .
    """
```

NB : notez que les post-conditions sont peu spécifiques : seul le *signe* de *o* est spécifié ; sa valeur est laissée à la discrétion de l'implémentation [1].

4. ⋆ Calculer la différence entre deux heures de la journée :

```
def difference_heures(h1: int, m1: int, s1: int, h2: int, m2: int, s2: int) ->
↪ int:
    """
    :entrée h1: int
    :entrée m1: int
    :entrée s1: int
    :entrée h2: int
    :entrée m2: int
    :entrée s2: int
    :pré-cond: 0 ≤ h1 < 24 , 0 ≤ m1 < 60 , 0 ≤ s1 < 60 ,
               0 ≤ h2 < 24 , 0 ≤ m2 < 60 , 0 ≤ s2 < 60 ,
               ordre_heures(h1, m1, s1, h2, m2, s2) ≥ 0
    :sortie d: int
    :post-cond: d est le nombre de secondes entre h2:m2:s2 et h1:m1:s1 .
    """
```

Variante : relâcher la contrainte sur l'ordre des heures de la journée passées en entrée, en retournant une valeur négative si la première est antérieure à la deuxième.

5. ⋆ Calculer une heure de la journée relativement à une autre :

```
def decale_heure(h: int, m: int, s: int, d: int) -> (int, int, int):
    """
    :entrée h: int
    :entrée m: int
    :entrée s: int
    :entrée d: int
    :pré-cond: 0 ≤ h < 24 , 0 ≤ m < 60 , 0 ≤ s < 60 , 0 ≤ d < 24*3600
    :sortie h2: int
    :sortie m2: int
    :sortie s2: int
```

(suite sur la page suivante)

1. Cette sous-spécification vous permet une utilisation astucieuse de l'algorithme *duree_en_secondes* (page 4).

(suite de la page précédente)

```
    :post-cond: h2:m2:s2 est l'heure de la journée située d secondes
                après h:m:s .
    """
```

NB : L'heure retournée en sortie peut-être « inférieure » à celle passée en entrée si la durée d fait changer de jour.

Variante : autoriser d à prendre une valeur négative pour calculer une heure située $-d$ secondes *avant* `h:m:s`.

6. Déterminer si une année est bissextile :

```
def annee_bissextile(a: int) -> bool:
    """
    :entrée a: int
    :pré-cond: a > 0
    :sortie b: bool
    :post-cond: b est True ssi l'année a est bissextile.
    """
```

NB : on rappelle que les années bissextiles sont
— les années multiples de 4,
— sauf les années multiples de 100 qui ne le sont pas,
— sauf les années multiples de 400 qui le sont tout de même.

7. Déterminer le nombre de jours d'une année donnée :

```
def jours_par_annee(a: int) -> int:
    """
    :entrée a: int
    :pré-cond: a > 0
    :sortie j: int
    :post-cond: j est le nombre de jour de l'année a.
    """
```

NB : attention aux années bissextiles –voir l'exercice *annee_bissextile* (page 6).

8. Déterminer le nombre de jours d'un mois :

```
def jours_par_mois(m: int) -> int:
    """
    :entrée m: int
    :pré-cond: 1 ≤ m ≤ 12
    :sortie j: int
    :post-cond: j est le nombre de jour du m-ième mois
                d'une année non bissextile.
    """
```

9. Déterminer le nombre de jours d'un mois d'une année donnée :

```
def jours_par_annee_mois(a: int, m: int) -> int:
    """
    :entrée a: int
    :entrée m: int
    :pré-cond: a > 0 , 1 ≤ m ≤ 12
    :sortie j: int
    :post-cond: j est le nombre de jour du m-ième mois de l'année a.
    """
```

NB : attention aux années bissextiles –voir l'exercice *annee_bissextile* (page 6).

10. Déterminer l'ordre entre deux dates :

```
def ordre_dates(a1: int, m1: int, j1: int, a2: int, m2: int, j2: int) -> int:
   """
    :entrée a1: int
```

(suite sur la page suivante)

(suite de la page précédente)

```
    :entrée m1: int
    :entrée j1: int
    :entrée a2: int
    :entrée m2: int
    :entrée j2: int
    :pré-cond: a1 > 0 , 1 ≤ m1 ≤ 12 , 1 ≤ j1 < jours_annee_mois(a1, m1) ,
               a2 > 0 , 1 ≤ m2 ≤ 12 , 1 ≤ j2 < jours_annee_mois(a2, m2)
    :sortie o: int
    :post-cond: o est un entier positif si j1/m1/a1 est après j2/m2/a2 ,
                un entier negatif si j1/m1/a1 est avant j2/m2/a2 ,
                0 si les deux dates sont identiques .
    """
```

NB : notez que les post-conditions sont peu spécifiques : seul le *signe* de *o* est spécifié.

Question subsidiaire : pouvez-vous appliquer la même astuce que pour *ordre_heures* (page 5) ? Expliquez ?

11. ⋆⋆ Calculer une date relativement à une autre :

```
def decale_date (a: int, m: int, j: int, d: int) -> (int, int, int):
    """
    :entrée a: int
    :entrée m: int
    :entrée j: int
    :entrée d: int
    :pré-cond: a > 0 , 1 ≤ m ≤ 12 , 1 ≤ j ≤ jours_par_annee_mois(a, m) ,
               d >= 0
    :sortie a2: int
    :sortie m2: int
    :sortie j2: int
    :post-cond: j2/m2/a2 est la date située d jours après j/m/a
    """
```

Variante : accepter une valeur négative pour *d* (pour trouver une date située *avant* la date passée en entrée).

12. ⋆⋆ Calculer la différence entre deux dates :

```
def difference_dates(a1: int, m1: int, j1: int, a2: int, m2: int, j2: int) ->␣
↪int:
    """
    :entrée a1: int
    :entrée m1: int
    :entrée j1: int
    :entrée a2: int
    :entrée m2: int
    :entrée j2: int
    :pré-cond: a1 > 0 , 1 ≤ m1 ≤ 12 , 1 ≤ j1 < jours_annee_mois(a1, m1) ,
               a2 > 0 , 1 ≤ m2 ≤ 12 , 1 ≤ j2 < jours_annee_mois(a2, m2) ,
               ordre_dates(a1, m1, j1, a2, m2, j2) ≥ 0
    :sortie d: int
    :post-cond: d est le nombre de jours entre j2/m2/a2 et j1/m1/a1 .
    """
```

Variante : relâcher la contrainte sur l'ordre des dates passées en entrée, en retournant une valeur négative si la première est antérieure à la deuxième.

13. Calculer le jour de la semaine d'une date donnée :

```
def jour_de_la_semaine(a: int, m: int, j: int) -> int:
    """
    :entrée a: int
    :entrée m: int
    :entrée j: int
    :pré-cond: a ≥ 1900 , 1 ≤ m ≤ 12 , 1 ≤ j < jours_annee_mois(a, m)
    :sortie js: int
```

(suite sur la page suivante)

(suite de la page précédente)

```
    :post-cond: js représente le rang dans la semaine de la date j/m/a ,
                où 0 représente le lundi et 6 représente le dimanche.
    """
```

NB : on pourra utiliser la solution de l'exercice *difference_dates* (page 7), en se souvenant que le 1^er^ janvier 1900 était un lundi.

1.4 Chiffres romains

1. Convertir un petit nombre en chiffres romains :

```
def romain_chiffre(n: int) -> str:
    """
    :entrée n: int
    :pré-cond: 0 < n < 10
    :sortie r: str
    :post-cond: r est le représentation en chiffres romains de n
    """
```

2. ⋆ Convertir de manière générique un petit chiffre en chiffres romains :

```
def romain_chiffre_generique(n: int, un: str, cinq: str, dix: str) -> str:
    """
    :entrée n: int
    :entrée un: str
    :entrée cinq: str
    :entrée dix: str
    :pré-cond: 0 < n < 10
    :sortie r: str
    :post-cond: r est le représentation en chiffres romains de n,
                en utilisant les valeurs d'entrée un, cinq et dix
                pour au lieu des symboles I, V et X respectivement.
    """
```

Par rapport à la précédente, cette fonction présente plusieurs avantages :
— elle permet d'écrire les chiffres romains en minuscules,
— elle offre un outil utile pour écrire des chiffres romains plus grands (comme dans l'exercice ci-dessous).

3. ⋆⋆ Convertir un nombre en chiffres romains :

```
def romain(n: int) -> str:
    """
    :entrée n: int
    :pré-cond: 0 < n < 4000
    :sortie r: str
    :post-cond: r est le représentation en chiffres romains de n
    """
```

On rappelle les symboles utilisés pour les chiffres romains :

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

Il pourra être utile d'utiliser la fonction *romain_chiffre_generique* (page 8) [2].

1.5 Nombres en base 10

1. Compter le nombre de chiffres :

2. Plus spécifiquement, on pourra extraire les unités, les dizaines, les centaines et les milliers, et pour chacun d'eux appeler *romain_chiffre_generique* (page 8) avec les valeurs appropriées pour un, $cinq$ et dix.

