Structures séquentielles et itération

UE Informatique, M1 Mathématiques et applications,
EADS, Université d'Aix-Marseille *

2024–2025

Une structure de données est une convention pour représenter un certain type d'information en vue d'un traitement informatique. Plus explicitement, on peut distinguer deux niveaux :

- à un niveau abstrait, il s'agit de spécifier une structure au sens mathématique du terme, c'est-à-dire l'ensemble des valeurs qu'on souhaite représenter, et les opérations dont on le munit ;
- à un niveau concret, il s'agit d'expliciter :
  - la manière dont les éléments de la structure sont représentés (*in fine*, par un codage comme un mot binaire écrit dans la mémoire de l'ordinateur, elle-même vue comme un mot binaire arbitrairement grand ; mais le plus souvent, en s'appuyant sur des structures de données plus basiques, qu'on a définies auparavant) ;
  - la manière dont les opérations sont réalisées sur cette représentation (soit directement sur les mots binaires, soit en termes d'opérations sur les structures de données intermédiaires).

En général on spécifie aussi certaines exigences d'efficacité pour les opérations abstraites, qui doivent être satisfaites par la réalisation concrète. Le choix une structure abstraite appropriée pour une tâche peut permettre de simplifier la conception d'un algorithme et son étude ; le choix d'une réalisation efficace permet d'optimiser les ressources utilisées en pratique (temps d'exécution et espace occupé en mémoire).

On a vu que les suites finies, ou mots, jouent un rôle central dans la représentation des données en informatique. Ceci se reflète dans la multitude de structures de données qui permettent de représenter les mots et de les manipuler avec plus ou moins d'efficacité : on parle de structures *séquentielles*, auxquelles ce chapitre est consacré.

1 Préliminaire : éléments de complexité algorithmique

Avant de rentrer dans le vif du sujet, on rappelle ou énonce les concepts basiques de complexité algorithmique. L'efficacité d'un algorithme est mesurée par la quantité de ressources consommées par son exécution : le temps et l'espace. Dans la suite, on s'intéresse principalement au temps d'exécution, mais ces considérations s'appliquent tout aussi bien à la complexité en espace.

Il est difficile de raisonner de manière générale sur des valeurs exactes de temps d'exécution ou de mémoire occupée : les temps d'exécution des instructions peuvent être variables, et la place exacte occupée par un objet en mémoire dépend de l'architecture sur laquelle on travaille. De plus,

plutôt que de mesurer l'efficacité de l'algorithme sur une instance particulière, on s'intéresse aux ressources nécessaires dans le pire cas, ou bien en moyenne, pour une classe d'instances dépendant d'un paramètre : typiquement la taille de l'entrée. Et on étudie comment cette mesure évolue avec le paramètre. Par exemple, si $T(n)$ est le temps d'exécution dans le pire cas pour une entrée de taille n, on s'intéresse au comportement asymptotique de $T(n)$ quand n tend vers l'infini.

On va donc mobiliser des notions classiques en analyse.

1.1 Comparaisons asymptotiques

On rappelle les définitions usuelles de comparaisons asymptotiques. Soient $f, g : \mathbf{N} \to \mathbf{R}^+$:

- on dit que f est *négligeable devant* g si, pour tout $\varepsilon > 0$, on a $f(n) \leq \varepsilon g(n)$ pour tout n suffisamment grand ;
- on dit que f est *dominée par* g s'il existe une constante strictement positive b telle que $f(n) \leq ag(n)$ pour tout n suffisamment grand ;
- on dit que f est *de l'ordre de* g s'il existe deux constantes strictement positives a et b telles que $ag(n) \leq f(n) \leq bg(n)$ pour tout n suffisamment grand ;
- on dit que f est *équivalente à* g si $|f - g|$ est négligeable devant g.

On note :

- $o(g)$ la classe des fonctions négligeables devant g,
- $O(g)$ la classe des fonctions dominées par g,
- $\Theta(g)$ la classe des fonctions de l'ordre de g,
- $f \sim g$ quand f est équivalente à g.

et, par abus de notation, on écrit par exemple $o(g(n))$ plutôt que $o(g)$ pour mettre en avant le paramètre considéré. Par exemple, $f(n) \in o(n^2)$ signifie que la fonction f est négligeable devant la fonction de mise au carré.

On peut alors résumer la plupart des résultats sur les comparaisons asymptotiques avec ces notations :

$$g \in O(g) \qquad O(O(g)) = O(g) \qquad f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$$
$$f \in \Theta(g) \iff f \in O(g) \wedge g \in O(f) \qquad f \in \Theta(g) \iff g \in \Theta(f)$$
$$o(g) \subseteq O(g) \supseteq \Theta(g) \qquad O(o(g)) = o(g) = o(O(g))$$

d'où on déduit, par exemple, $\Theta(o(g)) = o(g)$ ou bien $O(\Theta(g)) = O(g)$. En particulier, en posant $f \sim_\Theta g$ si $f \in \Theta(g)$, on obtient une relation d'équivalence (« être du même ordre »).

Il est également bien connu que $\sim$ est une relation d'équivalence, mais la plupart du temps ce n'est pas une notion pertinente pour mesurer l'efficacité des algorithmes. En effet, on néglige généralement les détails d'architecture en encadrant la complexité des opérations élémentaires par des constantes non spécifiées : c'est l'ordre de grandeur qui nous intéresse.

1.2 Temps d'exécution

Plutôt que de se mesurer en secondes, le temps d'exécution est considéré de manière abstraite : il est de l'ordre du nombre d'instructions élémentaires exécutées (opérations de base, tests, appel de fonction, affectations, *etc.*), qu'on suppose toutes réalisées en temps borné par une (petite) constante.

On note généralement $T(n)$ le temps d'exécution pour les entrées de taille n. On dira que l'algorithme (ou le programme, l'opération, *etc.*) considéré s'exécute en temps :

— *constant* si $T(n) \in O(1)$,
— *linéaire* si $T(n) \in \Theta(n)$,
— *quadratique* si $T(n) \in \Theta(n^2)$,
— *logarithmique* si $T(n) \in \Theta(\ln(n))$,
— *etc.*

On pourra aussi dire qu'il s'exécute, par exemple, en temps *au plus linéaire* si $T(n) \in O(n)$, et en temps *sous-quadratique* si $T(n) \in o(n^2)$.

Par défaut, on considère le pire cas, mais on pourra préciser si on regarde le temps moyen, ou le meilleur cas par exemple. Et si le meilleur et le pire cas sont du même ordre, on dira que la complexité est *homogène*.

Dans certains cas, on considèrera d'autres paramètres que la taille d'entrée, ou plusieurs paramètres (par exemple s'il y a plusieurs arguments), et on pourra dire des choses comme « le temps d'exécution est linéaire en la somme des paramètres » pour signifier que, en notant $T(n)$ le temps d'exécution lorsque la somme des paramètres vaut n, on a $T(n) \in \Theta(n)$.

1.2.1. Exercice (Complexité de l'addition)**.** Vérifiez que l'addition de deux entiers naturels écrits dans une base quelconque se fait en temps linéaire homogène en le maximum des longueurs des écritures. ◇

L'exercice précédent montre qu'il est crucial de bien choisir le paramètre n : la taille d'un nombre, c'est celle de son écriture (c'est-à-dire la taille de sa représentation). On pourrait dire que l'addition sur des entiers naturels de *valeur* inférieure ou égale à n se fait en temps *logarithmique* en n, mais cette dépendance n'est pas franchement pertinente d'un point de vue algorithmique.

2 Tableaux

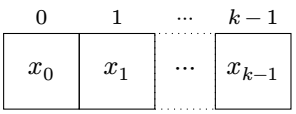
La première structure séquentielle rencontrée en algorithmique est sans doute celle de tableau, qui traduit le plus directement la définition de mot comme suite finie.

2.1 Construction et opérations

On a déjà vu comment, étant donné un codage $\gamma : X \to \mathbf{B}^n$ des éléments de X comme mots de longueur n fixée, on déduit directement un codage

$$\begin{array}{rcccl} \gamma^* & : & X^* & \to & \mathbf{B}^* \\ & & u & \mapsto & \gamma(u[0]) \cdot \cdots \cdot \gamma(u[|u|-1]) \end{array}$$

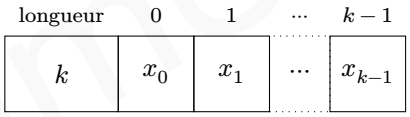
vérifiant $|\gamma^*(u)| = nk = n|u|$.[1] On peut alors voir $u = x_0 \cdots x_{k-1}$ comme un *tableau* à k cases, indexées de 0 à $k-1$, chaque case i contenant l'élément x_i :



Connaissant l'adresse du codage de u en mémoire, on obtient le codage de $u[i] = x_i$, pour chaque $i \in [\![k]\!]$, comme le sous-mot de longueur n, à la position $i \times n$: ceci se fait en temps constant (c'est-à-dire ne dépendant pas de i ni de k). Mais comment savoir si $i \in [\![k]\!]$? Plus précisément, une fois le mot $\gamma^*(u)$ écrit en mémoire, comment retrouver $|u| = k$? Trois solutions :

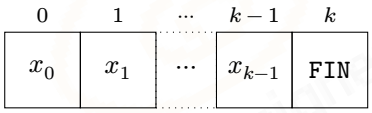
1. On rappelle que $|u|$ est la longueur du mot u.

