Recherche dichotomique dans un tableau trié

S6 - Algorithmique (1)

On s'intéresse ici au problème de la recherche d'une valeur dans un tableau que l'on supposera **triée** dans l'ordre croissant.

1. Approche naïve

La première idée qui peut venir à l'esprit est de considérer les éléments du tableau les uns après les autres et de les comparer avec l'élément recherché.

On peut ainsi écrire une fonction `recherche_naive` qui prend en paramètre un tableau `tàbleau` et une valeur `valeur` et qui renvoie l'indice de la première occurrence de `valeur` dans `tableau` ou `-1` si `valeur` n'est pas dans `tableau`.

```
def recherche_naive(tableau, valeur):
    for i in range(len(tableau)):
        if tableau[i] == valeur:
            return i
    return -1
```

Test de cette fonction :

```
recherche_naive([1, 2, 3, 4, 5], 3)
```

```
2
```

```
recherche_naive([1, 2, 3, 4, 5], 6)
```

```
-1
```

La complexité dans le pire des cas correspond ici au cas où la valeur recherchée n'est pas dans la liste. Il faut alors parcourir toutes les valeurs du tableau et faire n comparaisons, où n est la taille du tableau. L'algorithme naïf a donc une **complexité linéaire** en $\mathcal{O}(n)$.

Il est possible d'être plus efficace en exploitant le fait que la liste est triée.

2. Recherche dichotomique

Le principe

De façon intuitive, la recherche dichotomique consiste à **diviser par deux** la zone de recherche à chaque étape.

On commence par considérer l'ensemble des éléments du tableau. On regarde ensuite la valeur de l'élément du **milieu** du tableau. Si cette valeur est inférieure à la valeur recherchée, alors on poursuit la recherche sur la moitié supérieure du tableau. Si la valeur est supérieure à la valeur recherchée, alors on poursuit la recherche sur la moitié inférieure. Si la valeur est égale à la valeur recherchée, on a trouvé l'élément recherché.

! Principe de l'algorithme

- On considère le tableau `tableau` trié dans l'ordre croissant dans lequel on recherche la valeur `valeur`.
- On définit les bornes `gauche` et `droite` du tableau : indices du premier et du dernier élément de la partie du tableau dans laquelle on recherche.
- Tant que `gauche` est inférieur ou égal à `droite` :
  - On calcule l'indice `milieu` du milieu du tableau.
  - Si `tableau[milieu]` est égal à `valeur`, on renvoie `milieu`.
  - Si `tableau[milieu]` est inférieur à `valeur`, on met à jour `gauche` à `milieu + 1`. Lors de l'itération suivante, on ne considèrera donc que la partie du tableau située à droite de `milieu`.
  - Si `tableau[milieu]` est supérieur à `valeur`, on met à jour `droite` à `milieu - 1`. Lors de l'itération suivante, on ne considèrera donc que la partie du tableau située à gauche de `milieu`.
- On renvoie `-1` si `valeur` n'est pas dans `tableau`.

Exemple

On considère le tableau `[1,4,7,10,13,16,19,22,25]` et on cherche la valeur 22.

On commence par considérer l'ensemble des éléments du tableau (`gauche=0` et `droite=8`). On regarde ensuite la valeur de l'élément du **milieu** du tableau. Ici, il s'agit de l'élément d'indice 4 (`milieu=4`), qui vaut 13. La valeur recherchée est supérieure à 13, donc on ne considère que la partie du tableau située à droite de l'élément du milieu.

On répète alors l'opération sur la partie du tableau située à droite de l'élément du milieu (`gauche=5` et `droite=8`). On obtient le tableau `[16,19,22,25]` et on cherche la valeur 22. L'élément du milieu de ce tableau vaut 19 (`milieu=6`), qui est inférieur à 22. On ne considère donc que la partie du tableau située à droite de l'élément du milieu.

On répète donc l'opération sur la partie du tableau située à droite (`gauche=7` et `droite=8`). On obtient le tableau `[22,25]` et on cherche la valeur 22. L'élément du milieu de ce tableau vaut 22 (`milieu=7`), qui est égal à 22. On a donc trouvé la valeur recherchée et l'algorithme est terminé en trois étapes.

Programmation

Écrivons une fonction `recherche_dichotomique` qui prend en paramètre un tableau `tableau` et une valeur `valeur` et qui renvoie l'indice d'une occurrence de `valeur` dans `tableau` ou `-1` si `valeur` n'est pas dans `tableau`.

```python
def recherche_dichotomique(tableau, valeur):
    gauche = 0
    droite = len(tableau) - 1
    while gauche <= droite:
        milieu = (gauche + droite) // 2
        if tableau[milieu] == valeur:
            return milieu
        elif tableau[milieu] < valeur:
            gauche = milieu + 1
```

