Recherches

Objectifs

- Mettre en œuvre et analyser des techniques de recherches sur tableau
- Connaître et savoir rédiger l'algorithme de recherche par dichotomie

Introduction

Rechercher une information au sein d'un ensemble de données est une tâche fondamentale en informatique, présente au cœur d'applications très diverses : bases de données, moteurs de recherche, intelligence artificielle, optimisation, etc.

Un algorithme de recherche est une méthode systématique permettant de localiser un élément dans une structure de données, que ce soit dans un tableau, une liste, un arbre, ou un graphe.

L'efficacité des algorithmes de recherche conditionne directement les performances des logiciels lorsqu'on manipule de grands volumes de données. Il est donc essentiel de comprendre à la fois la logique de ces algorithmes (séquentiels, dichotomiques) et leur complexité, afin de choisir l'approche la plus adaptée au contexte rencontré.

Dans ce chapitre, nous explorerons et étudierons certaines méthodes de recherche sur des tableaux unidimensionnels.

Recherche naïve : séquentielle

Principe

On parcourt le tableau en comparant chaque élément avec la valeur cible. On s'arrête dès qu'on la trouve et on renvoie l'indice du tableau correspondant, sinon on affiche un message d'erreur et on renvoie -1.

Pseudo-code

```
ALGORITHME RechercheSequentielle(T, cible):
DONNEES
  T : tableau d'entiers de taille n
  cible : entier
VARIABLES:
  i : entier
DEBUT
  i ← 1
  TQ i   n FAIRE
    SI T[i] = cible ALORS
      RENVOYER i
    FSI
    i ← i + 1
  FTQ
  AFFICHER("Cible non trouvée")
  RENVOYER -1
FIN
```

Arrêt

Il y a une seule boucle dépendante de i, celle-ci est incrémentée de 1 à chaque passage, donc dépassera la valeur n à un certain moment.

L'algorithme s'arrête.

Validité

De manière assez évidente, cet algorithme renvoie le premier indice qui correspond à la valeur cible et renvoie -1 sinon.

Complexité

Meilleur cas : La valeur cible est en tête de tableau, la complexité est en $\Theta(1)$.

Pire cas : La valeur cible n'est pas dans le tableau, la complexité est en $\Theta(n)$.

Conclusion : La complexité moyenne est donc en $O(n)$.

! Remarque

Cet algorithme montre que la complexité maximale pour une recherche séquentielle est en $O(n)$.
Peut-on faire mieux ?

Binary search : la dichotomie

Principe

Si le tableau est **trié**, on peut faire mieux en le découpant en deux à chaque tour et en ne gardant que la partie qui contient la valeur cible.

C'est le même principe qui permet de deviner un nombre entre 0 et 100, en n'ayant comme indice uniquement "mon nombre est plus petit" ou "mon nombre est plus grand".

i Exemple : Recherche de 15

Soit le tableau trié : `[3, 5, 7, 9, 11, 13, 15]` (indices 1 à 7)
Étape 1 : - Espace de recherche : indices 1 à 7 - $g = 1$, $d = 7$, $m = \lfloor (1+7)/2 \rfloor = 4$ - $T[4] = 9$ - Comparaison : $15 > 9 \rightarrow$ La cible est dans la moitié droite - Nouveau sous-tableau : indices 5 à 7
Étape 2 : - Espace de recherche : indices 5 à 7 - $g = 5$, $d = 7$, $m = \lfloor (5+7)/2 \rfloor = 6$ - $T[6] = 13$ - Comparaison : $15 > 13 \rightarrow$ La cible est dans la moitié droite - Nouveau sous-tableau : indices 7 à 7
Étape 3 : - Espace de recherche : indice 7 - $g = 7$, $d = 7$, $m = \lfloor (7+7)/2 \rfloor = 7$ - $T[7] = 15$ - Comparaison : $15 = 15 \rightarrow$ **Trouvé !** - Renvoyer 7
En seulement **3 comparaisons** au lieu de 7 avec la recherche séquentielle !

⚠ Pseudo-code

```
ALGORITHME Dichotomie(T, cible):
DONNEES
  T : tableau d'entiers de taille n
  cible : entier
VARIABLES
  g, d, m : entiers
DEBUT
  g ← 1
  d ← n
  TQ g  d FAIRE
```

```
    m ← (d+g)/2
    SI T[m] = cible ALORS
      RENVOYER m
    SINON
      SI T[m] > cible ALORS
        d ← m - 1
      SINON
        g ← m + 1
      FSI
    FSI
  FTQ
  AFFICHER("Cible non trouvée")
  RENVOYER -1
FIN
```

Arrêt

La boucle principale est fonction des variables d et g. On peut exprimer la condition d'arrêt ainsi : $g - d < 0$.