1. on peut écrire k à un emplacement déterminé de la représentation de u : par exemple si la valeur de k est bornée (disons $k < 2^s$)[2], on peut faire précéder la représentation des symboles par une représentation de k sur s bits :



   — par exemple, en Python, la taille d'un tableau (de type `list`, `tuple`, `bytes`, *etc.*) est toujours stockée quelque part (sans que la spécification de Python indique clairement où), et accessible via la fonction `len()` ;

2. ou bien, s'il existe un code non utilisé (γ n'est pas surjective), disons $\texttt{FIN} \in \mathbf{B}^n \setminus \gamma(X)$, on le réserve pour marquer la fin du tableau :



   — par exemple, en C, c'est la manière standard de représenter les chaînes de caractères : une suite d'octets représentant la chaîne de caractères, terminée par l'octet nul $\underline{0}^8$;

3. ou encore, on peut partir du principe que c'est à l'utilisatrice de gérer cette question, par exemple en maintenant k dans une variable séparée — c'est souvent ce qu'on fait quand on décrit un algorithme sur les tableaux de manière un peu informelle, en utilisant la longueur comme une donnée « magiquement » présente, sans expliquer comment elle est calculée ; c'est aussi l'usage par exemple dans le langage C, où la taille des tableaux est ou bien fournie une bonne fois pour toute lors de la compilation, ou bien laissée à la gestion de la programmeuse.

Notez que dans le cas 2, calculer $|u|$ à partir du codage de u ne se fait pas en temps constant : partant du début du codage de u, il faut compter le nombre de cases parcourues avant d'arriver sur le symbole `FIN`.

Construire un tableau d'éléments de X, de longueur k, c'est donc réserver un espace en mémoire de l'ordre de nk, ce qui est linéaire en k (n étant fixé) : ceci peut se faire par exemple en remplissant chaque case avec un élément fixé (typiquement $\underline{0}^n$), ou bien en copiant du contenu ailleurs en mémoire, chaque fois en temps linéaire (il faut écrire le contenu de chacune des cases) ; alternativement, on peut réserver de l'espace en mémoire sans spécifier ce que contient le tableau, ce qui permet d'économiser le coût de l'initialisation (c'est la pratique usuelle en C, mais à peu près interdit en Python).

En résumé, les tableaux sont des mots, qu'on munit des opérations de base suivantes :

— la création d'un tableau (soit en recopiant son contenu depuis un autre tableau, soit en initialisant ses cases avec une valeur fixée) ;

— le calcul de la taille $|u|$;

— l'obtention de la valeur $u[i]$ lorsque $i \in [\![|u|]\!]$;

— l'assignation d'une nouvelle valeur à $u[i]$;

2. Cette hypothèse n'est pas très exigeante : sur 32 bits, on couvre déjà des tailles de l'ordre de $2^{32} \simeq 4 \times 10^9$.

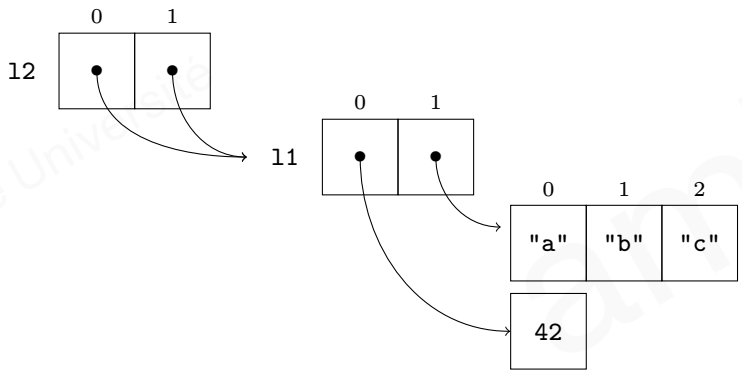
la création se faisant en temps linéaire en la taille, les autres opérations en temps constant. Et lorsqu'on dispose d'un codage de longueur fixée pour les éléments du tableau, on peut satisfaire ces exigences en stockant le contenu des cases de manière contigüe, et en stockant la taille quelque part dans la structure du tableau.

2.1.1. *En Python* . Tous les types séquentiels de Python (`list`, `tuple`, `str`, *etc.*) peuvent être vus comme des tableaux de longueur connue : on obtient la longueur du tableau `t` en évaluant `len(t)`, et `t[i]` donne l'élément numéro `i` lorsque `i` s'évalue en un entier tel que `0<=i<len(t)`. Il y a même un raccourci pour obtenir le contenu d'une case en partant de la fin : si `-len(t)<=i<0`, `t[i]` est un raccourci pour `t[len(t)-i]`.

Pour les types mutables (principalement `list`), on change la valeur de l'élément numéro `i` par l'affectation `t[i]=valeur`. Il faut cependant noter que la représentation en mémoire de `list` et `tuple` n'est pas aussi simple que suggéré ci-dessus : ces tableaux peuvent avoir comme éléments des objets Python quelconques, de taille non fixée à l'avance. Ce qui est stocké dans une case du tableau, ce n'est pas l'objet lui-même, mais son *adresse* en mémoire (la taille des adresses étant fixée, elle), qu'on peut voir comme un *pointeur* vers l'objet. Par exemple, après les deux affectations :

```
l1 = [42,"abc"]
l2 = [l1, l1]
```

on peut représenter la situation en mémoire comme suit :



ce qui explique qu'après l'instruction supplémentaire

```
l1[0] = 0
```

on obtienne `l2 == [[0,"abc"],[0,"abc"]]` : on a changé le contenu de la case 0 de `l1`, et les deux cases de `l2` pointent toujours au même endroit ![3]

Il y a de nombreuses manières de construire des tableaux en Python. Les plus courantes sont :

1. donner les éléments directement, par exemple `[0,1,2]` ;

3. Si le modèle mémoire de Python vous semble encore mystérieux, vous pouvez en premier lieu consulter les premiers paragraphes du chapitre *3. Modèle de données* de la documentation de référence : `https://docs.python.org/fr/3.10/reference/datamodel.html`. L'utilisation d'un outil de débogage visuel, comme Python Tutor (`https://pythontutor.com/python-compiler.html#mode=edit`) est un moyen efficace de comprendre ce qui se passe sur des exemples concrets : n'hésitez pas à tester les exemples au fil de la lecture pour voir ce qui se déroule pas à pas.

2. créer un tableau à partir des éléments fournis par n'importe quel *itérable*,[4] par exemple `list(range(3))` produira aussi la liste `[0,1,2]` ;
3. construire une liste à partir de la liste vide, en ajoutant ses éléments successifs : après

```
t = []
for x in range(3):
    t.append(x*x+2*x+1)
```

la variable `t` contiendra la liste `[1,4,9]` ;[5]
4. construire un tableau `t+u` par concaténation d'autres tableaux `t` et `u` (ce qui implique une copie des contenus de `t` et `u`), voire par répétition d'un tableau donné : si `n` est un entier positif, `n*t` est un tableau du même type que `t`, obtenu en concaténant `n` copies de `t` (typiquement utilisé dans `n* [0]` pour obtenir un tableau de `n` cases, chacune initialisée à `0`) ;
5. utiliser une définition par compréhension c'est à dire quelque chose comme

`[expression for x in iterable]`,
`list(expression for x in iterable)` ou
`tuple(expression for x in iterable)`,

ce qui s'évalue en un tableau (`list` ou `tuple`) contenant les valeurs de l'`expression` lorsque `x` prend les valeurs successives de l'`iterable`, par exemple : `[x*x+2*x+1 for x in range(3)]` s'évalue en `[1,4,9]`.[6]

Les types `bytearray` (mutable), `str` et `bytes` (immuables) sont plus conformes à la présentation des tableaux ci-dessus : les cases de `str` sont des points de caractères Unicode (eux mêmes gérés comme des `str` de longueur 1, comme discuté au chapitre précédent) ; et celles de `bytearray` et `bytes` contiennent des octets, identifiables aux entiers de ⟦256⟧. On manipulera peu ces derniers, dont l'utilisation est sans doute plus restreinte à des usages strictement informatiques.

Le module `array` de la bibliothèque standard fournit plus généralement une représentation efficace des tableaux pour les classes de nombres de taille fixée les plus courantes (entiers sur 1, 2, 4 ou 8 octets, signés ou non signés, et nombres en virgule flottante). La bibliothèque `numpy`[7] généralise ce mécanisme à plus de types encore, et permet en plus de représenter des tableaux multi-dimensionnels, tout en offrant des opérations efficaces (par exemple pour la somme de matrices).

2.2 Parcours et itération

Le traitement algorithmique de base des tableaux consiste à les parcourir de la première à la dernière case, suivant le schéma :

pour x **dans** t**:**
    faire des choses avec x

4. Voir : `https://docs.python.org/fr/3.10/glossary.html#term-iterable`. Ce peut être un autre tableau du même type, ou un objet d'un autre type séquentiel, ou bien un `range`, ou tout autre objet utilisable dans une boucle `for`.
5. Ce mécanisme consiste en fait à construire `t` comme une pile : on y revient plus bas.
6. C'est en fait un cas particulier de la méthode 2, l'expression `expression for x in iterable` définissant un nouvel itérable.
7. `https://numpy.org/`

ou

pour i **de** 0 **à** $|t|$**:**
    faire des choses avec $t[i]$

ce qui se traduit en Python par :

```
for x in t:
    # faire des choses avec x
```

ou bien en :

```
for i in range(len(t)):
    # faire des choses avec t[i]
```

si le traitement implique d'utiliser aussi le numéro `i` de la case courante.

Par exemple, pour calculer la somme des éléments d'un tableau, on pourrait définir :

```
def somme(t):
    somme = 0
    for x in t:
        somme += x
    return somme
```

mais pour calculer une fonction polynomiale, on écrira :

```
def valeur_poly(P,x):
    """Calcule la valeur en `x` du polynôme `P` représenté comme la suite
    de ses coefficients (`P[i]` est le coefficient de degré `i`)."""
    somme = 0
    for i in range(len(P)):
        somme += P[i] * x**i
    return somme
```

(c'est le traitement utilisé pour la représentation gros-boutiste des entiers naturels en base 2 dans le chapitre précédent).

