TRIER ET RECHERCHER

À la fin de ce chapitre, je sais :

- ☞ expliquer le fonctionnement d'algorithmes de tri simple,
- ☞ compter le nombre d'opérations élémentaires nécessaires à l'exécution d'un algorithme,
- ☞ caractériser un tri (comparatif, en place, stable, en ligne)
- ☞ rechercher un élément dans un tableau trié par recherche dichotomique

Trier est une activité que nous pratiquons dès notre plus jeune âge : les couleurs, les formes, les tailles sont autant d'entrées possibles pour cette activité. Elle est également omniprésente dans l'activité humaine, dès l'instant où il faut rationaliser une activité. Dans le cadre du traitement de l'information, on y a si souvent recours qu'on n'y prête quasiment plus attention. C'est pourquoi les algorithmes de tri ont été intensément étudiés et raffinés à l'extrême : il s'agit d'être capable de trier n'importe quel jeu de données pourvu qu'un ordre puisse être défini et de sélectionner le tri le plus efficace selon le tri à effectuer.

Lors du traitement des informations, le tri est souvent associé à une autre activité : la recherche d'un élément dans un ensemble trié. En effet, la recherche dichotomique permet d'accélérer grandement la recherche d'un élément, pourvu que celui-ci soit trié.

Ce chapitre esquisse donc quelques algorithmes de tri simples à coder puis aborde la recherche recherche dichotomique dans un tableau trié.

Dans tout le chapitre, on considère :

- un tableau noté t, indiçable avec des crochets, c'est à dire que l'on peut accéder à un élément en écrivant t[i],
- des éléments à trier dont le type importe peu, pourvu qu'on dispose d'un ordre sur ces éléments,

On donnera au cours de l'année une définition précise d'une relation d'ordre, mais, pour l'instant, on considère juste qu'il est possible de comparer deux éléments entre eux. C'est à dire qu'on peut réaliser un test de type `t[i] < t[j]` et récupérer une réponse booléenne à ce test. C'est d'ailleurs ainsi que la plupart des contexte informatique définissent un ordre dans un type de donnée.

P En Python, on implémentera ces tableaux par des listes.

A Comment caractériser un algorithme de tri ?

Pour distinguer les différents algorithmes de tri, on dispose de nombreux qualificatifs. Les définitions suivantes en expliquent quelques uns parmi les plus courants.

■ **Définition 1 — Tri par comparaisons.** Un tri par comparaison procède en comparant les éléments deux à deux [a].

a. ce qui sous-entend que certains algorithmes, comme l'algorithme de tri par comptage, procèdent différemment.

■ **Définition 2 — Tri en place.** Un tri est dit en place s'il peut être directement effectué dans le tableau initial et ne nécessite pas l'allocation d'une nouvelle structure en mémoire de la taille du tableau initial [a].

a. Cette propriété est importante car la mémoire est un paramètre très couteux des systèmes électroniques, tant sur le plan énergétique que sur le plan financier.

■ **Définition 3 — Tri incrémental ou en ligne.** Il s'agit d'un tri capable de commencer le tri avant même d'avoir reçu l'intégralité des données [a].

a. Cette propriété est très intéressante dans le cadre pour le traitement en temps réel des systèmes dynamiques (qui évoluent dans le temps)

■ **Définition 4 — Tri stable.** Un tri est dit stable s'il préserve l'ordonnancement initial des éléments que l'ordre considère comme égaux mais que l'on peut distinguer [a].

a. par exemple, dans un jeu de cartes, deux valets (même type de carte) de couleurs différentes.

B Trier un tableau

a Tri par sélection

Le tri par sélection est un tri par comparaisons et échanges, en place, non stable et hors ligne.