Or $g - d$ va prendre les valeurs de la suite (u_k) définie par $u_0 = n - 1$ et pour tout $k \geq 0$, $u_{k+1} < u_k$.

Soit k un entier, dans la boucle, $m = \lfloor \frac{d+g}{2} \rfloor \leq \frac{d+g}{2}$.

Cas 1 : Si on passe dans l'instruction $d \leftarrow m - 1$, alors :

$$u_{k+1} = (m-1) - g \leq \frac{d+g}{2} - 1 - g = \frac{d+g-2g}{2} - 1 < \frac{d-g}{2} = \frac{u_k}{2} < u_k$$

Cas 2 : Si on passe dans l'instruction $g \leftarrow m + 1$, alors :

$$u_{k+1} = d - (m+1) \leq d - \frac{d+g}{2} - 1 = \frac{2d-d-g}{2} - 1 < \frac{d-g}{2} = \frac{u_k}{2} < u_k$$

Ainsi, u_k est une suite strictement décroissante d'entiers naturels, donc elle atteint nécessairement une valeur négative, ce qui prouve l'arrêt de l'algorithme.

Validité

Posons la propriété $\mathcal{P}(k)$ suivante pour l'entier $k \geq 0$:

$\mathcal{P}(k)$: "Si la cible est dans le tableau, alors elle est entre les indices g et d à l'itération k"

- **Initialisation :** Au début de l'algorithme, $g = 1$ et $d = n$, donc si la cible est dans le tableau, elle est nécessairement entre les indices 1 et n. $\mathcal{P}(0)$ est vraie.
- **Hérédité :** Supposons $\mathcal{P}(k)$ vraie. À l'itération $k + 1$:
  - Si $T[m] = cible$, l'algorithme s'arrête et renvoie m (correct)
  - Si $T[m] > cible$, alors l'indice g ne change pas et comme le tableau est trié, la valeur cible ne peut pas être à l'indice m ou au-dessus. Donc la valeur cible est entre les indices g et $m - 1$ qui devient la nouvelle valeur de d
  - Si $T[m] < cible$, de manière symétrique, la cible est entre $m + 1$ (nouvelle valeur de g) et d

Dans tous les cas, $\mathcal{P}(k + 1)$ est vraie.

- **Conclusion :** Par récurrence, $\mathcal{P}(k)$ est vraie pour tout $k \geq 0$. L'algorithme maintient l'invariant et trouve la cible si elle existe, ou renvoie -1 sinon.

Complexité

À chaque itération, la taille de l'intervalle de recherche est **divisée par 2**.

Si on note $T(n)$ le nombre d'itérations nécessaires dans le pire cas pour un tableau de taille n :

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

En résolvant cette récurrence (ou en raisonnant directement) :

- Après 1 itération : taille $\frac{n}{2}$
- Après 2 itérations : taille $\frac{n}{4}$
- Après 3 itérations : taille $\frac{n}{8}$
- ...
- Après k itérations : taille $\frac{n}{2^k}$

L'algorithme s'arrête quand la taille devient inférieure à 1, soit quand :

$$\frac{n}{2^k} \leq 1 \Rightarrow n \leq 2^k \Rightarrow k \geq \log_2(n)$$

Conclusion : La complexité de la recherche dichotomique est en $\Theta(\log n)$.

Comparaison des algorithmes

Algorithme	Condition	Meilleur cas	Pire cas	Moyen	Remarque
Séquentielle	Aucune	$\Theta(1)$	$\Theta(n)$	$O(n)$	Simple, fonctionne toujours
Dichotomie	**Tableau trié**	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	Très efficace sur grands tableaux

Exemple concret : - Pour $n = 1\,000\,000$ éléments : - Recherche séquentielle : jusqu'à 1 million de comparaisons - Recherche dichotomique : environ $\log_2(1\,000\,000) \approx 20$ comparaisons !

Points clés

- La dichotomie nécessite un **tableau trié** pour fonctionner
- Si on doit trier le tableau avant de chercher, le coût total est $O(n \log n)$ pour le tri + $O(\log n)$ pour la recherche
- Pour une seule recherche sur un tableau non trié, la recherche séquentielle est plus efficace
- Pour de nombreuses recherches sur le même tableau, il est rentable de le trier une fois puis d'utiliser la dichotomie

Applications et extensions

La recherche dichotomique est à la base de nombreux algorithmes :

- **Recherche dans des structures triées** : arbres binaires de recherche
- **Optimisation** : recherche du minimum/maximum d'une fonction unimodale
- **Algorithmes de décision** : "trouver la plus petite valeur qui satisfait une propriété"
- **Compression de données** : codage par intervalles

Le principe “diviser pour régner” de la dichotomie se généralise à de nombreux problèmes algorithmiques.