Notez que la copie d'un tableau (par exemple `list(t)`) ou plus généralement l'initialisation d'un tableau à partir de ses valeurs (par exemple une définition par compréhension) sont essentiellement de la même nature qu'un tel parcours, et se font en temps linéaire en la taille du tableau.

2.2.1. Exercice (Itérations)**.** Les tâches suivantes ne devraient pas poser de difficulté :

1. Écrivez une fonction `produit(t)` qui, en supposant que les éléments de `t` sont des nombres, renvoie leur produit (`1` si `t` est vide).
2. Écrivez une fonction `indice_min(t)` qui renvoie l'indice de la plus petite valeur de `t` (le plus petit indice de la plus petite valeur, si le minimum de `t` apparait plusieurs fois). Par exemple, on veut que `indice_min([1,3,0,2])== 2` et `indice_min([1,2,1])== 0`. Le comportement n'est pas défini quand `t` est vide. ◇

Bien sûr ce schéma n'est que le plus simple, et on peut imaginer bien d'autres moyens de parcourir un tableau en vue d'un traitement particulier : en partant de la fin, en n'en parcourant qu'une partie, en le parcourant plusieurs fois, *etc.* Ce ne sont que des variations sur le même thème.

2.2.2. Exercice (Variations sur le thème des itérations)**.**

1. Écrivez une fonction `trouve_seuil(t,seuil)` qui, en supposant que `t` est un tableau de nombres, renvoie le plus petit entier `k` tel que `0 <= k <= len(t)` et tel que la somme des `k` premiers éléments de `t` dépasse ou égale `seuil` (`None` si la somme ne dépasse jamais le seuil).
2. Écrivez une fonction `produit_poly(p,q)` qui, en supposant `p` et `q` sont les coefficients de polynômes comme dans `valeur_poly()` ci-dessus, retourne le polynôme produit (c'est-à-dire la liste de ses coefficients). ◇

2.2.3. *En Python*  . Les exemples et exercices précédents se limitent à une traduction de l'algorithmique de base en Python, mais le langage et sa bibliothèque standard fournissent de nombreuses fonctions et constructions utiles, qui permettent d'alléger le code et le rendre plus lisible et robuste.

Par exemple il y a une fonction `sum` qui s'applique à n'importe quel itérable fournissant une suite de nombres (par exemple `sum(range(10))==45`) : combiné avec une définition par compréhension, ça remplace avantageusement certaines des boucles `for` ci-dessus. Par exemple on pourrait écrire :

```
def valeur_poly(P,x):
    """Calcule la valeur en `x` du polynôme `P` représenté comme la suite
    de ses coefficients (`P[i]` est le coefficient de degré `i`)."""
    return sum(P[i] * x**i for i in range(len(P)))
```

D'autres fonctions utiles du même genre :

— `max(t)` et `min(t)` avec des sens évidents ;

— `all(t)` attendant un itérable `t`, avec `all(t) == True` si tous les éléments fournis par `t` sont vrais (en tant que booléens), et `all(t) == False` si l'un au moins est faux ;

— `any(t)`, similaire à la précédente, en échangeant les quantifications universelle et existentielle ;

— *etc.*

Le comportement de chacune se résume à une boucle `for` évidente. Leur utilisation n'est pas exigée mais, si on les connait, il ne faut pas s'en priver ; et si on tombe dessus dans un exercice ou un exemple, il faut au moins savoir aller chercher leur signification dans la documentation.

2.3 Recherche d'élément

Une tâche parmi les plus basiques est la recherche d'un élément dans un tableau :

2.3.1. Exercice (Première occurrence d'un élément)**.**

1. Écrivez une fonction `indice(t,x)` qui renvoie le premier indice `j` tel que `t[j] == x`, ou bien `None` si `x` n'apparait pas dans `t`.[8]
2. Déduisez-en une fonction `present(t,x)` qui renvoie `True` si `x` apparait dans le tableau `t` et `False` sinon. ◇

8. On pourrait aussi renvoyer une valeur comme `-1` à la place de `None` : l'important est d'avoir une valeur indiquant l'absence de `x`, qu'on ne puisse pas confondre avec un indice de case légitime. En ce sens `None` est « moralement » un meilleur choix : c'est la constante Python représentant une absence de valeur (c'est aussi ce qui est retourné par une fonction qui se termine sans `return` explicite, ou bien avec un `return` sans argument). On peut ainsi tester l'absence de valeur retournée, par exemple avec `if indice(t,x)is None:...`. Une autre approche, légitime, serait de soulever une exception lorsque `x` est absent : vous pouvez faire ça si ça vous semble plus correct.

```python
def recherche_dichotomie(t, x):
    """Recherche l'indice de la première occurrence de `x` dans `t` par
    dichotomie, en supposant que `t` est trié (par valeurs croissantes).
    Renvoie `None` si `x` n'apparait pas dans `t`."""

    debut = 0
    fin = len(t)

    # On maintient l'invariant : `0 <= debut <= fin <= len(t)` et,
    # si `x` apparait dans `t`, alors l'indice `i` de sa première occurrence
    # vérifie `debut <= i < fin`.

    while fin > debut:
        longueur = fin - debut
        milieu = debut + longueur // 2
        # Comme `longueur > 0`, `0 <= longueur // 2 < longueur`,
        # donc `debut <= milieu < milieu+1 <= fin`

        # On considère une partition de `t` en trois morceaux :
        #  * de `debut` à `milieu-1` (vide si `début == milieu`)
        #  * la case d'indice `milieu`
        #  * de `milieu+1` à `fin-1` (vide si `milieu+1 == fin`)
        if debut < milieu and t[milieu-1] >= x:
            # On sait que s'il y a un `x` quelque part, le premier
            # se trouve entre `debut` et `milieu-1` (inclus) :
            # on change la `fin`.
            fin = milieu
        elif t[milieu] == x:
            # On sait que si `debut <= i < milieu`, alors `t[i] <= t[milieu-1] < x`
            # donc c'est bien la première occurrence et on termine.
            return milieu
        else:
            # On sait que si `debut <= i < milieu`, alors `t[i] <= t[milieu-1] < x`.
            # De plus `t[milieu] != x`. Donc s'il y a un `x` c'est après `milieu` :
            # on change le `debut`.
            debut = milieu+1

    # Si on arrive ici, c'est qu'on n'a pas trouvé `x`, vu que `fin==debut`.
    return None
```

FIGURE 1 – Recherche d'un élément par dichotomie, version itérative

Ces tâches sont si courantes que Python fournit, là encore, des outils par défaut pour les résoudre : l'expression `x in t` teste si `x` apparait dans `t`, et une méthode `index()` est fournie par la plupart des types itérables (dont les tableaux) de sorte que `t.index(x)` renvoie l'indice de la première occurrence de `x` dans `t`, et soulève une exception s'il n'y en a pas. Dès qu'on sort de ces cas de base, il faut toutefois travailler encore un peu :

2.3.2. Exercice (Recherche d'élément satisfaisant un critère)**.**

1. Écrivez une fonction `indice_critere(t,f)` qui, supposant que `t` est un tableau et `f(x)` est une fonction renvoyant un booléen pour chaque élément potentiel du tableau, renvoie le premier indice `j` tel que `f(t[j])`, ou bien `None` si aucun élément de `t` ne satisfait `f`.
2. Déduisez-en une fonction `present_critere(t,f)` qui renvoie `True` si `t` contient un élément satisfaisant `f` et `False` sinon. Par exemple, pour savoir si `t` contient un élément pair,

```python
def recherche_dichotomie_bornes(t, x, debut, fin):
    """Recherche l'indice de la première occurrence de `x` dans `t`,
    située entre `debut` et `fin-1` (inclus), en supposant que `t` est trié
    (par valeurs croissantes), et `0 <= debut <= fin <= len(t)`.
    Renvoie `None` si `x` n'apparait pas dans cet intervalle."""

    if fin <= debut:
        # L'espace de recherche est vide
        return None

    longueur = fin - debut
    milieu = debut + longueur // 2
    # Comme `longueur > 0`, `0 <= longueur // 2 < longueur`,
    # donc `debut <= milieu < milieu+1 <= fin`

    # On considère une partition de l'espace de recherche en trois morceaux :
    #  * de `debut` à `milieu-1` (vide si `début == milieu`)
    #  * la case d'indice `milieu`
    #  * de `milieu+1` à `fin-1` (vide si `milieu+1 == fin`)
    if debut < milieu and t[milieu-1] >= x:
        # On sait que s'il y a un `x` dans l'intervalle, le premier
        # se trouve entre `debut` et `milieu-1` (inclus).
        return recherche_dichotomie_bornes(t, x, debut, milieu)
    elif t[milieu] == x:
        # On sait que si `debut <= i < milieu`, alors `t[i] <= t[milieu-1] < x`
        # donc c'est bien la première occurrence dans l'intervalle.
        return milieu
    else:
        # On sait que si `debut <= i < milieu`, alors `t[i] <= t[milieu-1] < x`.
        # De plus `t[milieu] != x`. Donc s'il y a un `x` dans l'intervalle
        # c'est après `milieu`.
        return recherche_dichotomie_bornes(t, x, milieu+1, fin)

def recherche_dichotomie(t, x):
    return recherche_dichotomie_bornes(t, x, 0, len(t))
```

FIGURE 2 – Recherche d'un élément par dichotomie, version récursive

on pourrait appeler `present(t, lambda x:x %2 == 0)`.[9] ◇

Dans les cas précédents, la recherche se fait en temps linéaire : faute de structure particulière, on n'a pas d'autre choix que de parcourir tout le tableau. Si on a plus d'information sur la structure du tableau, on peut parfois faire mieux. Le cas emblématique est celui d'un tableau trié : ici on peut appliquer une recherche par *dichotomie*, en divisant la fenêtre de recherche par deux à chaque itération.

