Méthode « diviser pour régner »

Cours

La méthode « **diviser pour régner** » consiste à découper un problème en sous-problèmes similaires de plus en plus petits jusqu'à obtenir des cas simples permettant une résolution directe. Elle consiste en trois étapes :

1. **Diviser** : découper un problème initial de **taille n** en **sous-problèmes indépendants de taille n/2** (ou une fraction de n);
2. **Régner** : résoudre les sous-problèmes (récursivement ou directement s'ils sont assez petits) ;
3. **Combiner** : calculer une solution au problème initial à partir des solutions des sous-problèmes.

Cette méthode tire souvent avantage de la **récursivité**. Le **faible coût** des algorithmes diviser pour régner est l'un de leurs principaux intérêts.

Cette technique fournit des algorithmes efficaces pour de nombreux problèmes, comme la recherche dichotomique d'un élément dans un tableau trié, le tri fusion, la multiplication de grands nombres (algorithme de Karatsuba), etc.

Recherche dichotomique dans un tableau trié

Considérons un algorithme naïf de recherche dans un tableau en parcourant tous les éléments du tableau :

```
def recherche(x, T):
    for elt in T:
        if x == elt: return True
    return False
```

Dans le pire des cas (x n'est pas dans le tableau), l'algorithme parcourt l'ensemble du tableau, le coût est donc en $O\ n$.

Le principe de la **recherche dichotomique dans un tableau trié** est celui suivi naturellement par les enfants quand ils jouent à un jeu bien connu : un des joueurs doit découvrir en un minimum d'essais un nombre secret compris entre 0 et 100 choisi par l'autre joueur. A chaque proposition du permier joueur, l'autre lui répond s'il a trouvé le nombre secret ou s'il est plus petit ou plus grand. La meilleure technique pour le joueur qui cherche le nombre secret consiste à proposer un nombre « au milieu » de tous les nombres possibles.

Au début le joueur propose le nombre au milieu entre 0 et 100, c'est-à-dire 50.

- Si on lui répond « gagné », il a eu de la chance et il a trouvé le nombre secret immédiatement.
- Si on lui répond « perdu, c'est plus grand », alors il sait que le nombre secret est entre 51 et 100, il va donc proposer le nouveau milieu entre 51 et 100, c'est-à-dire 75.
- Si la réponse est « perdu, c'est moins », alors le nombre secret est entre 0 et 49, il va proposer 25.

Il va continuer ainsi de suite jusqu'à trouver le nombre secret. Cette technique consiste à **diviser un problème en deux sous-problèmes indépendants**, c'est un algorithme du type **diviser pour régner**.

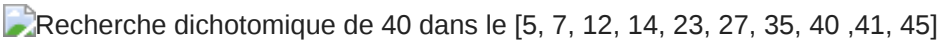
Recherche dichotomique de `x` dans un tableau **trié** `T` :

Etape	Description
Diviser	Découper le tableau trié `[T[debut], T[debut+1], ..., T[fin]]` en son milieu (`(debut + fin)//2`) pour avoir deux sous-tableaux `[T[debut], T[debut+1], ..., T[milieu-1]]` et `[T[milieu + 1], ..., T[fin]]`
Régner	- si `x < T[milieu]` , chercher `x` dans `[T[debut], T[debut+1], ..., T[milieu-1]]` - si `x > T[milieu]` , chercher `x` dans `[T[milieu + 1], ..., T[fin]]` - si `x == T[milieu]` , `x` a été trouvé
Combiner	

Faisons par exemple des recherches dans le tableau **trié** [5, 7, 12, 14, 23, 27, 35, 40 ,41, 45].

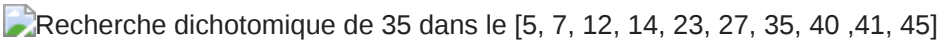
Plusieurs cas se présentent :

Recherche de la valeur 40


Recherche dichotomique de 40 dans le [5, 7, 12, 14, 23, 27, 35, 40 ,41, 45]

1. On cherche la valeur `40` dans le tableau `[5, 7, 12, 14, 23, 27, 35, 40 ,41, 45]` .
2. On partage le tableau en deux parties en son milieu : `T[milieu] = 23` .
3. `40 > T[milieu]` , on cherche la valeur `40` dans la partie supérieure du tableau `[27, 35, 40 ,41, 45]` .
4. On partage le tableau en deux parties en son milieu : `T[milieu] = 40` .
5. `40 = T[milieu]` , on a trouvé la valeur `40` .

Recherche de la valeur 35


Recherche dichotomique de 35 dans le [5, 7, 12, 14, 23, 27, 35, 40 ,41, 45]

1. On cherche la valeur `35` dans le tableau `[5, 7, 12, 14, 23, 27, 35, 40 ,41, 45]` .
2. On partage le tableau en deux parties en son milieu : `T[milieu] = 23` .
3. `35 > T[milieu]` , on cherche la valeur `35` dans la partie supérieure du tableau `[27, 35, 40 ,41, 45]` .
4. On partage le tableau en deux parties en son milieu : `T[milieu] = 40` .
5. `35 < T[milieu]` , on cherche la valeur `35` dans la partie inférieure du tableau `[27, 35]` .
6. On partage le tableau en deux parties en son milieu : `T[milieu] = 27` .
7. `35 > T[milieu]` , on cherche la valeur `35` dans la partie supérieure du tableau `[35]` .
8. On partage le tableau en deux parties en son milieu : `T[milieu] = 35` .
9. `35 = T[milieu]` , on a trouvé la valeur `35` .