```
def compte_chiffres(n: int) -> int:
    """
    :entrée n: int
    :pré-cond: n ≥ 0
    :sortie c: int
    :post-cond: c est le nombre de chiffres dans l'écriture en base 10 de n
    """
```

2. Calculer la somme des chiffres :

```
def somme_chiffres(n: int) -> int:
    """
    :entrée n: int
    :pré-cond: n > 0
    :sortie s: int
    :post-cond: s est la somme des chiffres dans l'écriture en base 10 de n
    """
```

3. Compter le nombre d'occurrences d'un chiffre :

```
def compte_chiffre(n: int, c: int) -> int:
    """
    :entrée n: int
    :entrée c: int
    :pré-cond: n > 0,  0 ≤ c ≤ 9
    :sortie nc: int
    :post-cond: nc est le nombre de fois où le chiffre c apparaît
                dans l'écriture en base 10 de n
    """
```

4. Compter le nombre de chiffres pairs :

```
def compte_chiffres_pairs(n: int) -> int:
    """
    :entrée n: int
    :pré-cond: n ≥ 0
    :sortie c: int
    :post-cond: c est le nombre de chiffres pairs apparaissant
                dans l'écriture en base 10 de n
    """
```

5. Déterminer le chiffre à un rang donné :

```
def chiffre_au_rang(n: int, i: int) -> int:
    """
    :entrée n: int
    :entrée i: int
    :pré-cond: n > 10^i
    :sortie c: int
    :post-cond: c est le chiffre au rang r dans l'écriture de n en base 10
                (cf. ci-dessous pour la définition du rang).
    """
```

Dans cet exercice (et les suivants), on appelle **rang** d'un chiffre c dans l'écriture en base 10 d'un nombre n la puissance de 10 que représente c. Ainsi,
— le chiffre des unités a le rang 0 ($1 = 10^0$),
— le chiffre des dizaines a le rang 1 ($10 = 10^1$),
— le chiffre des centaines a le rang 2 ($100 = 10^2$),
— et ainsi de suite...

6. Déterminer le rang maximum (le plus à gauche) d'un chiffre donné :

```
def rang_max(n: int, c: int) -> int:
    """
    :entrée n: int
    :entrée c: int
    :pré-cond: n > 0, 0 ≤ c ≤ 9
    :sortie r: int
    :post-cond: r est le rang maximal du chiffre c
                dans l'écriture de n en base 10,
                ou -1 si c n'y est pas présent.
    """
```

Reportez-vous à l'exercice *chiffre_au_rang* (page 9) pour la définition du *rang*.

7. ⋆ Déterminer le rang minimum (le plus à droite) d'un chiffre donné :

```
def rang_min(n: int, c: int) -> int:
    """
    :entrée n: int
    :entrée c: int
    :pré-cond: n > 0, 0 ≤ c ≤ 9
    :sortie r: int
    :post-cond: r est le rang minimal du chiffre c
                dans l'écriture de n en base 10,
                ou -1 si c n'y est pas présent.
    """
```

Reportez-vous à l'exercice *chiffre_au_rang* (page 9) pour la définition du *rang*.

1.6 Fonctions mathématiques

1. Calculer la valeur absolue de x :

```
def valeur_absolue(x: float) -> float:
    """
    :entrée x: float
    :sortie a: float
    :post-cond: a = |x|
    """
```

2. Calculer une puissance entière de x :

```
def puissance_n(x: float, n: int) -> float:
    """
    :entrée x: float
    :entrée n: int
    :pré-cond: n ≥ 0
    :sortie p: float
    :post-cond: p = x^n
    """
```

Écrivez cet algorithme sans utiliser l'opérateur `**` de Python.
Variante : relâcher la contrainte sur n en autorisant des valeurs négatives.

3. Détermine si n est premier ou non :

```
def premier(n: int) -> bool:
    """
    :entrée n: int
    :pré-cond: n > 1
    :sortie p: bool
    :post-cond: p est True si et seulement si n est premier
    """
```

On rappelle qu'un nombre premier admet exactement deux diviseurs : 1 et lui même.

⋆⋆ Question subsidiaire : quelle est la complexité de votre algorithme ? Pouvez-vous écrire un algorithme avec une complexité meilleure que O(n) ?

4. ⋆⋆ Approximer la racine carrée de x :

```
def racine_carree(x: float, ϵ: float) -> float:
    """
    :entrée x: float
    :entrée ϵ: float
    :pré-cond: x ≥ 0
    :sortie r: float
    :post-cond: |√x - r| ≤ ϵ
    """
```

Écrivez cet algorithme sans utiliser l'opérateur `**` de Python (ni bien sûr la fonction `sqrt` du module `math`).

Une méthode possible est la **recherche dichotomique**, qui consiste à encadrer la valeur cherchée par un intervalle, puis couper répétitivement cet intervalle en deux par le milieu jusqu'à ce que sa largeur soit inférieure à ϵ. Pour cela, on fait les remarques suivantes :

— si $0 \leq x \leq 1$, alors $x \leq \sqrt{x} \leq 1$

— si $x \geq 1$, alors $1 \leq \sqrt{x} \leq x$

— quels que soient x et y positifs, $x \leq y \leftrightarrow \sqrt{x} \leq \sqrt{y}$

5. ⋆⋆ Approximer la racine n^{ème} de x :

```
def racine_nieme(x: float, n: int, ϵ: float) -> float:
    """
    :entrée x: float
    :entrée n: int
    :entrée ϵ: float
    :pré-cond: x ≥ 0 , n > 0
    :sortie r: float
    :post-cond: |ⁿ√x - r| ≤ ϵ
    """
```

Pour écrire cet algorithme, vous pouvez utiliser l'opérateur `**` de Python à condition de n'utiliser que des entiers comme opérande de droite. Vous pouvez aussi vous passer complètement de cet opérateur et utiliser à la place la fonction *puissance_n* (page 10).

On pourra appliquer la même méthode dichotomique que pour *racine_carree* (page 11).

6. ⋆⋆ Approximer le logarithme à base b de x :

```
def log_base(x: float, n: float, ϵ: float):
    """
    :entrée x: float
    :entrée n: float
    :entrée ϵ: float
    :sortie l: float
    :pré-cond: x ≥ 1
    :post-cond: |logₙ(x) - l| ≤ ϵ
    """
```

On rappelle que le logarithme à base n de x est la valeur l telle que $b^l = x$. On pourra appliquer la même méthode dichotomique que pour *racine_carree* (page 11), en utilisant l'opérateur `**` de Python.

7. Calculer la factorielle de x :

```
def factorielle(n: int) -> int:
    """
    :entrée n: int
    :sortie f: int
    :pré-cond: n > 0
    :post-cond: f = n! = 1×2×3×...×(n-2)×(n-1)×n
    """
```

8. ⋆ Calculer le nombre d'arrangements A_n^k de k éléments pris dans n :

```
def arrangements(k: int, n: int) -> int:
    """
    :entrée k: int
    :entrée n: int
    :sortie akn: int
    :pré-cond: n > 0
    :post-cond: akn = n!/(n-k)!
    """
```

   Ceci correspond au nombre de séquences que l'on peut obtenir en tirant au hasard k numéros parmi n et en tenant compte de l'ordre.

   On peut bien sûr utiliser la fonction *factorielle* (page 11) ci-dessus, mais il existe une solution plus judicieuse, qui effectue moins de calculs et évite notamment à l'ordinateur de calculer de très grands entiers pour les diviser ensuite.

9. ⋆⋆ Approximer le sinus de x :

```
def sin(x: float, ϵ: float) -> float:
    """
    :entrée x: float
    :entrée ϵ: float
    :sortie s: float
    :pré-cond: x > 0
    :post-cond: |sin(x) - s| ≤ ϵ
    """
```

   On pourra utiliser le développement limité du sinus :

   $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + o(x^{2n+2})$

10. ⋆⋆ Calculez le terme de rang n de la suite de Fibonacci :

```
def fibo(x: int) -> int:
    """
    :entrée n: int
    :sortie f: int
    :pré-cond: n > 0
    :post-cond: f est le terme de rang n de la suite de Fibonacci
    """
```

   On rappelle que la suite de Fibonnaci [3] est définie par :

$$
\begin{aligned}
&F_0 = 1 \\
&F_1 = 1 \\
&\forall n > 1, \quad F_n = F_{n-1} + F_{n-2}
\end{aligned}
$$

3. Une version récursive naïve de cette algorithme (invoquant deux fois la fonction `fibo`) est un piège à éviter car elle répétera les mêmes calculs un nombre exponentiel de fois. Une meilleuure solution consiste à écrire une boucle dans laquelle on mémorise les *deux* derniers termes calculés.

CHAPITRE 2

Chaînes de caractères

1. Calculer le nombre d'occurrences d'un caractère :

```
def compte_car(s: str, c: str) -> int:
    """
    :entrée s: str
    :entrée c: str
    :pré-cond: len(c) == 1
    :sortie n: int
    :post-cond: n est le nombre de valeurs i telles que s[i] == c
    """
```

   Exemple :

