Recherche dichotomique [re04] - Exercice

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 21 mai 2018

Table des matières

Python - Recherche dichotomique (Solution)

Mots-Clés Algorithmes de recherche ■
Requis Axiomatique impérative sauf Fichiers, Preuve et Notations asymptotiques ■
Fichiers UtilsRech ■
Difficulté • ○ ○

Objectif
Cet exercice réalise l'algorithme de la recherche dichotomique, prouve l'algorithme et calcule sa complexité. Dans le même ordre d'idées, l'exercice @[Recherche séquentielle] construit un algorithme dans le cas où le multi-ensemble n'est pas trié.

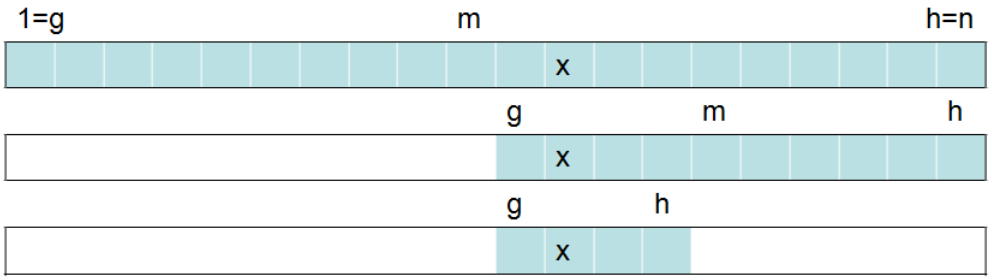
1 Algorithme de la recherche dichotomique

Soit une structure tabulaire `A[1..n]` triée en ordre croissant. On effectue une recherche dichotomique d'une valeur `x` comme suit.
Soient :

- `g` l'indice de gauche initialisé à 1.
- `h` l'indice de droite initialisé à `n`.
- `trouve` le booléen de la recherche initialisé à `Faux`.
- `m` l'indice milieu de l'intervalle `[g..h]`.

Alors :

1. Si `A[m]=x` alors on fixe `trouve` à `Vrai`.
2. Sinon si `A[m]<x` alors la position de `x` est forcément après `m` d'où on fixe `g` en `m+1`. Dans le cas contraire, on fixe `h` en `m-1` (car `A[m]>x`).
3. On répète les opérations (1) et (2) jusqu'à ce que `trouve` soit `Vrai` ou que `g` soit plus grand que `h`.



2 Algorithmique, Programmation

2.1 Rappel : Représentation et opérations

Soient la définition et les opérations de base sur une `Sequence` :

Soient la définition et les opérations définies sur une `Sequence` :

```
MAXELEMS = ... # Taille maximale de la Sequence
class Sequence:
   self.taille = ... # nombre d'éléments dans la Sequence
   def ieme(self,k): # Valeur du k-ième élément de la Sequence A
   def fixerIeme(self,k,valeur): # Fixe la valeur du k-ième élément de A à valeur
   def afficherSeq(self): # Affiche la Sequence A
   def saisirSeq(self): # Saisie une Sequence dans A
   def permuterSeq(self,j,k): # Permute les éléments en position j et k de A
```

Téléchargez le fichier suivant et mettez-le dans votre dossier.
Python @[Sequence.py]

Copiez/collez ensuite la ligne suivante :

Python Au début de votre programme :

```
from Sequence import Sequence
```

2.2 Séquence d'éléments

On considère un multi-ensemble représenté par le type `Sequence` (structure tabulaire) sur lequel est défini une relation d'ordre `<` et que l'on suppose *trié* en ordre croissant. On cherche à construire un algorithme permettant de savoir à quel endroit se trouve une valeur `x`.

On suppose que l'`Element x` est dans la séquence. Écrivez une fonction `rechDicho1(A,x)` qui effectue une recherche dichotomique de `x` dans une `Sequence A` et qui renvoie l'indice d'une occurrence (pas forcément la première) de `x`.

Validez votre fonction avec la solution.

Solution Python @[pgdichoseq.py]

```
def rechDicho1(A,x):
   trouve = False
   g,h = 0,A.taille-1
   while (not trouve):
      m = (g+h)//2
      if (A.ieme(m) == x):
         trouve = True
      elif (A.ieme(m) < x):
         g = m+1
      else:
         h = m-1
   return m
```

Exécutez votre algorithme sur les données suivantes :
`A=[1,7,8,9,12,15,15,22,30,31]` et `x=15`.

