Chapitre XII : Algorithmique

I - Recherche dans un tableau

Dans cette partie, j'utiliserai deux langages de programmation : python et javascript. Cela ne devrait pas créer de difficulté supplémentaire car il s'agit de montrer que les algorithmes ne dépendent pas du langage.

a) Recherche séquentielle

On dispose d'un tableau (une liste en python) et on souhaite savoir si la valeur x appartient ou non à ce tableau et donner son index dans le cas où elle serait présente.

La première méthode consiste à parcourir le tableau par balayage : c'est la méthode séquentielle.

On crée une fonction recherche. En python, le tableau sera du type liste, alors qu'en javascript il sera du type Array.

En python	En Javascript

`def recherche(liste, x):`
`  i=0`
`  while i<len(liste) and x!=liste[i]:`
`  i=i+1`
`  if i<len(liste):`
`  return i`
	`function recherche(tab, x){`
	`  let i=0`
	`  while (i<tab.length && x!=tab[i]){`
	`  i=i+1`
	`  }`
	`  if (i<tab.length){`
	`  return i`
	`  } else {`
	`  return -1`
	`  }`

Remarque : si la valeur n'est pas présente dans la liste, **la fonction ne renvoie rien en python**. Le résultat affiché sera alors la valeur `None`. On pourrait en faire de même en javascript, mais on a choisi dans ce cas de renvoyer la valeur -1 (ce que renvoie `tab.indexOf(x)` en javascript).
On ne peut pas renvoyer -1 en python car cela signifierait `len(liste)-1`.

On peut s'intéresser au coût de cet algorithme, c'est-à-dire au nombre de comparaisons effectuées (toujours prendre en compte le pire des cas).
Si l'élément cherché `x` n'est pas dans le tableau, alors il faut parcourir les n éléments du tableau. Le coût de cet algorithme est donc linéaire c'est-à-dire de l'ordre de n.

En python et en javascript, il est possible d'utiliser une boucle pour (mais cela ne change pas le coût de l'algorithme). Compléter les programme ci-dessous :

En python	En Javascript
`def recherche(liste, x):`	`function recherche(tab, x){`

La parcours séquentiel d'un tableau sert aussi bien pour rechercher un élément que pour rechercher une valeur maximale (ou minimale), calculer la somme des termes, ...

b) Recherche dichotomique dans un tableau trié

Le précédent algorithme fonctionne avec des tableaux quelconques. Lorsque les **données sont triées**, nous allons trouver un algorithme plus efficace.
Le principe est le même que celui consistant à deviner un nombre pris au hasard dans un intervalle par un partenaire lorsque celui-ci répond par « plus petit » ou « plus grand » en cas d'échec.
Pour être certain de trouver ce nombre dans un temps relativement court, il est judicieux de proposer la moyenne des bornes de l'intervalle.

Nous allons adapter cette méthode à la recherche d'une valeur `x` dans un tableau (nommé `liste` en python et `tab` en javascript). Nous le ferons avec des tableaux d'entiers, mais cela peut se faire avec n'importe quelles données qui admettent une relation d'ordre.

Le principe de l'algorithme :
La recherche se fera en plusieurs étapes successives, sans que nous sachions au départ combien d'étapes seraient nécessaires. Nous allons donc utiliser une boucle `Tant que`.

- Avant chaque étape, on saura que si `x` est dans le tableau, alors il sera entre deux index nommés `g` et `d` (pour gauche et droite).
- À chaque étape, on prendra l'index central (moyenne de g et d dont on prendra la partie entière car un index ne peut être un flottant) et on vérifiera si `x` est au-dessus ou en-dessous de cette valeur d'index central.
- À la fin de l'étape, si on ne trouve pas `x`, on actualise les valeurs de `g` et `d`. Le nombre d'index possibles pouvant contenir la valeur x est alors divisé par 2 par rapport à ce qu'il était au début.

Voici la traduction de cet algorithme en python et en javascript :

En python

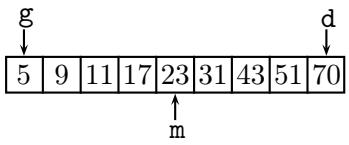
```python
def rechercheDicho(liste, x):
    g=0
    d=len(liste)-1
    while g<=d:
      m=(g+d)//2
      if x==liste[m]:
        return m
      elif x<liste[m]:
        d=m-1
      else :
        g=m+1
```

En Javascript

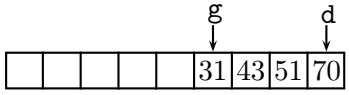
```javascript
function rechercheDicho(tab, x){
    let g=0
    let d=tab.length-1
    while (g<=d){
      let m=Math.floor((g+d)/2)
      if (x==tab[m]){
        return m
      } else if (x<tab[m]){
        d=m-1
      } else {
        g=m+1
      }
    }
    return -1
}
```

Appliquons cet algorithme à un exemple.
On cherche l'index de la valeur 31 dans le tableau [5, 9, 11, 17, 23, 31, 43, 51, 70].