```
compte_car("hello", "l") # retourne 2
compte_car("hello", "z") # retourne 0
compte_car("hello", "H") # retourne 0 (différence entre "h" et "H")
```

2. Calculer le nombre de lettres minuscules :

```
def compte_minuscules(s: str) -> int:
    """
    :entrée s: str
    :sortie n: int
    :post-cond: n est le nombre valeurs de i telles que
                s[i] est une minuscule.
    """
```

   Pour cet exercice, on se limitera aux 26 lettres simples de l'alphabet, sans considérer les caractères accentués ou altérés.

3. Déterminer l'indice minimum (le plus à gauche) d'un caractère :

```
def indice_min_car(s: str, c: str) -> int:
    """
    :entrée s: str
    :entrée c: str
    :pré-cond: len(c) == 1
    :sortie imin: int ou None
    :post-cond: imin est la plus petite valeur telle que s[imin] == c ,
                ou None si s ne contient pas c.
    """
```

4. Déterminer l'indice maximum (le plus à droite) d'un caractère :

```
def indice_max_car(s: str, c: str) -> int:
    """
    :entrée s: str
    :entrée c: str
    :pré-cond: len(c) == 1
    :sortie imax: int ou None
    :post-cond: imax est la plus grande valeur telle que s[imax] == c ,
                ou None si s ne contient pas c.
    """
```

5. ⋆ Déterminer l'indice suivant (c.à.d. plus à droite) d'un caractère :

```
def indice_suivant_car(s: str, c: str, ig: int) -> int:
    """
    :entrée s: str
    :entrée c: str
    :entrée ig: int
    :pré-cond: len(c) == 1
    :sortie ic: int ou None
    :post-cond: ic est la plus petite valeur telle que
                ig < ic et s[ic] == c ,
                ou None si s[ig+1:] ne contient pas c.
    """
```

6. ⋆ Déterminer l'indice précédent (c.à.d. plus à gauche) d'un caractère :

```
def indice_prec_car(s: str, c: str, id: int) -> int:
    """
    :entrée s: str
    :entrée c: str
    :entrée id: int
    :pré-cond: len(c) == 1
    :sortie ic: int ou None
    :post-cond: ic est la plus grande valeur telle que
                ic < id et s[ic] == c ,
                ou None si s[:id] ne contient pas c.
    """
```

7. Déterminer si une chaîne commence par une autre :

```
def commence_par(s: str, t: str) -> bool:
    """
    :entrée s: str
    :entrée t: str
    :sortie c: bool
    :post-cond: c est True si et seulement si
                pour tout i tel que 0 ≤ i < len(t) , s[i] == t[i] .
    """
```

8. ⋆ Déterminer l'indice minimum (le plus à gauche) d'une sous-chaîne :

```
def indice_min(s: str, t: str) -> int:
    """
    :entrée s: str
    :entrée t: str
    :sortie imin: int ou None
    :post-cond: imin est la plus petite valeur telle que,
                pour tout i tel que 0 ≤ i < len(t), s[imin+i] == t[i] ,
                ou None si s n'admet pas t pour sous-chaîne.
    """
```

9. ⋆ Déterminer l'indice maximum (le plus à gauche) d'une sous-chaîne :

```
def indice_max(s: str, t: str) -> int:
    """
    :entrée s: str
    :entrée t: str
    :sortie imax: int ou None
    :post-cond: imax est la plus grande valeur telle que,
                pour tout i tel que 0 ≤ i < len(t), s[imax+i] == t[i] ,
                ou None si s n'admet pas t pour sous-chaîne.
    """
```

10. ⋆ Déterminer l'indice suivant (c.à.d. plus à droite) d'une sous-chaîne :

```
def indice_suivant(s: str, t: str, ig: int) -> int:
    """
    :entrée s: str
    :entrée t: str
    :entrée ig: int
    :pré-cond: len(c) == 1
    :sortie it: int ou None
    :post-cond: it est la plus petite valeur telle que ig < it et
                pour tout i tel que 0 ≤ i < len(t), s[imin+i] == t[i] ,
                ou None si s[ig+1:] n'admet pas t pour sous-chaîne.
    """
```

11. ⋆ Déterminer l'indice précédent (c.à.d. plus à gauche) d'une sous-chaîne :

```
def indice_prec(s: str, t: str, id: int) -> int:
    """
    :entrée s: str
    :entrée t: str
    :entrée id: int
    :sortie it: int ou None
    :pré-cond: len(c) == 1
    :post-cond: it est la plus petite valeur telle que it < id et
                pour tout i tel que 0 ≤ i < len(t), s[imin+i] == t[i] ,
                ou None si s[:id] n'admet pas t pour sous-chaîne.
    """
```

12. Calcule une chaîne inversée :

```
def inverse(s: str) -> str:
    """
    :entrée s: str
    :sortie t: str
    :post-cond: len(t) == len(s) et
                pour tout i tel que 0 ≤ i < len(s), t[i] == s[-i-1] .
    """
```

13. Détermine si une chaîne est un palindrome :

```
def palindrome(s: str) -> bool:
    """
    :entrée s: str
    :sortie p: bool
    :post-cond: p est True si et seulement si
                pour tout i tel que 0 ≤ i < len(s), s[i] == s[-i-1] .
    """
```

14. ⋆⋆ Compte le nombre de mots dans une chaîne :

```
def compte_mots(s: str) -> int:
    """
    :entrée s: str
```

(suite sur la page suivante)

(suite de la page précédente)

```
    :sortie m: int
    :post-cond: m est le nombre de mots dans s
    """
```

On considère comme un mot toute séquence de caractères différents de l'espace (même si ce ne sont pas des lettres : chiffres, symboles de ponctuation. . .). Compter les mots consiste donc à compter le nombre de « non-espaces » situés juste après une espace (ou en début de chaîne).

15. ⋆ Vérifier si une chaîne de caractères est bien parenthésées :

```
def bien_parenthesee(txt: str) -> bool:
    """
    :entrée txt: str
    :sortie bp: bool
    :post-cond: bp est True si et seulement si txt est bien parenthésée
    """
```

Toute parenthèse ouverte doit ensuite être fermée, et une parenthèse ne peut pas être fermée si elle n'a pas été préalablement ouverte [1]. Le tableau ci-dessous donne des exemples de chaînes bien et mal parenthésées.

Bien parenthésées	Mal parenthésées
`abc`	`(`
`(abc)`	`)`
`ab(cd)ef`	`abc)`
`a(b)c(d)e`	`ab)c`
`a((b)c)d`	`a(b(c)d`
`a(b(c()e)f)g`	`a(b)c)d(e)f`

16. Calculer la valeur numérique d'un entier représenté en binaire par une chaîne de caractères :

```
def eval_binaire(txt: str) -> int:
    """
    :entrée txt: str
    :pré-cond: txt ne contient que des caractères de ``0`` et ``1``
    :sortie val: int
    :post-cond: val est la valeur de l'entier représenté (en base 2)
                par txt
    """
```

Vous n'utiliserez bien sûr pas la fonction `int` de Python, qui permet de faire cela.

17. Calculer la valeur numérique d'un entier représenté par une chaîne de caractères :

```
def eval_decimal(txt):
    """
    :entrée txt: str
    :pré-cond: txt ne contient que des caractères de ``0`` à ``9``
    :sortie val: int
    :post-cond: val est la valeur de l'entier représenté (en base 10)
                par txt
    """
```

Vous n'utiliserez bien sûr pas la fonction `int` de Python, qui permet de faire cela.

NB : bien que ce ne soit pas obligatoire, l'algorithme peut-être simplifié en utilisant la fonction `ord(c)`, qui retourne le code numérique (un entier) du caractère c, et en exploitant le fait que les codes des caractères numériques se suivent, donc `ord('1') == ord('0')+1`, `ord('2') == ord('1')+1`, etc.

Variante : on autorise maintenant le premier caractère à être le signe moins `-`.

18. Calculer la représentation binaire d'un entier :

1. Une solution consiste donc à parcourir la chaîne de gauche à droite en maintenant un compte du nombre de parenthèses ouvertes et non encore fermées.