Son principe est le suivant. On cherche le plus petit élément du tableau (de droite) et on l'insère à la fin du tableau trié de gauche. Au démarrage de l'algorithme, on cherche le plus petit élément du tableau et on l'insère à la première place. Puis on continue avec le deuxième plus petit élément du tableau que l'on ramène à la deuxième place et ainsi de suite.

b Tri par insertion

Le tri par insertion est un tri par comparaison, stable et en place. De plus, si les données ne sont pas toutes présentes au démarrage de l'algorithme, le tri peut quand même commencer,

Algorithme 1 Tri par sélection

```
1 : Fonction TRIER_SELECTION(t)
2 :     n ← taille(t)
3 :     pour i de 0 à n − 1 répéter
4 :         min_index ← i                          ▷ indice du prochain plus petit
5 :         pour  j de i + 1 à n − 1 répéter       ▷ pour tous les éléments non triés
6 :             si t[j] < t[min_index ] alors
7 :                 min_index ← j                  ▷ c'est l'indice du plus petit non trié!
8 :         échanger(t[i], t[min_index])           ▷ c'est le plus grand des triés!
```

ce qui n'est pas le cas du tri par sélection. C'est donc un tri en ligne également.

Son principe est le suivant : on considère deux parties gauche (triée) et droite (non triée) du tableau. Puis, on cherche à insérer le premier élément non trié (du tableau de droite) dans le tableau trié (de gauche) à la bonne place.

Algorithme 2 Tri par insertion

```
1 : Fonction TRIER_INSERTION(t)
2 :     n ← taille(t)
3 :     pour i de 1 à n-1 répéter
4 :         à_insérer ← t[i]
5 :         j ← i
6 :         tant que t[j-1] > à_insérer et j>0  répéter
7 :             t[j] ← t[j-1]                      ▷ faire monter les éléments
8 :             j ← j-1
9 :         t[j] ← à_insérer                       ▷ insertion de l'élément
```

c Tri par comptage

Le tri par comptage est un tri par dénombrement de valeurs entières. Il ne porte donc que sur des tableaux contenant des entiers. C'est un tri non comparatif, stable et hors ligne.

Le principe du tri par comptage est de compter le nombre d'occurrences de chaque valeur entière puis de construire un nouveau tableau à partir de ce comptage.

R Ce tri est certes non comparatif car il n'utilise pas explicitement de comparaisons. Mais l'ordre des entiers est utilisé lors du comptage.

Algorithme 3 Tri par comptage

```
1 : Fonction TRIER_COMPTAGE(t, v_max)          ▷ v_max est le plus grand entier à trier
2 :     n ← taille(t)
3 :     comptage ← un tableau de taille v_max + 1 initialisé avec des zéros
4 :     résultat ← un tableau de taille n
5 :     pour i de 0 à n − 1 répéter             ▷ compter chaque valeur du tableau.
6 :         key ← t[i]
7 :         comptage[key] ← comptage[key] + 1
8 :     i ← 0                                   ▷ Indice du tableau résultat
9 :     pour key de 0 à v_max − 1 répéter       ▷ pour chaque valeur du tableau
10 :        pour j de 0 à comptage[key] répéter  ▷ autant de fois que la valeur est présente
11 :            résultat[i] ← key
12 :            i ← i+1
13 :    renvoyer résultat
```

C Comparatif des tris

Les tableaux 1 et 2 résument les caractéristiques et les complexités des principaux algorithmes de tri au programme.

Tris	Comparaison	Stable	En place	En ligne
par sélection	oui	non	oui	non
par insertion	oui	oui	oui	oui
par comptage	non	oui	non	non
fusion	oui	oui	non	non
rapide	oui	non	oui	non

TABLE 1 – Comparatif des caractéristiques des algorithmes de tri. Ils existent de nombreuses variantes de ces algorithmes. Les informations de ce tableau concernent les versions implémentées en cours ou en TP.

Tris	Complexité temporelle **pire des cas**	Complexité temporelle **moyenne**	Complexité temporelle **meilleur des cas**	Complexité spatiale
par sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
par insertion	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
par comptage	$O(n+v\_max)$	$O(n+v\_max)$	$O(n+v\_max)$	$O(n+v\_max)$
fusion	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$
rapide	$O(n^2)$	$O(n\log n)$	$O(n\log n)$	$O(\log n)$

TABLE 2 – Complexité des différents algorithmes de tri.