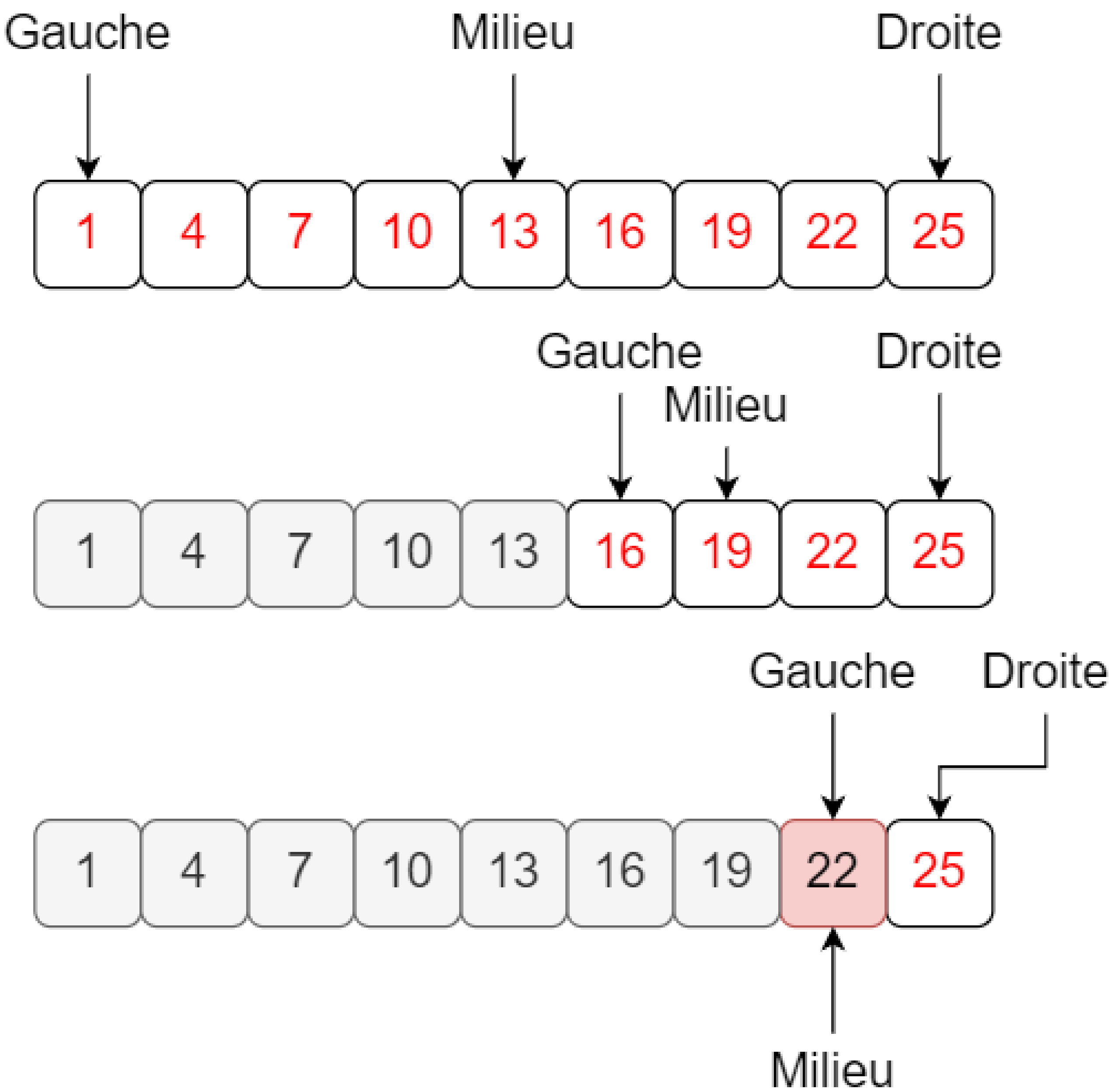

Figure 1: Recherche dichotomique

```
        else:
            droite = milieu - 1
    return -1
```

Test de cette fonction :

```
recherche_dichotomique([1, 2, 3, 4, 5], 2)
```

```
1
```

```
recherche_dichotomique([1, 2, 3, 4, 5], 6)
```

```
-1
```

Preuve de terminaison

Pour prouver que l'algorithme se termine, il faut prouver que la boucle `while` se termine, et donc que la condition d'arrêt `gauche > droite` finit par être vérifiée **ou bien** que la condition `tableau[milieu] == valeur` est vérifiée.

Choisissons comme **variant de boucle** la différence `droite - gauche` et plaçons-nous dans le cas le plus défavorable où la condition `tableau[milieu] == valeur` n'est jamais vérifiée. À chaque passage dans la boucle, soit `gauche` est incrémenté de 1, soit `droite` est décrémenté de 1. La différence `droite - gauche` est donc toujours diminuée de 1. Au bout d'un nombre fini d'itérations (au maximum égal à la taille du tableau moins un), la condition notre variant de boucle devient donc égal à zéro et la boucle se termine.

La terminaison de l'algorithme est donc prouvée.

Preuve de correction

Considérons la propriété suivante : à chaque étape de l'algorithme, la valeur recherchée est située dans la partie du tableau située entre les indices `gauche` et `droite` inclus, ou bien elle n'est pas dans le tableau.