On voit ici que la recherche s'effectue jusqu'à ce que le tableau n'ait plus qu'une seule valeur, c'est à dire que `debut` est égal à `fin` .

Recherche de la valeur 34

 Recherche dichotomique de 34 dans le [5, 7, 12, 14, 23, 27, 35, 40 ,41, 45]

1. On cherche la valeur `34` dans le tableau `[5, 7, 12, 14, 23, 27, 35, 40 ,41, 45]`.
2. On partage le tableau en deux parties en son milieu : `T[milieu] = 23`.
3. `34 > T[milieu]`, on cherche la valeur `34` dans la partie supérieure du tableau `[27, 35, 40 ,41, 45]`.
4. On partage le tableau en deux parties en son milieu : `T[milieu] = 40`.
5. `34 < T[milieu]`, on cherche la valeur `34` dans la partie inférieure du tableau `[27, 35]`.
6. On partage le tableau en deux parties en son milieu : `T[milieu] = 27`.
7. `34 > T[milieu]`, on cherche la valeur `34` dans la partie supérieure du tableau `[35]`.
8. On partage le tableau en deux parties en son milieu : `T[milieu] = 35`.
9. `34 < T[milieu]`, on cherche la valeur `34` dans la partie inférieure du tableau `[]`
10. On n'a pas trouvé la valeur `34`.

On voit ici que la recherche s'effectue jusqu'à ce que le tableau soit vide , c'est-à-dire que `debut` est plus grand que `fin`.

Voilà ce que l'on peut écrire en mode itératif :

```
1  def recherche(x, T) :
2      debut = 0
3      fin = len(T) - 1
4      while debut <= fin:
5          milieu = (debut + fin)//2
6          if x < T[milieu]:
7              fin = milieu - 1
8          elif x > T[milieu]:
9              debut = milieu + 1
10         else:       # donc x == T[milieu]:
11             return True   # ou return milieu si on veut la position dans T
12
13     return False    # ou return None par exemple si on veut la position dans T
```

⚠️ Un bug classique est d'écrire `while debut < fin:` à la ligne 4, alors qu'on a vu que la recherche doit se poursuivre jusqu'à ce que `debut` soit plus grand que `fin`, c'est-à-dire quand le tableau est vide.

Ce programme contient une boucle `while`, il faut donc s'assurer qu'il se termine. Ici le variant de boucle est `fin - debut`. A chaque itération de boucle, on voit qu'il y a trois cas :

- `x < T[milieu]` : dans ce cas, `fin` devient `milieu - 1`, donc le variant décroit strictement ;
- `x > T[milieu]` : dans ce cas, `debut` devient `milieu + 1`, donc le variant décroit strictement ;
- `x == T[milieu]` : dans ce cas, l'instruction `return True` sort de la boucle et même de la fonction.

Tant qu'on est dans la boucle, le variant de boucle `fin - debut` décroit strictement, la boucle `while debut <= fin:` se terminera donc.

Pour prouver la correction de cet algorithme, on va utiliser la technique de l'invariant de boucle. Ici, un invariant de boucle est : si x est dans T alors `T[debut] <= x <= T[fin]`. Si l'invariant est vrai quand on entre dans la boucle, alors il y a les mêmes trois possibilités :

- `x < T[milieu]` : alors la recherche se poursuit dans `[T[debut], ..., T[milieu-1]]`, l'invariant est encore vrai quand on retourne dans la boucle;
- `x > T[milieu]` : alors la recherche se poursuit dans `[T[milieu+1], ..., T[fin]]`, l'invariant est encore vrai quand on retourbne dans la boucle ;
- `x == T[milieu]` : alors on l'a trouvé.

On a donc bien un invariant de boucle et l'algorithme trouve bien si une valeur est dans un tableau trié ou pas.

Etudions la complexité temporelle pour un tableau de taille n. A chaque itération de la boucle on divise la taille du tableau par 2, cela revient donc à se demander combien de fois faut-il diviser la taille du tableau par 2 pour obtenir dans le cas le plus défavorable (`x` n'est pas dans `T`) un tableau vide ? Cela revient à trouver le nombre a tel que ^{a} n. La solution est a log n .

Cours

La **complexité en temps de l'algorithme de recherche dichotomique est logarithmique en** O log n .

On peut bien sûr écrire le même algorithme en mode récursif, en passant en paramètre de la fonction les valeurs de `debut` et `fin`. Les paramètres sont des paramètres facultatifs par mot-clé, ils sont initialisés à `0` et `lent(T)-1` au premier appel.

```
1  def recherche(x, T, debut=None, fin=None):
2
3      # initialisation de debut et fin au premier appel
4      if debut is None or fin is None:
5          debut = 0
6          fin = len(T)-1
7
8      if debut > fin:
9          return False
10     else:
11         milieu = (debut + fin)//2
12         if x < T[milieu]:
13             return recherche(x, T, debut, milieu-1)
14         elif x > T[milieu]:
15             return recherche(x, T, milieu+1, fin)
16         else:
17             return True
18
19 assert recherche(35, [5, 7, 12, 14, 23, 27, 35, 40 ,41, 45])
20 assert not recherche(34, [5, 7, 12, 14, 23, 27, 35, 40 ,41, 45])
```