D Recherche séquentielle

Rechercher d'un élément dans un tableau est une opération nécessaire pour de nombreux algorithmes. Il est donc important de disposer d'algorithmes rapides pour exécuter cette tâche. La recherche séquentielle est l'algorithme le plus naïf pour réaliser cette tâche. L'algorithme 4 rappelle la recherche séquentielle d'un élément dans un tableau.

Algorithme 4 Recherche séquentielle d'un élément dans un tableau

```
1 : Fonction RECHERCHE_SÉQUENTIELLE(t, elem)
2 :     n ← taille(t)
3 :     pour i de 0 à n − 1 répéter
4 :         si t[i]= elem alors
5 :             renvoyer i                    ▷ élément trouvé, on renvoie sa position dans t
6 :     renvoyer l'élément n'a pas été trouvé
```

Dans le pire des cas, c'est à dire si l'élément recherché n'appartient pas au tableau, la recherche séquentielle nécessite un nombre d'instructions proportionnel à n, la taille du tableau. On dit que sa complexité est en $O(n)$.

E Recherche dichotomique

Si le tableau dans lequel la recherche est à effectuer est trié, alors la recherche dichotomique à privilégier. Celle-ci est illustrée sur la figure 1.

■ **Définition 5 — Recherche dichotomique.** La recherche dichotomique recherche un élément dans un tableau trié en éliminant à chaque itération la moitié du tableau qui ne contient pas l'élément. Le principe est le suivant :

- comparer la valeur recherchée avec l'élément **médian** du tableau,

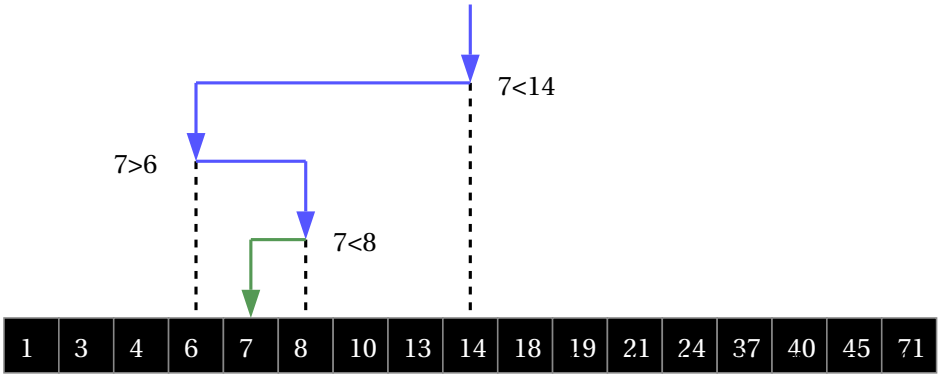

FIGURE 1 – Illustration de la recherche dichotomique de la valeur 7 dans un tableau trié (Source : Wikimedia Commons)

- si les valeurs sont égales, la position de l'élément dans le tableau a été trouvée,
- sinon, il faut poursuivre la recherche dans la moitié du tableau qui contient l'élément à coup sûr.

Si l'élément existe dans le tableau, alors l'algorithme renvoie son indice dans le tableau. Sinon, l'algorithme signifie qu'il ne l'a pas trouvé.

Le mot dichotomie vient du grec et signifie couper en deux. En prenant l'élément médian du tableau, on opère en effet une division de la taille du tableau en deux parties. On ne conserve que la partie qui pourrait contenir l'élément recherché.