Le code de la figure 1 (c'est long, mais il y a bien plus de commentaire que de code) présente une mise en œuvre de cette approche. Dans cette version, on a favorisé un style *itératif*, c'est-à-dire qu'on a présenté l'algorithme principal comme une boucle, en faisant varier les bornes `debut` et `fin` à chaque itération. On peut tout aussi bien en définir une version récursive : voir la figure 2. Suivant la formation ou l'habitude, on peut comprendre mieux l'une ou l'autre version, mais il devrait être clair que c'est essentiellement une affaire de style : les deux fonctions appliquent le même principe de dichotomie, et explorent le tableau rigoureusement de la même manière.

9. Au cas où vous n'auriez jamais croisé cette construction : l'expression `lambda x: corps_de_fonction` définit une fonction *anonyme*, qui attend un argument `x` et retourne la valeur de l'expression `corps_de_fonction`. Voir : `https://docs.python.org/fr/3.10/glossary.html#term-lambda`.

```python
def sousmot_en_position(u,v,i):
    """Renvoie `True` si `u` contient une copie de `v` à la position `i`,
    `False` sinon. On suppose que `i >= 0` et `i + len(v) <= len(u)
    (le comportement n'est pas défini sinon)."""
    for j in range(len(v)):
        if v[j] != u[i+j]:
            return False
    return True

def position_sousmot(u,v):
    """Renvoie la position de la première copie de `v` dans `u`,
    ou bien `None` s'il n'y en a pas."""
    for i in range(len(u)-len(v)+1):
        if sousmot_en_position(u,v,i):
            return i
    return None
```

FIGURE 3 – Recherche de sous-mot en Python

2.3.3. Exercice (Complexité logarithmique de la recherche dans un tableau trié)**.** Vérifiez que si `len(t)` $< 2^n$ alors, dans un appel à `recherche_dichotomie(t,x)`, le corps de la boucle `while` est itéré au plus n fois (première version), ou bien le nombre d'appels à la fonction auxiliaire `recherche_dichotomie_bornes` est borné par n (deuxième version). ◇

Donc la recherche d'un élément dans un tableau trié se fait en temps logarithmique : d'où l'intérêt de trier…

2.4 Recherche de sous-mots

Une variante du problème précédent est de rechercher les occurrences d'un sous-mot : étant donnés deux mots u et v, on se demande si v apparait comme un sous-mot de u et, si oui, à quel endroit ? Un algorithme naïf consiste à chercher à toutes les positions possibles : on imagine une fenêtre de longueur $|v|$, qui parcourt u de gauche à droite, à la recherche d'une occurrence de v. On en donne une réalisation en Python, en figure 3.

2.4.1. Exercice. Évaluez le temps d'exécution de `position_sousmot(u,v)` lorsque `u == (n+k)*"a"` et `v == k*"a"+b`, en fonction de `n` et `k` (disons, en comptant le nombre d'itérations de chacune des deux boucles). Vérifiez que c'est le pire cas possible pour des mots de cette longueur. ◇

On peut faire (un peu) mieux : le problème est discuté en détail sur la page `https://fr.wikipedia.org/wiki/Algorithme_de_recherche_de_sous-chaîne`[W] : en travaillant d'abord sur une représentation efficace du motif recherché et de ses redondances, on obtient un fonctionnement linéaire en la somme des longueurs de `u` et `v`, plutôt que leur produit.

2.4.2. *En Python* . Pour les chaînes de caractères, on peut tester la présence d'une sous-chaîne avec `u in v` : par exemple `"bcd"in "abcd"== True`. Et on peut obtenir l'indice de début avec la méthode `find()` : par exemple `"abcd".find("cd")==2`. Cette méthode renvoie `-1` si la sous-chaîne n'apparait pas. Dans les deux cas, l'algorithme utilisé par Python est un subtil florilège des meilleures méthodes connues. [10]

Pour les autres types séquentiels, rien de tel n'est fourni, il faut donc programmer soi-même :

10. Voir les commentaires au début du fichier source `https://github.com/python/cpython/blob/main/Objects/stringlib/fastsearch.h`.

sauf réel besoin de performance (mais alors Python est-il le bon outil ?), l'algorithme naïf ne fonctionne pas si mal…

3 Listes, piles, files

La structure de tableau est la traduction la plus directe de la notion de mot comme suite finie de symboles. Mais elle n'est pas forcément adaptée pour refléter toute la structure du monoïde des mots X^*.

3.1 Listes

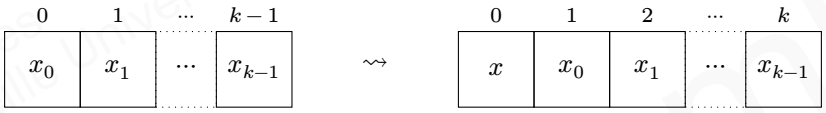
Tout mot peut être engendré à partir du mot vide par adjonction successive d'un symbole en tête : étant donnés $x \in X$ et $u \in X^k$, on forme $x \cdot u \in X^{k+1}$. C'est l'idée de liste : un liste est ou bien la liste vide, ou bien la concaténation d'un élément (sa *tête*) et du reste de la liste (sa *queue*). Pour refléter cette structure, on voudrait que les opérations suivantes se fassent en temps constant :

— créer une liste vide : ε ;

— former la concaténation $x \cdot u$ (traditionnellement prononcé x *cons* u, comme le début du mot *construction*) ;

— obtenir la tête $\texttt{te}(x \cdot u) = x$ ou la queue $\texttt{qu}(x \cdot u) = u$ d'une liste non vide $x \cdot u$.

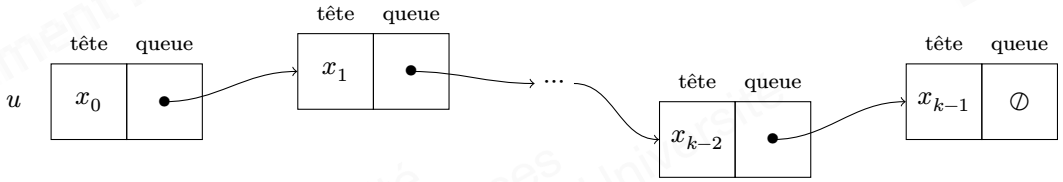
Or la structure de tableau n'est pas bien adaptée pour réaliser ces opérations. Considérons par exemple l'insertion d'un symbole en tête : étant donnés $x \in X$ et $u \in X^k$, on veut former $x \cdot u \in X^{k+1}$:



ce qui est problématique si on retient que le mot est représenté comme un tableau écrit quelque part en mémoire : il faut s'assurer qu'on peut libérer une case supplémentaire au début (ce qui est faux en général, et mal géré dans la plupart des systèmes pour des raisons d'efficacité de l'adressage) ou à la fin (ce qui est tout aussi faux en général). La méthode standard est donc plutôt de créer un nouveau tableau, de longueur $k+1$, de coller x dans la première case, et recopier les suivantes : ceci se fait en temps linéaire en la taille du tableau, alors qu'on a ajouté une seule case, c'est cher !

La structure concrète de *liste chaînée* est une réponse possible. L'idée est de stocker chaque élément de la liste séparément en mémoire, tout en lui adjoignant un *pointeur* (c'est-à-dire le codage d'une adresse) vers l'élément suivant, s'il en existe un — sinon on utilise un pointeur spécial, disons ⌀. Une liste non vide est alors entièrement déterminée par l'adresse du premier élément, sa tête, munie d'un pointeur vers la queue.

Graphiquement, on peut voir la liste chaînée représentant $u = x_0 \cdots x_{k-1}$ comme :



```python
class Liste:
    """Classe abstraite pour les listes chaînées :'
    lattribut `vide` indique si la liste est vide ;
    on peut obtenir la tête ou la queue 'dune liste non vide."""
    def te(self):
        if self.vide:
            raise ValueError("La liste vide n'a pas de tête.")
        return self.tete
    def qu(self):
        if self.vide:
            raise ValueError("La liste vide n'a pas de queue.")
        return self.queue

class Vide(Liste):
    """Classe pour la liste vide."""
    def __init__(self):
        self.vide = True
    def __str__(self):
        return 'Vide()'

class Cons(Liste):
    """Classe pour les listes non-vides."""
    def __init__(self, x, u):
        self.vide = False
        self.tete = x
        self.queue = u
    def __str__(self):
        return f"Cons({self.tete},{self.queue})"
```

FIGURE 4 – Une classe pour les listes chaînées en Python

chaque élément étant stocké de manière indépendante. Les opérations listées plus haut se font alors effectivement en temps constant.

On peut représenter les listes en python de manière très basique, en utilisant des `tuple`s :

```python
vide = ()
def cons(x,u):
    return (x,u)
def te(u):
    x,_ = u
    return x
def qu(u):
    _,v = u
    return v
```

Après ces définitions, on représente par exemple le mot `"ab"` comme `cons("a",cons("b",vide))` : avec `u == cons("a",cons("b",vide))`, on a `te(qu(u))== "b"`, `qu(qu(u))== vide`, et `te(qu(qu(u)))` plante parce qu'on essaie d'extraire la tête d'une liste vide.

On peut bien sûr faire quelque chose de plus abstrait, en définissant des classes, comme en figure 4 : on représente le mot `"ab"` comme `u == Cons("a",Cons("b",Vide))`, et alors on a `m.qu().te()== "b"`, `m.qu().qu().vide ==True`, et `m.qu().qu().te()` plante.