```
def repr_binaire(val: int) -> str:
    """
    :entrée val: int
    :pré-cond: val >= 0
    :sortie txt: str
    :post-cond: txt est la représentation binaire de val
    """
```

Vous n'utiliserez bien sûr pas la fonction `format` de Python, qui permet de faire cela.
Variante : on autorise maintenant val à être négatif.

19. Calculer la représentation en base 10 d'un entier :

```
def repr_decimal(val: int) -> str:
    """
    :entrée val: int
    :pré-cond: val >= 0
    :sortie txt: str
    :post-cond: txt est la représentation en base 10 de val
    """
```

Vous n'utiliserez bien sûr pas les fonctions `format` ou `str` de Python, qui permettent de faire cela.
NB : bien que ce ne soit pas obligatoire, l'algorithme peut-être simplifié en utilisant
— la fonction `ord(c)`, qui retourne le code numérique (un entier) du caractère c,
— la fonction `chr(i)`, qui retourne le caractère ayant i pour code numérique,
— et le fait que les codes des caractères numériques se suivent, donc `chr(ord('0') + 1) == '1'`, `chr(ord('0') + 2) == '2'`, etc.
Variante : on autorise maintenant val à être négatif.

20. ⋆ Calculer la représentation hexadécimale (en base 16) d'un entier :

```
def repr_hexadecimal(val: int) -> str:
    """
    :entrée val: int
    :sortie txt: str
    :pré-cond: val >= 0
    :post-cond: txt est la représentation en base 10 de val
    """
```

On rappelle qu'en hexadécimal, on utilise 16 chiffres, de `0` à `9` et de `A` à `F`.
Vous n'utiliserez bien sûr pas la fonction `format` de Python, qui permet de faire cela.
NB : On pourra utiliser, comme dans l'exercice précédent, les fonctions `ord` et `chr`, mais en faisant attention au fait que `chr(ord('0') + 10)` n'est *pas* égal à `'A'`...

21. Décompresse une chaîne de caractère :

```
def decompresse(comp: str) -> str:
    """
    :entrée comp: str
    :pré-cond: len(comp) est paire; tous les caractères d'indice pair sont␣
↪des chiffres (entre 0 et 9)
    :sortie decomp: str
    :post-cond: 'décomp' est calculé en répétant chaque caractère d'indice␣
↪impair de 'comp'
                par la valeur qui le précède
                par exemple "3a0b1c2a49" -> "aaacaa9999"
    """
```

22. ⋆ Compresse une chaîne de caractère :

```
def compresse(txt: str) -> str:
    """
    :entrée txt: str
    :pré-cond: len(str) est paire; tous les caractères d'indice pair sont des␣
↪chiffires (entre 0 et 9)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    :sortie compresse: str
    :post-cond: 'compressé' est la une chaîne telle que
                ``decompresse(compresse)`` donne ``txt``
                (cf. l'algo ``decompresse`` ci-dessus)
    """
```

CHAPITRE 3

Appels et passages de paramètres

1. Écrivez l'algorithme suivant :

```
def reutilisation(a: int, b: int, c: int) -> int:
    """
    :pré-cond: a ≥ 0, b ≥ 0, c ≥ 0
    :post-cond: d = x^n ou x est le plus petit nombre parmi a, b et c,
                et n est le plus grand nombre parmi a, b et c
    """
```

   Écrivez cet algorithme sans utiliser l'opérateur `**` de Python. Vous pouvez cependant utiliser certaines des fonctions du chapitre précédent [1], notamment *plus_petit* (page 4) et *puissance_n* (page 10).

2. Soient les trois algorithmes (idiots) suivants :

```
def toto(a: int) -> int:
    """
    :post-cond: ?
    """
    n = 4
    n = titi(a*2, n*a)
    b = 3*n - 2
    return b

def titi(a: int, b: int) -> int:
    """
    :post-cond: ?
    """
    n = tutu(a+b, a*2)
    c = n-a
    return c

def tutu(a: int, b: int) -> int:
    """
    :post-cond: ?
    """
    if a < b:
        c = a
```

(suite sur la page suivante)

1. Pour trouver le plus *grand* nombre parmi trois, vous pouvez écrire une nouvelle fonction, ou réutiliser la fonction *plus_petit* (page 4) en choisissant astucieusement les valeurs passées en paramètres.

(suite de la page précédente)

```
    else:
        c = b
    return c
```

On souhaite connaître la valeur obtenue après l'exécution de `toto(3)`. Pour cela, simulez l'exécution de cette instruction en représentant les environnement d'exécution des fonctions.

3. Calculer la somme des n premières factorielles :

```
def somme_fact(n: int) -> int:
    """
    :pré-cond: n ≥ 1
    :post-conf: sf = somme des i! pour i ∈ [0, n-1]
    """
```

On pourra bien sûr réutiliser la fonction *factorielle* (page 11) définie au chapitre précédent… ou pas.

CHAPITRE 4

Tableaux

4.1 Comparaisons et tests de tableaux

1. Déterminer si deux tableaux sont identiques :

```
def tableaux_egaux(a: [int], b: [int]) -> bool:
    """
    :entrée a: tableau d'int
    :entrée b: tableau d'int
    :sortie te: bool
    :post-cond: te est True si et seulement si len(a) == len(b) et
                ∀ i ∈ [0;len(a)[, a[i] == b[i]

    >>> tableaux_egaux([1, 2, 4, 3], [1, 2, 4, 3])
    True
    >>> tableaux_egaux([1, 2, 4, 3], [9, 2, 4, 3])
    False
    >>> tableaux_egaux([1, 2, 4, 3], [1, 2, 4])
    False
    >>> tableaux_egaux([1, 2, 4], [1, 2, 4, 3])
    False
    """
```

2. Déterminer si le tableau b est un préfixe du tableau a :

```
def tableau_prefixe(a: [int], b: [int]) -> bool:
    """
    :entrée a: tableau d'int
    :entrée b: tableau d'int
    :sortie te: bool
    :post-cond: te est True si et seulement si len(a) >= len(b) et
                ∀ i ∈ [0;len(b)[, a[i] == b[i]

    >>> tableaux_prefixe([1, 2, 4, 3], [1, 2, 4, 3])
    True
    >>> tableaux_prefixe([1, 2, 4, 3], [9, 2, 4, 3])
    False
    >>> tableaux_prefixe([1, 2, 4, 3], [1, 2, 4])
    True
```

(suite sur la page suivante)

(suite de la page précédente)

```
    >>> tableaux_prefixe([1, 2, 4], [1, 2, 4, 3])
    False
    """
```

3. Retourner True si et seulement si le tableau a est un Palindrome :

```
def tableau_palindrome(a: [int]) -> bool:
    """
    :entrée a: tableau d'int
    :sortie p: bool
    :post-cond: p est True si et seulement si
                ∀ i ∈ [0;len(a)[,  a[i] == a[len(a)-1-i]

    >>> tableau_palindrome([1, 2, 4, 3])
    False
    >>> tableau_palindrome([1, 2, 4, 2, 1])
    True
    >>> tableau_palindrome([1, 2, 2, 1])
    True
    >>> tableau_palindrome([4])
    True
    """
```

4.2 Recherche de valeurs

1. Trouver l'indice du plus petit élément d'un tableau :

```
def indice_min(a: [float]) -> int:
    """
    :entrée a: tableau de float
    :pré-cond: len(a) > 0
    :sortie imin: int
    :post-cond: ∀ i ∈ [0;len(a)[, a[imin] ≤ a[i]

    >>> indice_min([2.0, 1.0, -3.0, 7.0, 3.0, 5.0, 1.0])
    2
    >>> indice_min([2.0, 1.0, -3.0, 7.0, -3.0, 5.0, 1.0])
    2
    >>> # mais 4 est aussi une réponse correcte
    """
```

NB : si le tableau contient plusieurs minima ex-æquo, cette spécification n'impose pas lequel doit être retourné.

2. Trouver l'indice le plus à gauche du plus petit élément d'un tableau :

```
def indice_min_gauche(a: [float]) -> int:
    """
    :entrée a: tableau de float
    :pré-cond: len(a) > 0
    :sortie imin: int
    :post-cond: ∀ i ∈ [0;imin[,       a[imin] < a[i],
                ∀ i ∈ ]imin;len(a)[,  a[imin] ≤ a[i],

    >>> indice_min_gauche([2.0, 1.0, -3.0, 7.0, 3.0, 5.0, 1.0])
    2
    >>> indice_min_gauche([2.0, 1.0, -3.0, 7.0, -3.0, 5.0, 1.0])
    2
    """
```

NB : La différence avec *indice_min* (page 22) ci-dessus est que, si le tableau contient plusieurs minima ex-aequo, cette spécification impose de retourner l'indice du plus à gauche.

3. Trouver l'indice le plus à droite du plus petit élément d'un tableau :

```
def indice_min_droite(a: [float]) -> int:
    """
    :entrée a: tableau de float
    :pré-cond: len(a) > 0
    :sortie imin: int
    :post-cond: ∀ i ∈ [0;imin[,       a[imin] ≤ a[i],
                ∀ i ∈ ]imin;len(a)[,  a[imin] < a[i],

    >>> indice_min_droite([2.0, 1.0, -3.0, 7.0, 3.0, 5.0, 1.0])
    2
    >>> indice_min_droite([2.0, 1.0, -3.0, 7.0, -3.0, 5.0, 1.0])
    4
    """
```

4. Trouver l'indice du plus grand élément d'un tableau :

```
def indice_max(a: [float]) -> int:
    """
    :entrée a: tableau de float
    :sortie imax: int
    :pré-cond: len(a) > 0
    :post-cond: ∀ i ∈ [0;len(a)[, a[imax] ≤ a[i]

    >>> indice_max([2.0, 1.0, -3.0, 7.0, 3.0, 5.0, 1.0])
    3
    >>> indice_max([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0])
    3
    >>> # mais 5 est aussi une réponse correct
    """
```

NB : si le tableau contient plusieurs maxima ex-aequo, cette spécification n'impose pas lequel doit être retourné.

5. Trouver l'indice le plus à gauche de la valeur v dans un tableau :