Algorithme 5 Recherche d'un élément par dichotomie dans un tableau trié

```
1 : Fonction RECHERCHE_DICHOTOMIQUE(t, elem)
2 :     n ← taille(t)
3 :     g ← 0
4 :     d ← n-1
5 :     tant que g ⩽ d répéter                ▷ ⩽ cas où valeur au début, au milieu ou à la fin
6 :         m ← (g+d)//2                      ▷ Division entière : un indice est un entier !
7 :         si t[m] < elem alors
8 :             g ← m + 1                     ▷ l'élément devrait se trouver dans t[m+1, d]
9 :         sinon si t[m] > elem alors
10 :            d ← m - 1                     ▷ l'élément devrait se trouver dans t[g, m-1]
11 :        sinon
12 :            renvoyer m                    ▷ l'élément a été trouvé
13 :    renvoyer l'élément n'a pas été trouvé
```

À chaque tour de boucle, l'algorithme 5 divise par deux la taille du tableau considéré. Dans le pire des cas, le dernier tableau considéré comporte un seul élément. Supposons que la taille du tableau est une puissance de deux : $n = 2^p$. On a alors $1 = \frac{n}{2^k} = \frac{2^p}{2^k}$, si k est le nombre d'itérations effectuées pour en arriver là. C'est pourquoi le nombre d'itérations peut s'écrire :

$k = \log_2 2^p = p \log_2 2 = p = \log_2(n)$. La complexité de cet algorithme est donc logarithmique en $O(\log_2(n))$, ce qui est bien plus efficace qu'un algorithme de complexité linéaire.

R Si le tableau contient plusieurs occurrences de l'élément à chercher, alors l'algorithme 5 renvoie un indice qui n'est pas celui de la première occurrence de l'élément. C'est pourquoi, l'algorithme 6 propose une variation qui renvoie l'indice de la première occurrence de l'élément.

Algorithme 6 Recherche d'un élément par dichotomie dans un tableau trié, renvoyer l'indice minimal en cas d'occurrences multiples.

```
1 : Fonction RECHERCHE_DICHOTOMIQUE_INDICE_MIN(t, elem)
2 :     n ← taille(t)
3 :     g ← 0
4 :     d ← n-1
5 :     tant que g < d répéter                          ▷ attention au strictement inférieur!
6 :         m ← (g+d)//2                                ▷ Un indice de tableau est un entier!
7 :         si t[m] < elem alors
8 :             g ← m + 1                               ▷ l'élément devrait se trouver dans t[m+1, d]
9 :         sinon
10 :            d ← m                                   ▷ l'élément devrait se trouver dans t[g, m]
11 :    si t[g] = elem alors
12 :        renvoyer g
13 :    sinon
14 :        renvoyer l'élément n'a pas été trouvé
```

L'algorithme de la recherche dichotomique peut s'écrire de manière récursive, comme on le verra dans le chapitre suivant.

R Attention à ne pas confondre cet algorithme avec la recherche de zéro d'une fonction par dichotomie. Le principe est le même. Cependant, dans le cas de la recherche dichotomique, on s'intéresse à des indices de tableaux, donc à des nombres entiers. C'est pourquoi on choisit de prendre le milieu via la division entière `//`. Lorsqu'on cherche les zéros d'une fonction mathématique, on s'intéresse à des nombres réels représentés par des flottants en machine. C'est pourquoi lorsqu'on prend le milieu, on choisit alors la division de flottants `/`.

R Le milieu d'un segment `[a,b]` vaut `m=(a+b)/2`, **non pas** `m=(b-a)/2`.

F Compter les opérations

Pour comparer l'efficacité des algorithmes, on compte le nombre d'opérations élémentaires nécessaires à leur exécution en fonction de la taille des données d'entrée. Dans notre cas, la taille des données d'entrée est souvent la taille du tableau à trier. Que se passe-t-il lorsque cette taille augmente? Le temps d'exécution de l'algorithme augmente-t-il? Et de quelle manière? Linéairement, exponentiellement ou logarithmiquement par rapport à la taille?