3.1.1. Exercice (Listes chaînées en Python).

1. Définissez une fonction `longueur_liste(l)` (ou une méthode `Liste.__len__(self)`) qui retourne le nombre d'éléments d'une liste.

2. Définissez une fonction `element_liste(l,i)` (ou une méthode `Liste.__getitem__(self,i)`) qui renvoie l'élément numéro `i` de la liste `l` (en commençant à `0`) : avec `u` comme ci-dessus, on devrait avoir `element_liste(u,1)=="b"`, et `element_liste(u,2)` devrait planter. Que pensez-vous de la complexité de cette opération ?
3. Définissez des fonctions `tableau_vers_liste(t)`, qui transforme un tableau (ou un itérable Python) en une liste, et `liste_vers_tableau(l)` qui transforme une liste au sens ci-dessus en un tableau (par exemple de type… `list`), contenant les mêmes éléments dans le même ordre. ◇

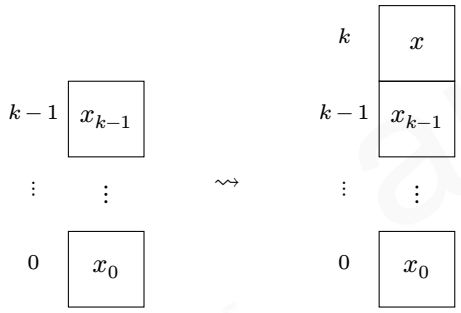
3.2 Piles

Les *piles* (*stack* en anglais) sont une autre structure séquentielle basique, pour laquelle les opérations principales sont aussi l'ajout ou la suppression d'un élément d'un côté fixé du mot. Elles diffèrent des listes par l'utilisation qu'on en fait, qui est de nature plus impérative, et par leur réalisation concrète : l'idée est d'empiler les éléments (en les stockant donc de manière contigüe comme dans un tableau), en les ajoutant ou retirant systématiquement du haut de la pile (comme pour une liste, on ne travaille qu'à un bout du mot, mais plutôt en transformant l'objet en place, plutôt qu'en construisant un nouveau).

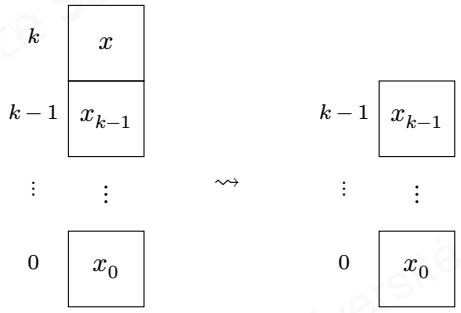
L'état courant d'une pile est donc un mot, vu comme un tableau, mais sur lequel on ne travaillerait que par un bout : disons qu'on pense que le haut de la pile est à la fin du mot. Les opérations sur les piles sont alors les suivantes :

— créer une pile vide : ε ;

— empiler (on dit aussi *pousser*, ou *push* en anglais) un élément x sur le haut d'une pile u, pour la transformer en $u \cdot x$;



— dépiler (on dit *pop* en anglais) l'élément au sommet d'une pile non vide $u \cdot x$, ce qui change la pile en u



et retourne la valeur x.

```python
class Pile:
    def __init__(self):
        """Construit une pile vide, dont les valeurs sont stockées
        dans une liste (la tête est le sommet de la pile)."""
        self.valeurs = None
        self.taille = 0

    def __str__(self):
        return f"Pile(taille: {self.taille}, valeurs: {self.valeurs})"

    def __len__(self):
        return self.taille

    def push(self,x):
        self.valeurs = (x,self.valeurs)
        self.taille += 1

    def pop(self):
        if self.taille == 0:
            raise IndexError("impossible de dépiler un élément d'une pile vide")
        tete, queue = self.valeurs
        self.valeurs = queue
        self.taille -= 1
        return tete
```

FIGURE 5 – Une classe pour les piles en Python

Une manière de réaliser ces opérations, c'est simplement de voir une pile comme une liste : le sommet de la pile est la tête de la liste. À titre d'illustration, une classe pour les piles en Python est donnée en figure 5.

3.2.1. *En Python* . Python fournit en fait directement une solution pour les piles : la classe `list` elle-même ! En effet, tout tableau de ce type est muni des méthodes `append()` (qui ajoute un élément à la fin du tableau) et `pop()` (qui supprime l'élément de la fin du tableau et renvoie sa valeur). Sous le capot, lors de la création d'un tableau Python, la quantité de mémoire allouée est *a priori* supérieur à celle strictement nécessaire, ce qui laisse de la marge pour ajouter des éléments ; et lorsqu'il n'y a plus assez de marge, Python alloue un espace mémoire plus grand — pour une grande pile, le temps d'exécution de la méthode `append()` n'est donc pas vraiment constant, mais c'est en moyenne bien plus efficace que d'allouer de la mémoire séparément pour chaque élément qu'on empile.

Les piles sont une structure de données typiquement utilisée dans les compilateurs et les interpréteurs des langages de programmations : elles servent par exemple à stocker les informations attachées aux appels récursifs de fonctions, d'où le nom de pile d'appel (*call stack*).

3.2.2. Exercice (Notation polonaise inverse)**.** La notation polonaise inverse consiste à présenter le calcul d'une expression, par exemple arithmétique, comme une suite d'opérations à réaliser sur une pile de nombres : celle-ci permet de réaliser les calculs de manière purement séquentielle, sans avoir recours aux parenthèses, ni donner de noms aux résultats intermédiaires. Par exemple, pour calculer $(2 \times 3) + 4$, on pousse 2, on pousse 3, on multiplie (on dépile les deux derniers nombres, et on pousse leur produit), on pousse 4 et on additionne (on dépile les deux derniers nombres, et on pousse leur somme) : le résultat attendu se trouve au sommet de la pile. Les instructions précédentes se notent par exemple comme la liste : `[2,3,"mul",4,"add"]`.

Définissez une fonction `npi(l)` qui attend une liste `l` d'instructions, qui sont des nombres ou

des chaînes, et renvoie le résultat produit de la manière suivante :

— on démarre avec une pile vide, puis on applique les instructions dans l'ordre ;
— pour un nombre, on l'empile ;
— pour l'instruction `"add"`, on dépile les deux derniers éléments, et on empile leur somme ;
— pour l'instruction `"mul"`, on dépile les deux derniers éléments, et on empile leur produit ;
— pour l'instruction `"sub"`, on dépile les deux derniers éléments, et on empile le résultat de la soustraction du dernier à l'avant-dernier ;
— pour l'instruction `"div"`, on dépile les deux derniers éléments, et on empile le résultat de la division de l'avant-dernier par le dernier ;
— pour l'instruction `"aff"`, on affiche le dernier élément (sans le dépiler) ;
— pour l'instruction `"dup"`, on empile une nouvelle copie de l'élément au sommet de la pile ;
— pour l'instruction `"ech"`, on échange les deux éléments au sommet de la pile ;
— pour l'instruction `"nbr"`, on empile la taille de la pile.

Le resultat est le sommet de la pile s'il y en a un, `None` sinon. Par exemple `npi([1,1,"add","aff","dup","mul","aff","dup","mul"])` devrait afficher `2` puis `4`, puis renvoyer `16`. ◇

3.3 Files

La structure de *file* (ou *file d'attente*, *queue* en anglais) se distingue de celle de pile, en ce que c'est le plus ancien élément inséré qui sort de la file en premier (premier arrivé, premier sorti — *first in, first out* en anglais, d'où le nom *FIFO* qu'on trouve parfois dans la littérature).

On veut donc pouvoir :

— créer une file vide : ε ;
— tester si la file est vide ;
— ajouter un élément en queue : $u \rightsquigarrow u \cdot x$;
— faire sortir l'élément de tête, ce qui tranforme $x \cdot u$ en u et renvoie x.

Une manière de traiter cette structure est de placer la file dans un tableau, en mémorisant où se trouvent la tête et la queue de la file :

— le curseur de queue avance quand on ajoute un élément ;
— le curseur de tête avance quand on libère une place ;
— la file est vide quand le curseur de tête rejoint le curseur de queue.

De plus plutôt que d'étendre le tableau lorsque la queue atteint la fin du tableau, on peut profiter que des places se sont libérées en tête, et revenir au début. Le seul cas où il faut étendre le tableau (ou bien planter) est celui où toutes les cases sont pleines. Une proposition de réalisation comme classe Python est en figure 6.

3.3.1. Exercice. On peut représenter une file en utilisant deux piles : l'une pour la queue qu'on remplit avec les nouveaux éléments, l'autre pour la tête, qu'on dépile pour faire sortir un élément de la queue.