```
def indice_gauche(a: [float], v: float) -> int:
    """
    :entrée a: tableau de float
    :entrée v: float
    :sortie ival: int
    :post-cond: si v ∈ a,   a[ival] = v  et  ∀ i ∈ [0;ival[, a[i] ≠ v
                sinon       ival = -1

    >>> indice_gauche([2.0, 1.0, -3.0, 7.0, 3.0, 5.0, 1.0], 2.0)
    0
    >>> indice_gauche([2.0, 1.0, -3.0, 7.0, 3.0, 5.0, 1.0], 7.0)
    3
    >>> indice_gauche([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 7.0)
    3
    >>> indice_gauche([2.0, 1.0, -3.0, 7.0, 3.0, 5.0, 1.0], 9.0)
    -1
    """
```

6. Trouver l'indice le plus à droite de la valeur v dans un tableau :

```
def indice_droite(a: [float], v: float) -> int:
    """
    :entrée a: tableau de float
    :entrée v: float
    :sortie ival: int
```

(suite sur la page suivante)

(suite de la page précédente)

```
    :post-cond: si v ∈ a,  a[ival] = v  et  ∀ i ∈ ]ival, len(a)[, a[i] ≠ v
                sinon      ival = -1

    >>> indice_droite([2.0, 1.0, -3.0, 7.0, 3.0, 5.0, 1.0], 2.0)
    0
    >>> indice_droite([2.0, 1.0, -3.0, 7.0, 3.0, 5.0, 1.0], 7.0)
    3
    >>> indice_droite([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 7.0)
    5
    >>> indice_droite([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 9.0)
    -1
    """
```

7. Trouver l'indice de la n-ème occurrence de v dans un tableau :

```
def indice_n(a: [float], v: float, n: int) -> int:
    """
    :entrée a: tableau de float
    :entrée v: float
    :entrée n: int
    :sortie ival: int
    :pré-cond: n > 0
    :post-cond: si v apparaît au moins n fois dans a,
                alors a[ival] = v,
                   et a[0:ival] contient v exactement n-1 fois,
                sinon ival = -1

    >>> indice_n([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 2.0, 1)
    0
    >>> indice_n([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 2.0, 2)
    -1
    >>> indice_n([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 7.0, 1)
    3
    >>> indice_n([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 7.0, 2)
    5
    >>> indice_n([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 7.0, 3)
    -1
    >>> indice_n([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 9.0, 1)
    -1
    >>> indice_n([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 9.0, 2)
    -1
    """
```

8. Compter le nombre d'occurrences de v dans un tableau :

```
def nb_occurences(a: [float], v: float) -> int:
    """
    :entrée a: tableau de float
    :entrée v: float
    :sortie n: int
    :post-conf: n = |{ i | i ∈ [0;len(a)[  et  a[i] = n }|

    >>> nb_occurences([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 2.0)
    1
    >>> nb_occurences([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 7.0)
    2
    >>> nb_occurences([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0], 9.0)
    0
    """
```

4.3 Calculs sur les valeurs

1. Calculer la somme des éléments d'un tableau :

```
def somme(a: [float]) -> float:
    """
    :entrée a: tableau de float
    :pré-cond: len(a) > 0
    :sortie s: float
    :post-cond: s = Σ a[i]  pour  i ∈ [0;len(a)[

    >>> somme([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0])
    18.0
    """
```

2. Calculer la moyenne des éléments d'un tableau :

```
def moyenne(a: [float]) -> float:
    """
    :entrée a: tableau de float
    :pré-cond: len(a) > 0
    :sortie m: float
    :post-cond: s est la moyenne des éléments de a

    >>> moyenne([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0])
    2.5714285714285716
    """
```

3. ⋆ Calculer la moyenne pondérée des éléments d'un tableau :

```
def moyenne_ponderee(a: [float], poids: [float]) -> float:
    """
    :entrée a: tableau de float
    :entrée poids: tableau de float
    :pré-cond: len(a) > 0,  len(poids) == len(a),  somme(poids) != 0
    :sortie mp: float
    :post-cond: s est la moyenne pondérée des éléments de a telle que
                ∀ i ∈ [0;len(a)[,  a[i] est pondérée par poids[i]

    >>> moyenne_ponderee([10.0, 12.0, 7.0], [1.0, 2.0, 1.0])
    10.25
    """
```

4.4 Génération et modification de tableaux

1. ⋆ Retourner un tableau contenant les n premiers nombres premier :

```
def premiers(n: int) -> [int]:
    """
    :entrée n: int
    :pré-cond: n > 0
    :sortie a: tableau d'int
    :post-cond: len(a) == n
                ∀ i ∈ [0;n[,  a[i] est le (i+1)-ème nombre premier

    >>> premiers(3)
    array([2, 3, 5])
    >>> premiers(10)
    array([2, 3, 5, 7, 11, 13, 17, 19, 23, 29])
    """
```

On rappelle qu'un nombre premier admet exactement deux diviseurs : 1 et lui même.
Par exemple, premiers(10) retournera le tableau `[2,3,5,7,11,13,17,19,23,29]` [1].

2. ⋆ Retourner un tableau contenant les n premiers termes de la suite de Fibonacci :

```python
def fibonacci(n: int) -> [int]:
    """
    :entrée n: int
    :pré-cond: n > 0
    :sortie a: tableau d'int
    :post-cond: len(a) == n
                ∀ i ∈ [0;n[,  a[i] = Fi

    >>> fibonacci(4)
    array([1, 1, 2, 3])
    >>> fibonacci(10)
    array([1, 1, 2, 3, 5, 8, 13, 21, 34, 55])
    """
```

On rappelle que la suite de Fibonnaci [2] est définie par :

$$
\begin{aligned}
&F_0 = 1\\
&F_1 = 1\\
&\forall n > 1,\ \ F_n = F_{n-1} + F_{n-2}
\end{aligned}
$$

3. ⋆ Retourner un tableau de n booléen indiquant, pour chaque nombre entre 0 et n-1, s'il est premier ou non :

```python
def eratosthene(n: int) -> [bool]:
    """
    :entrée n: int
    :pré-cond: n > 0
    :sortie a: tableau de bool
    :post-cond: len(a) == n
                ∀ i ∈ [0;n[,  a[i] est True si et seulement si i est premier

    >>> eratosthene(4)
    array([False, False, True, True])
    >>> eratosthene(10)
    array([False, False, True, True, False, True, False, True, False, False])
    """
```

La méthode proposée (le crible d'Ératosthène) consiste à :
— initialiser tous les éléments du tableau à True (à part 0 et 1),
— pour chaque élément à partir de 2, s'il vaut True, mettre tous ses multiples à False.
À la fin de ce processus, les nombres premiers et uniquement eux seront à True.

4. Retournerr le tableau inverse de a :

```python
def tableau_inverse(a: [float]) -> [float]:
    """
    :entrée a: tableau de float
    :sortie b: tableau de float
    :post-cond: ∀ i ∈ [0;len(a)[,  b[i] == a[len(a)-1-i]

    >>> tableau_inverse([])
    array([])
    >>> tableau_inverse([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0])
    array([1.0, 7.0, 3.0, 7.0, -3.0, 1.0, 2.0])
    """
```

1. Bien que cela soit possible, il n'est pas conseillé de réutiliser la fonction *premier* (page 10), car il existe une solution plus efficace qui ne l'utilise pas.
2. Bien que cela soit possible, il n'est pas conseillé de réutiliser la fonction *fibo* (page 12), car il existe une solution plus efficace qui ne l'utilise pas.

5. ⋆ Calculer les moyennes par groupe pour un tableau contenant toutes les notes d'une promotion d'étudiants :

```
def moyennes_par_groupe(notes: [float], nbEtu: [int]) -> [float]:
    """
    :entrée notes: tableau de float
    :entrée nbEtu: tableau d'int
    :pré-cond:
       - la somme des éléments de nbEtu est égale à len(notes)
       - notes[0 : nbEtu[0]] contient les notes du premier groupe ;
         notes[nbEtu[0] : nbEtu[0] + nbEtu[1]] contient les notes du
         deuxième groupe, et ainsi de suite...
    :sortie moy: tableau de float
    :post-cond:
       - len(moy) = len(nbEtu)
       - ∀ i ∈ [0;len(moy)[,  moy[i] contient la moyenne des étudiants
         du (i+1)ème groupe

    >>> moyennes_par_groupes([11.0, 9.0, 10.0, 15.0, 16.0, 2.0, 3.0, 4.0],
                             [3, 2, 3])
    array([10.0, 15.5, 3.0])
    """
```

6. ⋆⋆ Calculer les moyennes par groupe pour un tableau contenant toutes les notes d'une promotion d'étudiants :

```
def moyennes_par_groupe2(notes: [float], groupe: [int], nbGroupe, int) ->␣
↪[float]:
    """
    :entrée notes: tableau de float
    :entrée groupe: tableau d'int
    :entrée nbGroupes: int
    :pré-cond:
       - len(notes) = len(groupe)
       - ∀ i ∈ [0;len(notes)[,
           notes[i] est la note du (i+1)ème étudiant, et
           groupe[i] est le numéro de groupe (entre 0 et nbGroupes-1)
           de cet étudiant
    :sortie moy: tableau de float
    :post-cond:
       - len(moy) = nbGroupes
       - ∀ i ∈ [0;len(moy)[,  moy[i] contient la moyenne des étudiants
         du groupe n° i

    >>> moyennes_par_groupes2([11.0, 15.0, 9.0, 3.0, 10.0, 16.0, 2.0, 4.0],
                              [   0,    1,   0,   2,    0,    1,   2,   2])
    array([10.0, 15.5, 3.0])
    """
```

7. ⋆⋆ Générer toutes les chaînes de caractères de longueur n composées des caractères `'a'` et `'b'` :

```
def mots_ab(n: int) -> [str]:
    """
    :entrée n: int
    :pré-cond: n ≥ 0
    :sortie mots: tableau de str
    :post-cond: mots contient toutes les chaînes de caractères de
                longueur n composées des caractères 'a' et 'b'.