Dans ce qui suit, on suppose qu'on dispose d'une machine pour tester l'algorithme. On fait l'hypothèse réaliste que les opérations élémentaires suivantes sont réalisées en des temps constants par cette machine :

- opération arithmétique $+, -, *, /, //, \%$, coûts associé c_{op},
- tests $==, !=, <, >$, coûts associé c_t,
- affectation $\leftarrow$, coût associé $c_{\leftarrow}$
- accès à un élément indicé $t[i]$, coût associé c_a
- structures de contrôles (structures conditionnelle et boucles), coût associé négligé,
- échange de deux éléments dans le tableau, coût c_e,
- accès à la longueur d'un tableau, coût c_l.

■ **Exemple 1 — Compter le nombres d'opération élémentaires de la recherche séquentielle.** Sur l'algorithme 7, on a reporté les coûts pour chaque instruction. On peut donc maintenant calculer le coût total. Ce coût C va dépendre de la taille du tableau que l'on va noter n. Il va également dépendre du chemin d'exécution : est-ce que le test ligne 4 est valide ou non? On choisit de se placer dans le pire des cas, c'est à dire qu'on suppose que le test est invalidé à toutes les itérations [a]. Dans ce cas, on peut dénombrer les coûts élémentaires :

$$C(n) = c_{\leftarrow} + \sum_{i=0}^{n-1} c_a + c_t \tag{1}$$

$$= c_{\leftarrow} + (c_a + c_t)n \tag{2}$$

Dans le pire des cas, on observe donc que le coût de la recherche séquentielle est proportionnel à n. On notera cette au deuxième semestre $C(n) = O(n)$, signifiant par là que le coût est dominé asymptotiquement par n. Lorsque n augmente, le coût de la recherche séquentielle dans le pire des cas n'augmente pas plus vite que n.

Si on se place dans le meilleur des cas, c'est à dire lorsque l'élément recherché se situe dans la première case du tableau, on obtient un coût total de $C(n) = c_{\leftarrow} + c_a + c_t$, c'est à dire un coût qui ne dépend pas de n. On le notera $C(n) = O(1)$ et on dira que ce coût est constant.

a. i.e. l'élément cherché n'est pas dans le tableau.

Algorithme 7 Recherche séquentielle d'un élément dans un tableau

```
1 : Fonction RECHERCHE_SÉQUENTIELLE(t, elem)
2 :     n ← taille(t)                                  ▷ affectation : c←
3 :     pour i de 0 à n − 1 répéter                    ▷ répéter n fois
4 :         si t[i]= elem alors                        ▷ accès, test : ca + ct
5 :             renvoyer i
6 :     renvoyer l'élément n'a pas été trouvé
```

R L'analyse des calculs précédents montre qu'on peut simplifier notre approche du décompte des opérations. Étant donné que ce qui nous intéresse, c'est l'évolution du coût en fonction de la taille du tableau, les valeurs des constantes de coût des opérations élémentaires importent peu, pourvu que celles-ci soient constantes. **C'est pourquoi, on supposera désormais qu'on dispose d'une machine de test pour nos algorithmes telle que chaque instruction élémentaire possède un coût constant que l'on notera c.**

Algorithme 8 Tri par insertion, calcul du nombre d'opérations

```
1 : Fonction TRIER_INSERTION(t)
2 :     n ← taille(t)                                  ▷ affectation : c
3 :     pour i de 1 à n-1 répéter
4 :         à_insérer ← t[i]                           ▷ accès, affectation : 2c
5 :         j ← i                                      ▷ affectation : c
6 :         tant que t[j-1] > à_insérer et j>0 répéter ▷ accès, tests : 3c
7 :             t[j] ← t[j-1]                          ▷ accès, affectation : 3c
8 :             j ← j-1                                ▷ affectation : c
9 :         t[j] ← à_insérer                           ▷ accès, affectation : 2c
```

■ **Exemple 2 — Compter le nombre d'opérations élémentaires pour le tri par insertion.** Sur l'algorithme 8, on a reporté les coûts pour chaque instruction. On peut donc maintenant calculer le coût total. On choisit de se placer dans le pire des cas, c'est à dire lorsqu'on insère l'élément systématiquement au début du tableau[a]. La boucle *tant que* est donc exécutée $i-1$ fois.