1. Quels sont les cas critiques (ceux où les opérations ne se font pas en temps constant) ?
2. Quel avantage voyez-vous à cette approche, comparée à celle présentée ci-dessus ?
3. Réalisez cette structure, par exemple en définissant une classe Python. ◇

```python
class File:
    def __init__(self, taille=2**8):
        """Retourne une file vide (avec de la place pour 1024 valeurs par défaut)."""
        self.valeurs = taille*[None]
        self.tete = None # indique la case où on a inséré plus ancien élément, ou bien
                                                    `None` si la pile est vide
        self.queue = 0 # indique la case où on insèrera le prochain élément

    def __str__(self):
        if self.est_vide():
            return "File(vide)"
        if self.tete < self.queue:
            vraies_valeurs = self.valeurs[self.tete:self.queue]
        else:
            vraies_valeurs = self.valeurs[self.tete:] + self.valeurs[:self.queue]
        return f"File(contenu: {vraies_valeurs})"

    def __len__(self):
        if self.est_vide():
            return 0
        elif self.tete < self.queue:
            return self.queue - self.tete
        else:
            return len(self.valeurs) + self.queue - self.tete

    def est_vide(self):
        return self.tete is None

    def entree(self,x):
        if self.est_vide():
            self.tete = 0
            self.valeurs[0] = x
            self.queue = 1
            return
        if self.tete == self.queue:
            # la file n'sest pas vide et la queue a rattrapé la tête
            # on choisit de planter au lieu de générer un tableau plus gros
            raise IndexError("tente d'insérer une valeur dans une file pleine")
        self.valeurs[self.queue] = x
        self.queue = (self.queue + 1) % len(self.valeurs)

    def sortie(self):
        if self.est_vide():
            raise IndexError("impossible de sortir un élément d'une file vide")
        x = self.valeurs[self.tete]
        self.tete = (self.tete + 1) % len(self.valeurs)
        if self.tete == self.queue:
            # on a retiré le dernier élément
            self.tete = None
        return x
```

FIGURE 6 – Une classe pour les files en Python

3.3.2. *En Python* . On pourrait être tenté d'utiliser encore le type `list` de Python, dont la méthode `pop()` admet un argument optionnel qui permet de faire sortir l'élément à n'importe quel indice plutôt que le dernier. Ce serait toutefois une mauvaise idée : lors d'un appel `t.pop(0)`, on obtient bien le premier élément, et le contenu du tableau est celui qu'on attend pour une file, mais pour ça, il a fallu *décaler tous les éléments restants.*

Pour faire mieux, Python propose un type `deque` dans le module `collections` : un objet de type `deque` s'utilise de manière similaire à un objet de type `list` mais, en plus de `append()` et `pop()`, il est muni de méthodes `appendleft()` et `popleft()` qui agissent de la même manière en début de liste, le tout en temps constant. [11] Par contre, l'indexage (obtenir ou modifier l'élément à un index donné) ne se fait plus en temps constant : la structure sous-jacente est celle de liste doublement chaînée, où chaque élément est muni d'un pointeur vers l'élément suivant, mais aussi vers le précédent.

4 Algorithmes de tri

Le tri est la tarte à la crême de l'algorithmique : vous avez sans doute déjà étudié un voire plusieurs algorithmes de tri au cours de votre cursus. Dans cette section, on revient sur trois exemples classiques. Avant ça, on établit que le nombre de comparaisons nécessaires dans le pire cas pour les tableaux de longueur k est asymptotiquement minoré par $k \times \log_2(k)$: un algorithme qui fonctionne asymptotiquement en temps $O(k \times \log_2(k))$ peut donc être considéré comme optimal.

4.1 Minoration de la complexité

On appelle *algorithme de tri* tout algorithme qui, recevant un tableau en entrée, dont les éléments sont deux-à-deux comparables pour une certaine relation d'ordre, produit un tableau contenant les mêmes éléments (avec les mêmes répétitions), dans l'ordre croissant. On dit que c'est un *tri par comparaison* si la seule opération utilisée sur les valeurs des éléments du tableau est la comparaison pour l'ordre considéré, et si le comportement de l'algorithme est entièrement déterminé par le résultat des comparaisons. [12]

4.1.1. Théorème (Borne inférieure de complexité pour un tri par comparaison)**.** *Supposons donné un algorithme de tri par comparaison : le nombre de comparaisons effectuées pour trier un tableau de taille k est au moins $\lceil \log_2(k!) \rceil$ dans le pire cas.*

Démonstration. Soit c_k le nombre maximum de comparaisons effectuées pour trier un tableau de longueur k. On peut associer à chaque permutation σ de $[\![k]\!]$ le mot binaire, de longueur au plus c_k, correspondant aux résultats des comparaisons effectuées pour trier cette permutation. Comme le comportement de l'algorithme est entièrement déterminé par le résultat des comparaisons, cette association est injective (sinon, on aurait deux comportements possibles pour un mot de

11. Voir `https://docs.python.org/fr/3.10/library/collections.html#collections.deque`.

12. Sans cette hypothèse, on pourrait par exemple trier un tableau d'entiers dans $[\![c]\!]$ en temps linéaire en $k+c$: il suffit de compter les occurrences de chaque élément de $[\![c]\!]$, en stockant les résultats dans un tableau de longueur c, puis de produire le tableau trié, en générant les éléments dans l'ordre.

```python
def insere_dans_liste_triee(x,l):
    if l == ():
        return (x,()) # un seul élément: c'est trié
    tete, queue = l
    if x <= tete:
        return (x, l) # `x` est le minimum
    # sinon on insère dans la `queue`
    return (tete, insere_dans_liste_triee(x, queue))

def tri_insertion_liste(l):
    l_triee = ()
    while l != ():
        x, l = l
        l_triee = insere_dans_liste_triee(x, l_triee)
    return l_triee
```

FIGURE 7 – Une réalisation du tri par insertion en Python

comparaisons) et c'est un codage préfixe (parce que lorsqu'un mot binaire est dans l'image, il représente une suite de comparaisons suffisante pour déterminer le tri). Comme un codage préfixe de longueur au plus c_k utilise au plus 2^{c_k} codes, on a $k! \leq 2^{c_k}$. □

4.1.2. Exercice. Déduisez du théorème précédent que la complexité (dans le pire cas) d'un tel algorithme est asymptotiquement minorée par $k \times \log_2(k)$. ◇

4.2 Un algorithme facile : le tri par insertion

Le tri le plus simple à expliquer et à réaliser est sans doute le *tri par insertion* : on construit une liste triée en démarrant avec la liste vide, et en insérant les éléments d'origine un par un. L'algorithme est particulièrement facile à écrire avec des listes chaînées : voir la figure 7 (par souci de concision, on utilise la version la plus élémentaire des listes chaînées, sans classe).

Il est facile d'en déduire une version pour les tableaux, modulo les fonctions de conversions que vous avez produites à l'exercice 3.1.1. Il est également possible de réaliser le tri par insertion en place dans un tableau, mais ça demande quelques contorsions (lorsqu'on insère un élément dans la partie triée, on doit décaler tous les suivants).

4.2.1. Exercice (Complexité du tri par insertion)**.**

1. Vérifiez que si on part d'une liste triée dans l'ordre croissant, le temps d'exécution est linéaire en la taille de la liste.
2. Vérifiez que si on part d'une liste triée dans l'ordre décroissant, et sans élément répété, temps d'exécution est quadratique en la taille de la liste — et c'est le pire cas. ◇

4.3 Algorithmes efficaces

Un autre algorithme de tri bien connu est le *tri rapide* (*quicksort*) qui, malgré son nom, n'est pas optimal dans le pire cas (mais très efficace en moyenne). La stratégie consiste à choisir un élément, le *pivot*, pour partitionner la liste ou le tableau en deux (les éléments inférieurs au pivot d'un côté, les autres de l'autre côté), puis à trier récursivement les deux parties. On en donne une version en figure 8. Notez que cette version trie le tableau en place au lieu de retourner un nouveau tableau trié.

```python
def partitionne(t,debut,fin):
    """Partitionne la tranche `[debut:fin]` de `t` au sens du tri rapide.
    Retourne la position du pivot après partition."""
    pivot = t[debut]
    # on stocke d'abord les deux morceaux de la partition dans des tableaux séparés
    t1 = []
    t2 = []
    for x in range(debut+1,fin):
        if x < pivot:
            t1.append(x)
        else:
            t2.append(x)
    # ensuite on recopie le contenu dans l'intervalle
    l1 = len(t1)
    l2 = len(t2)
    for i in range(l1):
        t[debut+i] = t1[i]
    t[debut+l1] = pivot
    for i in range(l2):
        t[debut+l1+1+i] = t2[i]
    return debut+l1

def tri_rapide_rec(t,debut,fin):
    """Trie la tranche `[debut:fin]` du tableau `t` en appliquant le tri rapide."""
    if fin <= debut + 1:
        return
    pivot = partitionne(t, debut, fin)
    tri_rapide_rec(t, debut, pivot)
    tri_rapide_rec(t, pivot+1, fin)

def tri_rapide(t):
    """Trie le tableau `t` en place en appliquant le tri rapide."""
    tri_rapide_rec(t,0,len(t))
```

FIGURE 8 – Une réalisation du tri rapide en Python

4.3.1. Exercice (Complexité du tri rapide)**.** Vérifiez que si le tableau est déjà trié, la complexité est quadratique (parce qu'on a choisi le premier élément comme pivot). ◇

L'algorithme le plus simple à décrire qui atteint l'optimalité est le *tri par fusion* : on coupe le tableau en deux parties égales, qu'on trie récursivement avant de les fusionner. On en donne une version en figure 9. Là encore, on renvoie un nouveau tableau au lieu de faire le tri en place.

4.3.2. Exercice (Complexité du tri par fusion)**.** Vérifiez que la fusion se fait en temps linéaire en la somme des tailles des deux tableaux. Déduisez-en que le tri par fusion est optimal, au sens ou le tri d'un tableau de taille n se fait en temps dominé par $n\ln(n)$. ◇

Malgré l'optimalité théorique du tri par fusion, le tri rapide lui est généralement préféré. On peut en effet montrer que sa complexité moyenne est aussi de l'ordre de $n\ln(n)$, et qu'elle est atteinte tant que le tableau à trier n'est pas trop ordonné : en pratique, en choisissant le pivot au hasard, on atteint l'optimalité presque sûrement. De plus, il est bien adapté à un traitement en place et, en pratique, il est très performant.