    >>> # l'ordre peut être différent
    >>> mots_ab(0)
    array([""])
    >>> mots_ab(1)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    array(["a", "b"])
    >>> mots_ab(2)
    array(["aa", "ab", "ba", "bb"])
    >>> mots_ab(3)
    array(["aaa", "aab", "aba", "abb", "baa", "bab", "bba", "bbb"])
    >>> mots_ab(4)
    array(["aaaa", "aaab", "aaba", "aabb", "abaa", "abab", "abba", "abbb",
           "baaa", "baab", "baba", "babb", "bbaa", "bbab", "bbba", "bbbb",
         ])
    """
```

NB : pour n = 0, la réponse est constituée de l'unique chaîne `""` (qui a bien pour longueur 0, et qui ne contient aucun autre caractère que `'a'` ou `'b'`).

Bien que ce ne soit pas une obligation, cet algorithme est plus simple à écrire sous forme récursive qu'itérative.

8. ⋆⋆ Générer toutes les chaînes de caractères de longueur n composées d'un ensemble de caractères donnés :

```
def mots(chars: str, n: int) -> [str]:
    """
    :entrée chars: str
    :entrée n: int
    :sortie mots: tableau de str
    :pré-cond: n ≥ 0, len(chars) > 0,
               tous les caractères de chars sont différents
    :post-cond: mots contient toutes les chaînes de caractères de
                longueur n composées des caractères de chars.

    >>> mots("abc", 0)
    array([""])
    >>> mots("abc", 1)
    array(["a", "b", "c"])
    >>> mots("abc", 2)
    array(["aa", "ab", "ac", "ba", "bb", "bc", "ca", "cb", "cc"])
    >>> mots("XyZ", 2)
    array(["XX", "Xy", "XZ", "yX", "yy", "yZ", "ZX", "Zy", "ZZ"])
    """
```

On pourra s'inspirer de la solution de *mots_ab* (page 27), dont cet algorithme est une généralisation.

4.5 Modification de tableau

1. Modifier le tableau a de sorte à inverser l'ordre de ses éléments :

```
def inverse_tableau(a: [float]):
    """
    :e/s a: tableau de float
    :post-cond: ∀ i ∈ [0;len(a)[,  aₑ[i] == aₛ[len(a)-1-i]

    >>> a = array([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0])
    >>> inverse_tableau(a)
    >>> a
    array([1.0, 7.0, 3.0, 7.0, -3.0, 1.0, 2.0])
    """
```

2. Insèrer une valeur dans un tableau, en décalant les autres valeurs vers la droite :

```
def insere_val(a: [float], v: float, i: int):
    """
    :e/s a: tableau de float
```

(suite sur la page suivante)

(suite de la page précédente)

```
    :entrée v: float
    :entrée i: int
    :pré-cond: 0 ≤ i < len(a)
    :post-cond: ∀ j ∈ [0;len(a)[,
                aₛ[j] == aₑ[j]   si j < i
                aₛ[j] == v       si j = i
                aₛ[j] == aₑ[j-1] si j > i
    >>> a = array([2.0, 1.0, -3.0, 7.0, 3.0, 7.0, 1.0])
    >>> insere_val(a, 9.0, 2)
    >>> a
    array([1.0, 7.0, 9.0, 3.0, 7.0, -3.0, 1.0])
    """
```

NB : le tableau ne change pas de taille, donc cette procédure perd la valeur initialement la plus à droite du tableau.

4.6 Tableaux triés

1. ⋆ Trouver l'indice de la valeur v dans un tableau trié :

```
def indice_gauche(a: [float], v: float) -> int:
    """
    :entrée a: tableau de float
    :entrée v: float
    :pré-cond: a est trié:  ∀ i ∈ [1;len(a)[,  a[i] ≥ a[i-1]
    :sortie ival: int
    :post-cond: si v ∈ a,  a[ival] = v
                sinon      ival = -1
    """
```

Si v a plusieurs occurences dans le tableau, on ne spécifie pas l'indice de laquelle doit être retourné. En revanche, il faut tirer partie du fait que le tableau est trié pour trouver *rapidement* le résultat[3].

2. ⋆ Insèrer une valeur à sa place dans un tableau trié, en décalant les autres valeurs vers la droite :

```
def insere_val(a: [float], v: float) -> int:
    """
    :e/s a: tableau de float
    :entrée v: float
    :pré-cond: a est trié:  ∀ i ∈ [1;len(a)[,  a[i] ≥ a[i-1]
    :sortie ival: int
    :post-cond: a est toujours trié ;
                ∀ i ∈ [0;len(a)[,
                aₛ[i] == aₑ[i]   si i < ival
                aₛ[i] == v       si i = ival
                aₛ[i] == aₑ[i-1] si i > ival
    """
```

NB : le tableau ne change pas de taille, donc cette procédure perd la valeur initialement la plus à droite du tableau.

3. Supprimer tous les doublons d'un tableau trié :

```
def suppression_doublons(a: [float]) -> int:
    """
    :e/s a: tableau de float
    :sortie newlen: int
    :pré-cond: a est trié:  ∀ i ∈ [1;len(a)[,  a[i] ≥ a[i-1]
    :post-cond:
```

(suite sur la page suivante)

3. Pour cela on effectuera une recherche dichotomique : à chaque étape, on regarde au milieu de la plage d'indice considérée, et selon que la valeur trouvée est plus grande ou plus petite que v, on n'en considère plus que la moitié gauche ou la moitié droite.

(suite de la page précédente)

```
        - 0 ≤ newlen ≤ len(a)
        - ∀ i ∈ [0;len(a)[, ∃ j ∈ [0;newlen[,  aₛ[j] = aₑ[i]
        - ∀ i ∈ [1;newlen[, aₛ[i-1] < aₛ[i]
    """
```

NB : le tableau ne change pas de taille, mais $newlen$ est la nouvelle longueur « utile » du tableau, c'est à dire l'indice à partir duquel les valeurs dans a_s ne sont plus pertinentes.

4. Copier les valeurs de deux tableaux triés dans un troisième tableau de sorte à ce que ce dernier soit trié lui aussi :

```
def interclassement(a1: [float], a2: [float]) -> [float]:
    """
    :entrée a1: tableau de float
    :entrée a2: tableau de float
    :pré-cond: a1 est trié:  ∀ i ∈ [1;len(a1)[,  a1[i] ≥ a1[i-1]
               a2 est trié:  ∀ i ∈ [1;len(a2)[,  a2[i] ≥ a2[i-1]
    :sortie a3: tableau de float
    :post-cond:
       - len(a3) = len(a1) + len(a2)
       - a3 est trié:  ∀ i ∈ [1;len(a3)[,  a3[i] ≥ a3[i-1]
       - a3 contient toutes les valeurs de a1 et a2
    """
```

CHAPITRE 5

Récursivité

Tout algorithme impliquant une répétition peut s'écrire de deux manière : avec une boucle, ou comme une fonction récursive.

Il n'y a donc pas d'exercice spécifique dans ce chapitre, vous pouvez reprendre tous les exercices du manuel (notamment des chapitres *Chaînes de caractères* (page 13) et *Tableaux* (page 21)) en vous *interdisant l'usage des boucles*, ce qui vous conduira, au besoin, à l'écrire de manière récursive.

Notez que, dans certains cas, la fonction demandée pour un exercice ne se prête pas directement à une écriture récursive, mais suppose l'écriture d'une *fonction intermédiaire*, acceptant plus de paramètres que l'originale. Par exemple, pour écrire la fonction *compte_car* (page 13), il peut être utile de définir la fonction récursive suivante :

1. Calculer le nombre d'occurrences d'un caractère :

```
def compte_car_depuis(s: str, c: str, pos: int) -> int:
    """
    :entrée s: str
    :entrée c: str
    :entrée pos: int
    :pré-cond: len(c) == 1
    :sortie n: int
    :post-cond: n est le nombre de valeurs i >= pos telles que s[i] == c
    """
```

Exemple :