$$C(n) = c + \sum_{i=1}^{n-1} (2c + c + (i-1)(3c+c) + 2c) \tag{3}$$

$$= c + c\sum_{i=1}^{n-1} (1+4i) \tag{4}$$

$$= c + c(n-1) + 4\frac{n(n-1)}{2} \tag{5}$$

$$= (c-2)n + 2n^2 \tag{6}$$

Dans le pire des cas, le coût total du tri par insertion est donc en $O(n^2)$.

Le meilleur des cas, pour l'algorithme de tri par insertion, c'est lorsque l'élément à insérer n'a jamais à être déplacé [b]. Alors la condition de sortie de la boucle *tant que* est valide dès le premier test. Le coût total devient alors :

$$C(n) = c + \sum_{i=1}^{n-1} (2c + c + 3c) \tag{7}$$
$$= c + 6c(n-1) \tag{8}$$
$$= -5c + 6cn \tag{9}$$

Dans le meilleur des cas, on a donc un coût du tri par insertion en $O(n)$.

a. Le tableau est donné en entrée dans l'ordre inverse à l'ordre souhaité.
b. Le tableau donné en entrée est déjà trié !

■ **Exemple 3 — Compter le nombre d'opérations élémentaires pour le tri par comptage.** Si on observe l'algorithme 3, on se rend compte qu'il est composé d'une suite de trois boucles, dont deux dépendent de la taille du tableau n et une de la valeur maximale des entiers qu'on chercher à trier v_{max}. On peut compter le nombre d'opérations et procéder de la même manière que précédemment mais en tenant compte de ces deux paramètres.

On note que pour cette algorithme, le nombre d'itérations est connu dès le départ, il n'y a pas de pire ou de meilleur cas.

$$C(n, v_{max}) = cv_{max} + c + \sum_{i=0}^{n-1} 6c + \sum_{i=1}^{v_{max}} 3c + \sum_{i=0}^{n-1} 5c \tag{10}$$
$$= cv_{max} + c + 6cn + 3c(v_{max} - 1) + 5cn \tag{11}$$
$$= c(v_{max} - 2) + 3cv_{max} + 11cn \tag{12}$$
$$= O(v_{max} + n) \tag{13}$$
$$= O(\max(v_{max}, n)) \tag{14}$$

Le coût total de l'algorithme de tri par comptage est donc linéaire par rapport aux deux paramètres d'entrée. Il est donc raisonnable de l'utiliser lorsque n est du même ordre que v_{max}. Par contre, il nécessite la création d'un tableau de même taille n [a], ce qui peut avoir des conséquences sur la mémoire du système.

a. ce tri n'est pas en place

G Trier avec les fonctions natives de Python

P Il existe deux solutions natives pour trier une liste en Python :

1. appeler la fonction `sorted` sur une liste ou un dictionnaire. Cette fonction renvoie une nouvel objet trié sans modifier l'original. Par exemple, `T = sorted(L)`. Dans le cas d'un dictionnaire, ce sont les clefs qui sont triées.
2. appeler la méthode `sort` de la classe List. Cette méthode effectue un tri en place sur une liste et la modifie. Par exemple, `L.sort()`. Cette méthode n'existe pas pour les dictionnaires.

Les tris opérés par ces fonctions sont garantis stables. Cela signifie que lorsque plusieurs enregistrements on la même clef, leur ordre original est préservé.

La documentation Python est à consulter ici pour les usages plus spécifiques de ces fonctions. On peut notamment spécifier :

- un tri ascendant ou descendant à l'aide du paramètre optionnel booléen `reverse`,
- à la fonction l'élément à utiliser pour le tri (en cas d'élément multiple) et même le comparateur à l'aide du paramètre optionnel `key`.

P Par défaut, Python trie les n-uplets dans l'ordre lexicograhique.