4.3.3. *En Python* . Le type `list` est équipé d'une méthode `sort()` qui trie le tableau en place : celle-ci utilise une version hautement optimisée du tri par fusion.[13] La fonction `sorted(`

13. Une description détaillée lui est dédiée dans le code source de Python : `https://github.com/python/cpython/blob/main/Objects/listsort.txt`.

```python
def fusion(t1,t2):
    """Fusionne deux tableaux triés."""
    t = []
    i1 = 0
    i2 = 0
    while i1 < len(t1) and i2 < len(t2):
        if t1[i1] < t2[i2]:
            t.append(t1[i1])
            i1 += 1
        else:
            t.append(t2[i2])
            i2 += 1
    if i1 < len(t1):
        for k in range(i1,len(t1)):
            t.append(t1[k])
    else:
        for k in range(i2,len(t2)):
            t.append(t2[k])
    return t

def tri_fusion(t):
    """Retourne une version triée du tableau `t`."""
    if len(t) <= 1:
        return t
    t1 = tri_fusion(t[:len(t)//2])
    t2 = tri_fusion(t[len(t)//2:])
    return fusion(t1,t2)
```

FIGURE 9 – Une réalisation du tri par fusion en Python

`i)` construit un tableau trié contenant les éléments de 'i' qui peut être n'importe quel itérable. À la louche, c'est comme si `sorted` était défini par :

```python
def sorted(i):
    l = list(i)
    l.sort()
    return l
```

La méthode `list.sort()` et la fonction `sorted()` admettent deux arguments optionnels : `reversed` pour trier dans l'ordre inverse (avec `reversed=True`) ; et `key` pour trier en fonction d'une clé. L'argument `key` doit être une fonction, et le tri se fait sur ses résultats : par exemple `t.sort(key=lambda x:x %2)` trie en fonction du reste modulo 2, ce qui place les pairs avant les impairs.

Le tri de Python est garanti stable : deux éléments égaux (au sens de `==`) ne sont jamais permutés. C'est crucial si on trie par clé, puisque ça préserve l'ordre entre éléments de même clé.

5 Dictionnaires : représenter des fonctions à support fini

Dans la famille des types séquentiels, les *dictionnaires* (on dit aussi *tableaux associatifs*) sont un peu à part. Comme un tableau, un dictionnaire permet d'associer des éléments à certains indices mais, au lieu d'être des entiers dans $[\![n]\!]$, n étant la taille du dictionnaire, les indices sont des objets quelconques (ou presque).

C'est-à-dire qu'un dictionnaire d représente non pas un mot, mais une fonction partielle à support fini : le support de d est l'ensemble des indices, appelés *clés*, auxquels des éléments sont

```python
class Dictionnaire:
    def __init__(self):
        """Construit le dictionnaire vide"""
        self.contenu = []
    def taille(self):
        return len(self.contenu)
    def contient_cle(self, cle):
        for c, _ in self.contenu:
            if c == cle:
                return True
        # on n'a pas trouvé la clé
        return False
    def valeur(self, cle):
        for c, v in self.contenu:
            if c == cle:
                return v
        # on n'a pas trouvé la clé demandée: on soulève l'exception attendue
        raise KeyError
    def change(self, cle, valeur):
        for i in range(len(self.contenu)):
            c, v = self.contenu[i]
            if c == cle:
                # le dictionnaire contient déjà la clé
                # on met à jour la valeur
                self.contenu[i] = valeur
                return
        # on n'a pas trouvé la clé demandée: on crée une nouvelle entrée
        self.contenu.append((cle,valeur))
    def liste_cles(self):
        return [c for c, _ in self.contenu]
```

FIGURE 10 – Un début de classe pour les dictionnaires en Python

associés, les *valeurs*. Mettre en œuvre ce type de structure, c'est réaliser les opérations suivantes de manière efficace :

- créer un dictionnaire vide ;
- étant donnés un dictionnaire d et une clé c, déterminer si d a une valeur pour cette clé ;
- obtenir cette valeur, notée $d[c]$, lorsqu'elle est définie ;
- affecter la valeur v à une clé c quelconque dans d (on ne se préoccupe pas de savoir si $d[c]$ était déjà défini) ;
- lorsque $d[c]$ est définie, supprimer cette association ;
- obtenir la liste des clés de d, et le nombre de clés.

5.0.1. Exercice (Une classe pour les dictionnaires en Python)**.** Une manière de réaliser la structure de dictionnaire, c'est de considérer un dictionnaire comme une liste de couples `(clé,valeur)`. La figure 10 en présente une réalisation en Python un peu naïve, et partielle. Par exemple, après

```python
d = Dictionnaire()
d.change("ab",True)
d.change("c",0)
```

on a `d.contient_cle("ab")==True`, `d.valeur("c")==0` et `d.liste_cles()==["ab", "c"]`.

1. Ajoutez une méthode `supprime(self,cle)` qui supprime le couple associé à cette clé dans `self.contenu` si un tel couple existe, et soulève l'exception `KeyError` sinon.
2. Vérifiez que mises à part la création d'un dictionnaire vide et l'obtention de la taille, toutes les opérations se font en temps linéaire (même en moyenne). ◇

On est donc loin de l'efficacité souhaitée : on voudrait que le test d'appartenance, l'obtention d'une valeur, l'insertion ou la suppression d'un élement, se fassent en temps constant ou presque. À ce stade du cours, on n'a pas tout-à-fait les outils pour expliquer comment réaliser cet objectif, mais c'est possible…

5.0.2. *En Python* . Le type `dict` est précisément une solution. On peut construire un dictionnaire avec une expression de la forme `{cle1: valeur1, cle2:valeur2, ...}`. En particulier `{}` crée un dictionnaire vide. Si `d` est un dictionnaire, on teste la présence d'une clé `c` avec `c in d`, on obtient la valeur associée avec `d[c]`, on la change avec `d[c] =valeur`, on la supprime avec `del d[c]`, on obtient taille avec `len(d)` et la liste des clés avec `list(d)`. Mise à part cette dernière opération, tout se fait en temps constant *en moyenne* (on conserve un temps linéaire dans le pire cas, mais la probabilité de ce pire cas est négligeable). Par ailleurs, le type est itérable : on peut itérer sur les clés d'un dictionnaire avec `for c in d:...`.

Il y a cependant une restriction sur la nature des clés : l'efficacité des opérations dépend de la possibilité d'associer un entier de taille fixée à chaque clé : sa *valeur de hachage*. On ne peut donc utiliser comme clé que des objets dit « hachables », c'est-à-dire pour lesquels la fonction `hash` retourne une valeur. Ceci exclut par exemple tous les types mutables, comme `list`.

5.0.3. Exercice (Remplacement de caractères)**.** Définissez une fonction `remplace(d,mot)` qui attend un dictionnaire `d` qui associe des caractères à des caractères, et une chaîne `mot`, et qui retourne la chaîne dans laquelle les caractères qui sont des clés de `d` sont remplacées par leur valeur. Par exemple, on veut que `remplace({'c':'p', 'm':'r'},"acme")` retourne `"apre"`. ◇

5.0.4. Exercice (Comptage)**.** Définissez une fonction `compte(t)` qui attend un tableau (ou plus généralement un itérable) dont les éléments sont hachables, et retourne un dictionnaire dont les clés sont les éléments présents, et tel que la valeur d'un élément est le nombre d'occurrences dans le tableau. ◇

6 Itérateurs : un principe de programmation plutôt qu'une structure de données

Jusqu'ici on a discuté de différentes manières de voir les mots ou suites d'objets comme des structures de données, favorisant certaines constructions et opérations au détriment d'autres. La caricature en la matière est la dichotomie entre tableaux (accès à un élément par son index en temps 1, mais insertion ou suppression d'un élément en temps linéaire) et listes (insertion ou suppression en temps 1 du côté de la tête, mais accès aux autres éléments en temps linéaire).

Le concept d'*itérateur* offre un point de vue alternatif en associant aux listes (au sens de la section section 3.1) un principe de calcul plutôt qu'une structure de données. Informellement, un itérateur est un objet qu'on peut interroger pour obtenir : ou bien un élément (correspondant à la tête), et alors l'itérateur change d'état pour progresser d'un cran (il représente maintenant la queue de la liste) ; ou bien l'information qu'il n'y a plus d'éléments (ce qui représente la liste vide). Il suffit de convenir d'un protocole pour interroger un objet de cette manière : tout objet qui s'y conforme est un itérateur. Évaluer la boucle :

```
pour x dans t:
    faire des choses avec x
```

se résume alors à répéter les opérations

demander un élément à t et le stocker dans x
faire des choses avec x

jusqu'à ce que t signale qu'il n'a plus rien à donner. Ce protocole peut être une simple discipline appliquée par la programmeuse, ou bien être plus ou moins formalisé dans le langage.

6.0.1. *En Python* . La spécification de Python définit [14] un itérateur comme un objet muni de la méthode `__next__()` qui retourne l'élément courant et passe à l'état suivant, ou bien soulève l'exception `StopIteration`. Mais surtout, le langage fournit un moyen de *programmer* des itérateurs !

Si un corps de fonction contient une ou des instructions `yield`, sa valeur de retour est un itérateur dont le comportement est décrit par l'exécution de la fonction. Lorsqu'on appelle sa méthode `__next__()`, le corps de la fonction est exécuté, jusqu'à la prochaine instruction `yield` : l'exécution est alors suspendue, et l'argument de `yield` fournit la valeur de cet appel. Lors du prochain appel, l'exécution reprendra là où elle s'était arrêtée. Si la fonction se termine sans atteindre une instruction `yield`, l'itérateur est terminé. Dans la terminologie de Python, la fonction définissant cet itérateur est appelée *fonction génératrice*, et l'itérateur est qualifié d'*itérateur de générateur*. [15]

Par exemple, si on définit la fonction :

```
def carres(n):
    """Génère les carrés des nombres de `1` à `n`."""
    for i in range(1,n+1):
        yield i*i
```

on pourra afficher les carrés des nombres de 1 à 10 avec la boucle :

```
for i in carres(10):
    print(i)
```

Une autre manière de générer des itérateurs, c'est d'utiliser une expression génératrice : précisément celles qui servent à définir des listes par compréhension. Lorsqu'on écrit `list(i*i for i in range(1,11))`, l'expression `i*i for i in range(1,11)` définit un itérateur, qui est convertit en `list` par l'appel à la fonction du même nom.