Solution simple

On représente les valeurs des variables `g`, `h`, `m|`, `\lstinline`A[m]|, `trouve` à l'initialisation et à la fin de chaque itération (F signifie `Faux` et V signifie `Vrai`).

	initialisation	fin itération 1	fin itération 2	fin itération 3
g	1	1	6	6
h	10	10	10	7
m	-	5	8	6
A[m]	-	12	22	15
trouve	F	F	F	V

Comment faut-il modifier l'algorithme si l'on n'est pas sûr que `x` appartienne à la séquence ?

Aide simple

Il faut ajouter la possibilité de terminer la boucle quand `g>h` ce qui signifie que `x` n'est pas dans la séquence.

Copiez/collez la fonction `rechDicho1` en la fonction `rechDicho2(A,x)` puis modifiez-la de sorte que la fonction renvoie -1 en cas de recherche infructueuse.

Validez votre fonction avec la solution.

Solution Python @[pgdichoseq.py]

```
def rechDicho2(A,x):
   trouve = False
   g,h = 0,A.taille-1
   m = -1
   while (g <= h and not trouve):
      m = (g+h)//2
      if (A.ieme(m) == x):
         trouve = True
      elif (A.ieme(m) < x):
         g = m+1
      else:
         h = m-1
   return m if trouve else -1
```

3 Preuve et complexité

On considère une structure tabulaire `A` trié.

Indiquez un invariant de boucle pour cet algorithme.

Solution simple

Considérons l'invariant de boucle suivant : « A la fin de la k^e itération, soit $A[m]$ contient la valeur x, soit l'intervalle $[g..h]$ a diminué (par rapport à l'itération précédente) et le sous-tableau $A[g..h]$ contient la valeur x ».

L'invariant précédent s'écrit :

$$\begin{array}{ll} \text{soit :} & A[m_k] = x \text{ et } trouve_k = Vrai \\ \text{soit :} & g_k \leq h_k \text{ et } x \in t[g_k..h_k] \text{ et } (g_k > g_{k-1} \text{ ou } h_k < h_{k-1}) \end{array}$$

Cet invariant n'est vrai que si le tableau A est trié et si x est bien un élément du tableau. Il permet de prouver à la fois la terminaison et la validité de l'algorithme. En effet, à chaque itération :

- Soit on a trouvé la valeur x.
- Soit on obtient un intervalle strictement plus petit que celui de l'itération précédente, dont on est sûr qu'il contient la valeur x. Au pire, l'algorithme se terminera sur un sous-tableau $A[g_k..h_k]$ réduit à un seul élément avec $A[g_k] = A[h_k] = x$.

On suppose que le tableau contient $n = 2^k$ éléments (où k est un entier positif). Combien d'itérations l'algorithme effectuera-t-il au maximum ?

Solution simple

A chaque itération, l'algorithme compare la valeur `x` avec l'élément central du tableau `A[g..h]`. Si ces deux éléments sont différents, à l'itération suivante le tableau de travail est de taille inférieure à la moitié de celle de l'itération courante. Exemple : pour `k=3` et `x=8` avec `A=[1,2,3,4,5,6,7,8]`, l'exécution de l'algorithme passera par les sous-tableaux : `[...,5,6,7,8]` puis `[ ...,7,8]` puis `[...,8]` et effectuera donc k boucles avant de tomber sur un sous-tableau de longueur 1 contenant forcément la valeur `x=8`.

Dans le cas général, on peut facilement prouver que l'algorithme effectuera au maximum $k = \log_2 n = \lg n$ itérations pour arriver à un tableau de taille 1.

Déduisez la complexité en O de l'algorithme.

Solution simple

D'après la question précédente l'algorithme est donc en

$$O(k) = O(\lg n) \subset O(\log n)$$

La complexité de la recherche dichotomique d'existence de `x` dans `A` est donc logarithmique si les comparaisons entre `Element`s s'effectuent en temps constant. Si ces comparaisons sont effectuées en temps linéaire, la complexité de la recherche dichotomique est presque linéaire, $O(k \log_2 n)$, où k est la complexité de la comparaison de `x` avec un `Element`s.

Comparez les complexités des algorithmes de recherche séquentielle et dichotomique pour $k = 100$. Calculez les temps d'exécution pour une machine capable d'effectuer 1 million de boucles en 1 seconde.

Solution simple

Pour $k = 100$ (et dans le pire des cas) l'algorithme séquentiel effectuera $2^{100} \approx 10^{30}$ itérations alors que l'algorithme dichotomique n'en effectuera que 100. Si l'on considère une machine capable d'effectuer 1 million de boucles en 1 seconde, l'algorithme séquentiel prendra 3×10^{16} années de calcul alors que l'algorithme dichotomique fournira le résultat en moins d'une milliseconde.

4 Références générales

Comprend [Felea-PG1 :c3 :ex77] ■