```
compte_car_depuis("hello", "l", 0) # retourne 2
compte_car_depuis("hello", "l", 2) # retourne 2
compte_car_depuis("hello", "l", 3) # retourne 1
compte_car_depuis("hello", "l", 4) # retourne 0
compte_car_depuis("hello", "z", 0) # retourne 0
```

CHAPITRE 6

Tris

Dans ce chapitre, nous nous intéressons à différentes manières d'implémenter le même algorithme, dont l'objectif est de trier un tableau :

```
def tri(a: [float]):
    """
    :e/s a: tableau de float
    :post-cond:
       - ∀ i ∈ [0;len(a)[, ∃ j ∈ [0;len(a)[,  aₛ[j] = aₑ[i]
         (les éléments de a ont simplent changé d'ordre)
       - ∀ i ∈ [1;len(a)[,  aₛ[i] ≥ aₛ[i-1]
         (les éléments de a sont triés par valeur croissante)
    """
```

6.1 Tri par sélection

Ce tri consiste à placer chaque élément du tableau à sa position définitive, du plus petit au plus grand :

1. on recherche le plus petit élément du tableau, et on l'échange avec le premier élément, puis
2. on recherche le plus petit élément entre le deuxième et le dernier, et on l'échange avec le deuxième élément, puis
3. on recherche le plus petit élément entre la troisième et la dernier, et on l'échange avec le troisième élément,
4. etc.

Pour cela, on pourra utiliser la fonction *indice_min* (page 22), lui passant à chaque étape le sous-tableau approprié. Attention cependant à interpréter correctement la valeur de retour de cette fonction : c'est un indice du *sous-tableau*. Par exemple, si `indice_min(a[3:])` retourne la valeur 5, c'est que la valeur recherchée est `a[8]` (8 = 3 + 5).

Complexité

Combien d'étapes de calcul sont elles nécessaires

— dans le pire des cas ?
— lorsque le tableau est déjà trié ?
— lorsque le tableau est trié dans l'ordre décroissant ?

Voir aussi

Avertissement : L'algorithme mis en œuvre dans la deuxième vidéo ci-dessus n'est pas exactement le même que celui décrit ci-dessus : lors de la recherche du minimum, certains éléments sont déplacés dans le tableau… Mais le principe général reste le même.

6.2 Tri par insertion

Ce tri consiste à insérer successivement chaque valeur du tableau dans un sous-tableau déjà trié :

1. au départ, le sous-tableau trié est constitué uniquement du premier élément du tableau ;
2. on insère alors le deuxième élément, c'est à dire que
   — s'il est plus grand que le premier, on ne change rien,
   — s'il est plus petit que le premier, on intervertit leurs positions,
   et ainsi, le sous-tableau constitué des *deux* premiers éléments est désormais trié ;
3. à chaque étape, on augmente d'un élément le sous-tableau déjà trié en décalant vers la gauche l'élément suivant jusqu'à sa position correcte dans le sous-tableau déjà trié ;
4. lorsqu'on a répété cette étape jusqu'au dernier élément du tableau, celui-ci est intégralement trié.

Pour cela, on aura besoin d'une procédure intermédiaire qui, étant donné un tableau partiellement trié, décale le premier élément non-trié pour l'insérer à sa position correcte dans le sous-tableau déjà trié :

```
def insere_element(a: [float], i: int):
    """
    :e/s a: tableau de float
    :entrée i: int
    :pré-cond:
       - 1 ≤ i < len(a)
       - ∀ j ∈ [1;i[,  a[j] ≥ a[j-1]  (a est trié entre 0 et i-1)
    :post-cond:
       - ∀ j ∈ [0;i+1[, ∃ k ∈ [0;i+1[,  aₛ[k] = aₑ[j]
         (les éléments entre 0 et i+1 ont simplement changé d'ordre)
       - ∀ j ∈ [i+1;len(a)[,  aₛ[j] = aₑ[j]
         (les éléments au delà de i n'ont pas été modifiés)
       - ∀ j ∈ [1;i+1[,  a[j] ≥ a[j-1]
         (a est trié entre 0 et i)
    """
```

Complexité

Combien d'étapes de calcul sont elles nécessaires
— dans le pire des cas ?
— lorsque le tableau est déjà trié ?
— lorsque le tableau est trié dans l'ordre décroissant ?

Voir aussi

6.3 Tri binaire

Également appelé tri rapide (*quicksort*), ce tri utilise le principe de la dichotomie. Il consiste à choisir une valeur *pivot* dans le tableau, puis à permuter les éléments de sorte que toutes les valeurs plus petites que le pivot soient à sa gauche, et que toutes les valeurs plus grandes que le pivot soient à sa droite. On trie ensuite récursivement les deux sous-tableaux à gauche et à droite du pivot.

Notons que le choix du pivot n'a pas d'influence sur le résultat de l'algorithme (même s'il peut en avoir sur ses performances). Aussi par souci de simplicité, on prendra comme pivot le premier élément du tableau.

Afin de pouvoir trier récursivement des sous-tableaux, on utilisera les notations `a[:i]` et `a[i:]`.

Il sera également utile de définir la procédure `partitionne` qui choisit un pivot dans un sous-tableau donné, permute les valeurs comme indiqué ci-dessus et retourne l'indice du pivot dans le sous-tableau ainsi permuté :

```
def partitionne(a: [float]) -> int:
    """
    :e/s a: tableau de float
    :sortie ip: int
    :post-cond:
       - 0 ≤ ip < len(a)
       - ∀ i ∈ [0;len(a)[, ∃ j ∈ [0;len(a)[,  aₛ[j] = aₑ[i]
         (les éléments de a ont simplement changé d'ordre)
       - ∀ i ∈ [0;ip], aₛ[i] ≤ aₛ[ip]
         (les éléments à gauche du pivot lui sont inférieurs ou égaux)
       - ∀ i ∈ ]ip;len(a)[, aₛ[i] > aₛ[ip]
         (les éléments à droite du pivot lui sont supérieurs)
    """
```

Voir aussi

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheQuickSort.htmlm

6.4 Tri selon d'autres fonctions de comparaison

Les algorithmes de tris ci-dessus ne se limitent pas aux nombres flottants. On peut bien sûr les appliquer à l'identique sur n'importe quel type de données supportant les opérateurs de comparaison (==, <, >, etc.) comme `int` ou `str`.

Mais on peut appliquer ces algorithmes à d'autres types de comparaison ; par exemple, on pourrait souhaiter trier un tableau de chaînes de caractères selon la longueur de ces chaînes (et non selon leur ordre « naturel » induit par les opérateur < et >). Le principe des algorithmes ne change pas, ce n'est que la manière de comparer les éléments des tableaux qui change.

Appliquez l'un des algorithmes ci-dessus pour écrire les algorithmes suivant :

```
def tri_dec(a: [float]):
    """
    :e/s a: tableau de float
    :post-cond:
       - ∀ i ∈ [0;len(a)[, ∃ j ∈ [0;len(a)[,  aₛ[j] = aₑ[i]
         (les éléments de a ont simplent changé d'ordre)
       - ∀ i ∈ [1;len(a)[,  aₛ[i] ≤ aₛ[i-1]
         (les éléments de a sont triés par valeur décroissante)
    """

def tri_par_longueur_croissante(a: [str]):
    """
    :e/s a: tableau de str
    :post-cond:
       - ∀ i ∈ [0;len(a)[, ∃ j ∈ [0;len(a)[,  aₛ[j] = aₑ[i]
         (les éléments de a ont simplent changé d'ordre)
       - ∀ i ∈ [1;len(a)[,  len(aₛ[i]) ≥ len(aₛ[i-1])
         (les éléments de a sont tries par longueur croissante)
    """
```

6.5 Tri indirect

Dans certaines situations, on souhaite pouvoir accéder aux données d'un tableau dans l'ordre (d'où la nécessité d'un tri), mais sans vouloir modifier directement le tableau. Par exemple, les données peuvent être volumineuse, et leur déplacement coûteux en temps. Ou encore, il peut exister plusieurs ordres pertinents pour les données du tableau (par exemple, on peut souhaiter accéder à un tableau de chaînes par longueur croissante, par longueur décroissante, ou dans l'ordre lexicographique).

Dans ces situations, on utilisera un tri *indirect* : le tableau de données ad n'est pas modifié, mais on crée un tableau d'entier ai ayant la même taille, et contenant tous les indices de ad, de sorte que :

$$\forall i \in [1; len(ad)[, \;\; ad[ai[i-1]] \leq ad[ai[i]]$$

En d'autres termes, chaque élément de ai représente un élément de ad, et les éléments de ai sont triés dans l'ordre des éléments *qu'ils représentent*. On peut bien sûr effectuer ce tri indirect avec n'importe lequel des algorithmes vu ci-dessus pour le tri, moyennant une adaptation (pour comparer les $ad[ai[i]]$ et non les $a[i]$).