L'avantage des itérateurs sur les listes, c'est qu'on n'a pas besoin de construire tous les éléments *avant* de commencer à calculer : si on n'utilise que le premier élément, ce sera le seul à être calculé. Et il n'est en fait même pas imposé que la liste produite soit finie !

6.0.2. Exercice (Un itérateur pour Syracuse)**.** Définissez un itérateur `syracuse(x)` qui produit les termes de la suite de Syracuse [16] de premier terme `x`, jusqu'au premier 1. Par exemple, on veut que que `list(syracuse(5))==[5, 16, 8, 4, 2, 1]` mais on veut aussi que `next(syracuse(n))==n` pour n'importe quelle valeur de `n`, en temps constant. ◇

14. Voir `https://docs.python.org/fr/3.10/glossary.html#term-iterator`.
15. Voir `https://docs.python.org/fr/3.10/glossary.html#term-generator`.
16. Voir `https://fr.wikipedia.org/wiki/Suite_de_Syracuse`[W].

Solutions des exercices

Solution de l'exercice 1.2.1. On fait une seule itération : pour chaque rang, on fait l'addition des deux chiffres et de la retenue de ce rang (ce qui se fait en temps constant).

Solution de l'exercice 2.2.1. Voir le fichier `iteration.py` déposé sur Amétice.

Solution de l'exercice 2.2.2. Voir le fichier `iteration.py` déposé sur Amétice. Dans la version initiale du document, la première question était mal formulée, on ne traite que la version corrigée ici.

Solution de l'exercice 2.3.1. Voir le fichier `iteration.py` déposé sur Amétice.

Solution de l'exercice 2.3.3. À chaque tour de boucle, la différence `fin-debut` est au moins divisée par 2 : en effet, `milieu-debut == (fin-debut)// 2 <= (fin-debut) / 2` et `fin-(milieu+1) == fin - ((fin-debut)// 2 + 1) < fin - (fin-debut)/2 == (fin-debut)/2`.

Solution de l'exercice 2.4.1. Le pire cas se produit lorsque la boucle de `position_sousmot` est se déroule jusqu'au bout, (ce qui se produit lorsque tous les appels à `sousmot_en_position` renvoient `False`, sauf peut-être le dernier), et de plus les boucles des appels à `sousmot_en_position` sont déroulées (ce qui se produit lorsque `v[j] == u[i+j]` pour `j<len(v)-1`.

Avec les valeurs de `u` et `v` choisies, c'est précisément ce qui se produit : la boucle de `position_sousmot` sera déroulée pour `len(u)-len(v)+1 == n` fois ; et pour chacune de ces itérations, la boucle de `sousmot_en_position` sera déroulée `len(v) == k` fois ; ce qui donne une complexité en $O(n \times k)$.

Solution de l'exercice 3.1.1. Voir le fichier `iteration.py` déposé sur Amétice, qui traite la version basique, dans un style récursif.

Un appel à `element_liste(l,i)` plantera après `len(l)` appels récursifs si `i>len(l)-1`, et fera `i` appels récursifs sinon (on a la réponse tout de suite si `i==0`, et on décrémente `i` à chaque appel sinon). Les cas qui ne plantent pas sont donc en $O(i)$.

Solution de l'exercice 3.2.2. Voir le fichier `npi.py` déposé sur Amétice,

Solution de l'exercice 3.3.1.

1. Créer une file vide, c'est créer deux piles vides, en temps constant. Tester si la file est vide, c'est tester si les deux piles le sont, encore en temps constant. Pousser un élément dans la file, c'est le pousser sur la pile d'entrée, ce qui se fait en temps constant. Sortir un élément de la file, c'est le sortir de pile de sortie, ce qui se fait en temps constant, *ou*, si elle est vide, renverser la pile d'entrée vers la pile de sortie : cette opération se fait en temps linéaire, c'est le seul cas critique.
2. Avec cette méthode, il n'y a pas de limite de taille (si on suppose que les piles sont de taille arbitraire).
3. Voir le fichier `piles_files.py` déposé sur Amétice,

Solution de l'exercice 4.1.2. La formulation « asymptotiquement minorée » est un peu floue. Formellement, on montre que la complexité est minorée par une fonction de l'ordre de $k\log_2 k$. La complexité est évidemment minorée par le nombre de comparaisons et donc, par le théorème 4.1.1, minorée par $\log_2(k!)$. Or $\log_2(k!) = \sum_{i=2}^{k} \log_2(i)$. Comme la fonction $\log_2$ est croissante, on a $\log_2(k!) \geq \int_1^k \log_2(x)dx = [x\log_2(x) - x]_1^k = k\log_2(k) - k + 1$ (c'est la même technique qu'au début de la démonstration de la formule de Stirling). Cette dernière fonction est équivalente à $k\log_2(k)$.

Solution de l'exercice 4.2.1.

1. La boucle de `tri_insertion_liste` sera déroulée exactement autant de fois qu'il y a d'éléments : l'ordre dans lequel ils apparaissent n'influe pas sur ce point.
   Si on part d'une liste triée dans l'ordre croissant, à chaque tour de boucle `x` est inférieur ou égal à tous les éléments de `l`, donc à tous les éléments de `l_triee`, donc l'appel à `insere_dans_liste_triee` termine immédiatement, sans faire d'appel récursif.
2. Le pire cas se produit lorsque chaque appel à `insere_dans_liste_triee(x, l_triee)` dans la boucle de `tri_insertion_liste` provoque des appels récursifs jusqu'à parvenir à la liste vide. C'est le cas si et seulement si tous les tests `x <= tete` échouent, c'est-à-dire si les éléments de la liste apparaissent en ordre strictement décroissant. Et dans ce cas, il y a `len(l_triee)` appels récursifs. Comme `len(l_triee)` varie de `0` à `len(l)-1` au cours de la boucle, la complexité est de l'ordre de $\sum_{i=0}^{k-1} i = \frac{(k-1)k}{2}$ avec $k =$ `len(l)`, ce qui est de l'ordre de k^2.

Solution de l'exercice 4.3.1. L'appel à `partitionne` se fait clairement en temps linéaire en `fin-debut`. Si le tableau est déjà trié entre `debut` et `fin`, dans chaque appel à `partitionne`, `pivot` sera toujours le minimum, `t1` sera toujours vide, `t2` sera toujours `t[debut+1:fin]`, et la valeur de retour sera `debut+1`.

Si `t` est trié et $k =$ `fin-debut`, chaque appel à `tri_rapide_rec(t,debut,fin)` avec $k > 1$ fera :

- un appel à `partitionne`, en temps linéaire en k ;
- un appel récursif à `tri_rapide_rec(t,debut,debut+1)`, qui termine immédiatement (en temps constant) ;
- un appel récursif à `tri_rapide_rec(t,debut+2,fin)`.

En notant c_k la complexité de l'appel à `tri_rapide_rec(t,debut,fin)`, on obtient que c_0 et c_1 sont constantes, et $c_{k+2} = ak + b + c_k$ avec des constantes a et b.[17] On en déduit que $c_{2k+2} = ak(k+1) + b(k+1) + c_0$ et $c_{2k+3} = ak(k+1) + b(k+1) + c_1$ donc c_k est de l'ordre de k^2.

Solution de l'exercice 4.3.2. Lors d'un appel à `fusion`, le corps de la boucle `while` est itéré au plus `len(t1)+len(t2)-1` fois, donc en temps linéaire en `len(t1)+len(t2)`. La boucle `for` empruntée ensuite est itérée a plus `max(len(t1),len(t2))` fois. Donc tout appel à `fusion` se fait en temps dominé par `len(t1)+len(t2)`.

Pour $n > 0$, un appel à `tri_fusion` sur un tableau de taille inférieure à 2^n termine en temps borné par une constante b pour les tableaux de taille ≤ 1, ou bien il fait deux appels récursifs

17. Formellement, il faudrait travailler avec des Θ : on minore c_{k+2} par un $ak + b + c_k$, et on le majore par un $a'k + b' + c_k$. De même c_0 et $c_1 \in \Theta(1)$. Et on raisonne comme dans la suite, pour montrer que c_k est minorée par une fonction de l'ordre de k^2, puis majorée par une autre fonction de l'ordre de k^2 (seules les constantes changent).

à des tableaux de taille inférieure à 2^{n-1}, puis effectue une fusion en temps dominé par 2^n. En notant c_k la complexité dans le pire cas du tri fusion pour des tableaux de taille inférieure à k, on obtient que $c_{2^n} \leq 2c_{2^{n-1}} + a2^{n-1}$ pour une certaine constante a. Donc $c_{2^n} \leq (na+b)2^n$ (par une récurrence facile). Donc $c_k \leq c_{2^{\lceil \log_2(k) \rceil}} \leq (\lceil \log_2(k) \rceil a+b)2^{\lceil \log_2(k) \rceil} \leq ((\log_2(k)+1)a+b)2^{\log_2(k)+1} = 2k(\log_2(k)a + a + b) \in O(k \log_2(k))$.

Solution de l'exercice 5.0.1.

1. Voir le fichier `dictionnaires.py`.
2. Chacune de ces méthodes comprend une itération sur le contenu.

Solution de l'exercice 5.0.3. Voir le fichier `iteration.py`.

Solution de l'exercice 5.0.4. Voir le fichier `iteration.py`.

Solution de l'exercice 6.0.2. Voir le fichier `syracuse.py`.