Montrons que cette propriété est un **invariant de boucle**. C'est-à-dire que cette propriété est vraie avant l'exécution de la première itération de la boucle et qu'elle est vraie après l'exécution de chaque itération de la boucle.

Initialisation : avant l'exécution de la première itération de la boucle, si la valeur est dans le tableau, alors elle est située dans la partie du tableau située entre les indices `gauche` et `droite` inclus. En effet, `gauche` vaut 0 et `droite` vaut la taille du tableau moins un, donc la valeur recherchée est située dans la partie du tableau située entre les indices 0 et la taille du tableau moins un inclus (c'est le tableau entier !).

Conservation : supposons que la propriété est vraie à l'entrée dans une itération de la boucle `while`. Il y a trois possibilités :

- `tableau[milieu] == valeur` est vérifiée. Dans ce cas, la propriété est vraie à la sortie de l'itération de la boucle et l'algorithme retourne l'indice attendu.
- `tableau[milieu] < valeur` est vérifiée. Dans ce cas, `gauche` est incrémenté de 1. La valeur recherchée est donc située dans la partie du tableau située entre les indices `gauche` et `droite` inclus, ou bien elle est absente du tableau.

- `tableau[milieu] > valeur` est vérifiée. Dans ce cas, `droite` est décrémenté de 1. La valeur recherchée est donc située dans la partie du tableau située entre les indices `gauche` et `droite` inclus, ou bien elle est absente du tableau.

Conclusion : la propriété est donc un invariant de boucle.

Deux cas sont à considérer pour conclure. Si le test `tableau[milieu] == valeur` est vérifié au cours des itérations, alors la valeur est trouvée dans le tableau et on retourne son indice `milieu` : c'est bien le comportement attendu. Si le test `tableau[milieu] == valeur` n'est jamais vérifié, alors l'algorithme se termine lorsque `gauche > droite`. D'après notre invariant de boucle, soit la valeur est alors absente du tableau, soit elle est située dans la partie du tableau située entre les indices `gauche` et `droite` inclus. Mais cette partie du tableau est un tableau vide `[]`. La valeur est donc absente du tableau et on retourne l'indice `-1` : c'est bien le comportement attendu.

L'algorithme est donc correct.

Complexité

Pour évaluer la complexité de cet algorithme, nous allons évaluer le nombre d'itérations nécessaires en fonction de la taille n du tableau, dans le pire des cas, c'est-à-dire lorsque la valeur recherchée n'est pas dans le tableau.

À chaque étape, la taille du sous-tableau contenant potentiellement la valeur recherchée est divisée par deux. Au bout de k étapes, la taille du sous-tableau est donc de $\frac{n}{2^k}$ environ. Si la valeur recherchée n'est pas dans le tableau, alors on finit par arriver à un tableau de taille 1 et la boucle se termine au tour suivant. Soit k le nombre d'itérations nécessaires pour que la taille du sous-tableau soit de 1. On a donc $\frac{n}{2^k} \approx 1$, et par conséquent $n = 2^k$. On en déduit que $k = \log_2 n$.

L'algorithme de recherche dichotomique est donc en $\mathcal{O}(\log n)$. On parle de **complexité logarithmique**. Cette complexité est meilleure que la complexité linéaire.

> **Notion de logarithme de base 2**
>
> Si x est une puissance de 2, alors $\log_2 x$ est égal à l'**exposant** de cette puissance. Par exemple, $\log_2 8 = 3$ car $8 = 2^3$.

Pour un tableau de départ de taille $16 = 2^4$ dans lequel on cherche une valeur qui n'y est pas, on effectue 4 itérations :

- au premier tour, on divise le tableau en deux parties de taille $8 = 2^3$ et on cherche dans la partie de gauche (par exemple) ;
- au deuxième tour, on divise la partie de gauche en deux parties de taille $4 = 2^2$ et on cherche dans la partie de gauche (par exemple) ;
- au troisième tour, on divise la partie de gauche en deux parties de taille $2 = 2^1$ et on cherche dans la partie de gauche (par exemple) ;
- au quatrième tour, on divise la partie de gauche en deux parties de taille $1 = 2^0$ et on cherche dans la partie de gauche (par exemple).

On retrouve bien un nombre d'itérations de l'ordre de $\log_2 n$.

Comparaison expérimentale des deux algorithmes

```
import timeit
import matplotlib.pyplot as plt

tailles = [i for i in range(1, 500)]
temps_naive = []
temps_dicho = []
# on applique la recherche dans le pire des cas : valeur absente su tableau
valeur = 1000
for n in tailles:
    temps_naive.append(timeit.timeit(
        "recherche_naive([k for k in range(n)], valeur)",
        globals=globals(),
        number=100
    ))
    temps_dicho.append(timeit.timeit(
        "recherche_dichotomique([k for k in range(n)], valeur)",
        globals=globals(),
        number=100
    ))
plt.plot(tailles,temps_naive, 'b', label="Recherche naïve")
plt.plot(tailles,temps_dicho, 'r', label="Recherche dichotomique")
plt.xlabel("Taille du tableau")
plt.ylabel("Temps d'exécution (en secondes)")
plt.legend()
plt.show()
```