RECHERCHER

À la fin de ce chapitre, je sais :

- ☞ rechercher un élément dans un tableau trié par recherche dichotomique
- ☞ compter le nombre d'opérations élémentaires nécessaires à l'exécution d'un algorithme,

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 aborde la recherche recherche dichotomique dans un tableau trié.

A Recherche séquentielle

L'algorithme **??** détaille la recherche séquentielle d'un élément dans un tableau : on cherche, élément après élément, si un certain élément appartient au tableau.

Algorithme 1 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 $O(n)$.

(P) Lorsqu'on utiliser l'opérateur `in` en Python pour tester l'appartenance d'un élément à une liste comme dans `e in L`, l'algorithme utilisé est celui de la recherche séquentielle. Le coût de cet usage est linéaire par rapport à la taille de la liste. Ce n'est donc pas une opération élémentaire.

B 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 **??**.

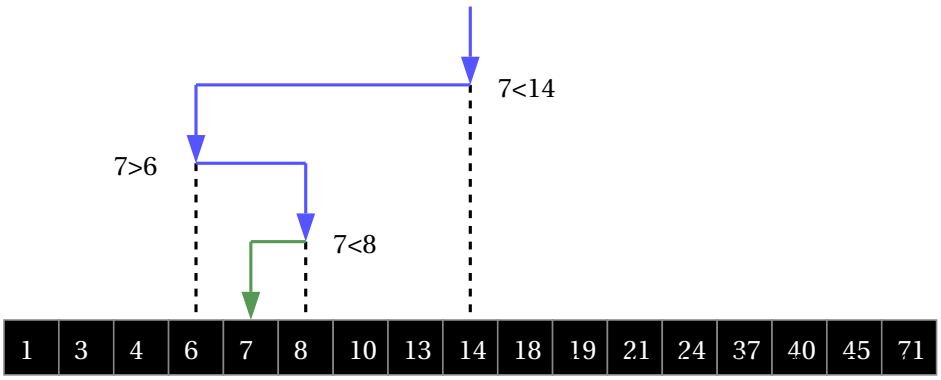

FIGURE 1 – Illustration de la recherche dichotomique de la valeur 7 dans un tableau trié (Source : Wikimedia Commons)

■ **Définition 1 — 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,
- 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.

À chaque tour de boucle, l'algorithme **??** divise par deux la taille du tableau considéré. À la fin de l'algorithme, le dernier sous-tableau considéré comporte un seul élément [1]. Supposons que la taille du tableau initial 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 à un sous-tableau de taille 1. 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.

1. ou zéro si l'élément n'appartient pas au tableau

Algorithme 2 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
```

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

Algorithme 3 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`.

C 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'opérations élémentaires de la recherche séquentielle.** Sur l'algorithme **??**, 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 **??** 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 propor-

tionnel 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 4 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
```

Ⓡ 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.** Il suffit pour cela de choisir une constante qui majore toutes les autres.

Algorithme 5 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 **??**, 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.** La première partie de l'algorithme **??** compte les occurrences de chaque élément. La complexité de cette opération est $O(n)$ puisqu'il s'agit de balayer tout le tableau d'entrée à chaque fois.

La seconde partie de l'algorithme **??** est composée deux boucles imbriquées qui dépendent de v_{max} et de n. La boucle imbriquée effectue c[v] tours à chaque fois que le nombre est présent dans le tableau. On considère que le test et les deux opérations dans la boucle imbriquée ont un coût constant de 1. On peut compter le nombre d'opérations pour les deux boucles imbriquées :

$$C(n, v_{max}) = \sum_{v=0}^{v_{max}} \left(1 + \sum_{j=0}^{c[v]-1} 1\right) \tag{10}$$
$$= \sum_{v=0}^{v_{max}} 1 + c[v] \tag{11}$$
$$= 1 + v_{max} + n \tag{12}$$

car $\sum_{j=0}^{c[v]-1} 1 = c[v]$ et $\sum_{v=0}^{v_{max}} c[v] = n$.

C'est pourquoi la complexité du tri par comptage est en $O(n+1+v_{max}+n) = O(v_{max}+n)$

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