```
def tri_indirect(ad: []) -> [int]:
    """
    :entrée ad: tableau
    :sortie ai: tableau d'entiers
    :post-cond:
       - len(ai) = len(ad)
       - ∀ i ∈ [0;len(ad)[, ∃ j ∈ [0;len(ai)[,  ai[j] = i
         (ai contient tous les indices de ad)
       - ∀ i ∈ [1;len(ad)[,  ad[ai[i-1]] ≤ ad[ai[i]]
         (les éléments de ai sont triés
          dans l'ordre des éléments de ad qu'ils représentent)
    """
```

CHAPITRE 7

Types abstraits

7.1 Géométrie

Dans cette section, on considère le type abstrait `Vecteur` pour représenter les vecteurs du plan :

```
class Vecteur:
    "type abstrait"

def base_v() -> (Vecteur, Vecteur):
    """
    :post-cond: retourne les deux vecteurs unitaires du plan
    """

def add_v(v1: Vecteur, v2: Vecteur) -> Vecteur:
    """
    :post-cond: retoune v1 + v2
    """

def mult_v(r: float, v: Vecteur) -> Vecteur:
    """
    :post-cond: retoune r·v
    """

def produit_scalaire(v1: Vecteur, v2: Vecteur) -> float:
    """
    :post-cond: retoune v1·v2 (le produit scalaire de v1 et v2)
    """
```

1. Créer un vecteur à partir de ses coordonnées :

   ```
   def cree_vecteur(x: float, y: float) -> Vecteur:
       """
       :post-conf: retourne le vecteur de coordonnées cartésiennes (x, y)
       """
   ```

   N'importe quel vecteur peut être créé à l'aide des fonctions spécifiées plus haut : si $\overrightarrow{v_x}$ et $\overrightarrow{v_y}$ sont les vecteurs unitaires, alors le vecteur $\overrightarrow{v}$ de coordonnées (x, y) est égal à $x \cdot \overrightarrow{v_x} + y \cdot \overrightarrow{v_y}$.
2. Calculer les coordonnées cartésiennes d'un vecteur :

```
def coordonnees(v: Vecteur) -> (float, float):
    """
    :post-cond: retourne x et y les coordonnées cartésiennes de v
    """
```

On rappelle que les coordonnées x et y d'un vecteur $\overrightarrow{v}$ sont en fait le produit scalaire de $\overrightarrow{v}$ avec les vecteurs unitaires $\overrightarrow{v_x}$ et $\overrightarrow{v_y}$, respectivement.

3. Calculer la norme (la longueur) d'un vecteur :

```
def norme(v: Vecteur) -> float:
    """
    :post-cond: retourne la norme du vecteur v
    """
```

On peut utiliser pour cela les *coordonnées* (page 37) du vecteurs, ou le fait que la norme est égale à la racine carrée du produit scalaire du vecteur avec lui même.

4. Calculer si deux vecteurs sont orthogonaux :

```
def orthogonaux(v1: Vecteur, v2: Vecteur) -> bool:
    """
    :post-cond: retourne True ssi v1 et v2 sont orthogonaux
    """
```

5. Calculer si deux vecteurs sont colinéaires :

```
def colineaires(v1: Vecteur, v2: Vecteur) -> bool:
    """
    :post-cond: retourne True ssi v1 et v2 sont colinéaires
    """
```

6. Implémenter le type abstrait `Vecteur` (c'est à dire les quatre fonctions définies au début de cette section) à l'aide d'un tableau de deux nombres flottants, représentant les coordonnées cartésiennes du vecteur.
7. ⋆⋆ Implémenter le type abstrait `Vecteur` à l'aide d'un tableau de deux nombres flottants, représentant les coordonnées polaires du vecteur, c'est à dire sa norme et l'angle (orienté) qu'il forme avec l'axe des x.

7.2 Date

Dans cette section, on considère le type abstrait `Date` pour représenter les dates du calendrier :

```
class Date:
    "type abstrait"

def cree_date(annee: int, mois: int, jour: int) -> Date:
    """
    :pré-cond:
       - annee > 0
       - 1 ≤ mois ≤ 12
       - 1 ≤ jour ≤ jours_par_annee_mois(annee, mois)
    :post-cond: retourne la Date correspondant aux entrées fournies
    """

def amj(d: Date) -> (int, int, int):
    """
    :post-cond: retounr (a, m, j) où
                a est l'année de la date d
                m est le numéro du mois (entre 1 et 12) de la date d
                j est le numéro du jour (entre 1 et 31) de la date d
    """
```

(suite sur la page suivante)

(suite de la page précédente)

```
def diff_dates(d1: Date, d2: Date) -> int:
    """
    :post-cond: retourne diff tel que
                |diff| est le nombre de jours qui séparent d1 et d2;
                diff est positif si d1 est après d2, négatif sinon.
    """

def add_jours(d: Date, j: int) -> Date:
    """
    :post-cond: si j ≥ 0, retourne la date située j jours après d,
                sinon, retourne la date située -j jours avant d.
    """
```

1. Implémenter le type abstrait `Date` en utilisant un tableau de trois entiers, représentant respectivement l'année, le mois et le jour.
   Vous pourrez réutiliser les fonctions suivantes (proposées au chapitre *Durées et dates* (page 4)) :
   — *jours_par_annee* (page 6)
   — *jours_par_annee_mois* (page 6)
2. Implémenter le type abstrait `Date` en utilisant un entier, représentant le nombre de jours entre cette date et le 1er janvier 1900.
   Vous pourrez réutiliser les fonctions suivantes (proposées au chapitre *Durées et dates* (page 4)) :
   — *jours_par_annee* (page 6)
   — *jours_par_annee_mois* (page 6)
3. Comparez les deux implémentations proposées ci-dessus pour le type abstrait `Date`. Lequel vous semble le plus judicieux ?

7.3 Pile

Dans cette section, on considère un type abstrait `Pile` possédant les méthodes suivantes :

```
class Pile:
    "type abstrait"

def cree_pile(n: int) -> Pile:
    """
    :post-cond: retourne p vérifiant pile_vide(p) == True
                p peut contenir au moins n éléments
    """

def pile_vide(p: Pile) -> bool:
    """
    :post-cond: retourne True si et seulement si la Pile est vide
    """

def pile_pleine(p: Pile) -> bool:
    """
    :entrée p: Pile
    :sortie v: bool
    :post-cond: retourne True si et seulement si la Pile est pleine
    """

def sommet(p: Pile) -> int:
    """
    :pré-cond: pile_vide(p) == False
    :post-cond: retourne la valeur au sommet de la pile
    """
```

(suite sur la page suivante)

(suite de la page précédente)

```
def empile(p: Pile, v: int):
    """
    :e/s p: Pile
    :pré-cond: pile_pleine(p) == False
    :post-cond: v est ajouté au sommet de p
    """

def depile(p: Pile):
    """
    :e/s p: Pile
    :pré-cond: pile_vide(p) == False
    :post-cond: la valeur au sommet de p est retirée
    """
```

1. Implémenter le type abstrait `Pile` en utilisant un tableau d'entiers a, tel que
   — `a[0]` contient le nombre d'éléments n contenus dans la pile,
   — ∀ i ∈ [1 ;n], `a[i]` est le i-ème élément de la pile (`a[n]` est le sommet),
   — ∀ i ∈ [n ;len(a)[, `a[i]` n'est pas utilisé (sa valeur est indéterminée).
2. Déterminer si une chaîne de caractères est bien parenthésée, avec deux types de parenthèses :

```
def bien_parenthesee_2(txt: str) -> bool:
    """
    :post-cond: retourne True si et seulement si txt est bien parenthésée
    """
```

Cette chaîne de caractères contient des parenthèses rondes (`(` et `)`) et carrées (`[` et `]`). À chaque parenthèse ouvrante doit correspondre une parenthèse fermante du même type, et réciproquement. Par ailleurs, si une parenthèse ouvrante est ouverte à l'intérieur d'un autre couple de parenthèses, sa parenthèse fermante doit elle aussi de trouver à l'intérieur du même couple.

Le tableau ci-dessous donne des exemples de chaînes bien et mal parenthésées.

Bien parenthésées	Mal parenthésées
`abc`	`(`
`(abc)`	`abc)`
`ab[cd]ef`	`ab)c`
`a[b]c(d)e`	`a(b]c`
`a((b)c)d`	`a(b(c)d`
`a(b[c()e]f)g`	`a(b[c)d]e`

Cet algorithme est une généralisation de *bien_parenthesee* (page 16), mais l'existence de deux types de parenthèses différentes oblige à appliquer une méthode différente. On va parcourir la chaîne de gauche à droite, et mémoriser dans une pile l'ordre et le type des parenthèses ouvertes non encore fermées (par exemple, 1 pour parenthèse ronde, et 2 pour parenthèse carrée). Plus précisément, à chaque parenthèse ouvrante, on empilera son type. À chaque parenthèse fermante, on vérifiera dans la pile qu'elle a bien le type attendu, et si c'est le cas, on peut le dépiler.

NB : cette méthode peut en fait s'appliquer avec un nombre arbitrairement grand de types de parenthèses. On l'utilise notamment avec des documents structurés de type HTML, ou les balises sont autant de types de « parenthèses ».