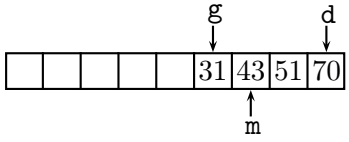
- Étape 1 : `g=0`, `d=8`, ainsi `g<=d` vaut `True` et on rentre dans la boucle.
  `m=4` comme ci-contre :



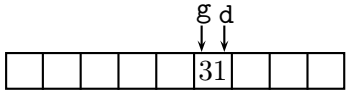
  Comme `x>liste[m]`, `g` devient `m+1` et la recherche se limite à :



- Étape 2 : `g=5`, `d=8`, ainsi `g<=d` vaut `True`
  `m=6` comme ci-contre :



  Comme `x<liste[m]`, `d` devient `m-1` :



- Étape 3 : `g=5`, `d=5`, ainsi `g<=d` vaut `True`
  `m=5` et `x==liste[m]` vaut True. Donc la fonction renvoie la valeur 5.

3 étapes ont été nécessaires pour trouver la valeur `x` alors qu'il en aurait fallu 6 avec l'algorithme de recherche par balayage.

Remarque : Cet algorithme n'a d'intérêt que si `x` peut se trouver dans ce tableau ordonnée, à savoir si `x` se situe entre `liste[0]` et `liste[-1]`. Cela peut être vérifié au début de la fonction par une instruction si ou une instruction `assert` en python. En javascript, il est aussi possible de lever des exceptions pour gérer les paramètres de fonctions.

Exemple : Appliquer l'algorithme à la liste [2, 3, 8, 10, 15, 19, 21, 23, 25, 29, 32, 35] et à la valeur x=22.
Donner les étapes successives et les valeurs prises par les différentes variables (comme dans l'exemple). Faire les schémas comme ceux de droite permet de ne pas oublier certains tests.
Que renvoie la fonction rechercheDicho dans ce cas ?

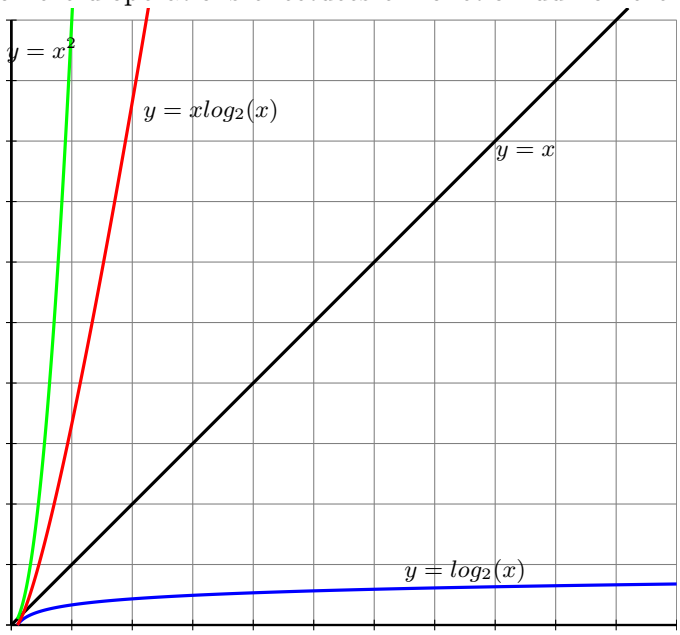
c) Coût de l'algorithme de recherche dichotomique

On s'intéresse de nouveau au nombre d'opérations (nombre de passages dans la boucle) dans le pire des cas. Cela se produit quand la valeur n'est pas présente dans le tableau.

Si nous partons d'un tableau de 100 valeurs, à l'issue du premier passage dans la boucle, la recherche ne s'effectue plus que sur 49 ou 50 valeurs, prenons 50 dans la pire des cas. À l'issue du deuxième passage, cela ne s'effectue plus que dans un tableau de 25 valeurs, puis de 12 valeurs, de 6, de 3, et enfin d'une seule valeur.
Ainsi, on fera au maximum 7 passages dans la boucle. Ce nombre 7 est le plus petit entier k vérifiant $2^k > 100$.
Combien de passages au maximum devra-t-on faire pour une liste de 3 millions de valeurs ?
On cherche le plus petit entier k tel que $2^k > 3\ 000\ 000$.
En cherchant pas à pas la valeur de k, on trouve $k = 22$. Donc même avec une liste de 3 millions d'éléments, 22 tests suffiront à trouver l'index de la valeur cherchée si elle est présente et affirmer qu'elle n'appartient pas au tableau dans le cas contraire.

Dans le cas général d'un tableau de taille n, le nombre maximal d'opérations nécessaires pour l'exécution de la fonction `rechercheDicho` est le plus petit entier k tel que $2^k > n$.
En mathématiques, l'équation $2^k = n$ d'inconnue k a pour solution le nombre noté $\log_2(n)$. Donc pour un nombre n de valeurs, le coût de l'algorithme de dichotomie est de l'ordre de $\log_2(n)$.

Pour comparer l'efficacité des différents algorithmes vus depuis le début de l'année, voici différentes courbes représentant en ordonnée y le nombre d'opérations effectuées en fonction du nombre de valeurs en abscisse x.



En conclusion, dans le cas d'une liste triée de n éléments, la recherche dichotomique (d'un coût de l'ordre de $\log_2(n)$) est beaucoup plus efficace qu'une recherche séquentielle (linéaire, c'est-à-dire de l'ordre de n).
Mais devoir trier les données n'est pas judicieux car le coût d'un tel algorithme est de l'ordre de $n \log_2(n)$ avec un algorithme efficace (comme celui de la fonction `sort` de python) voire de l'ordre de n^2 avec un tri par sélection ou insertion (dans le pire des cas pour ce dernier).

d) Validité de l'algorithme de recherche dichotomique

Nous devons vérifier les deux conditions :

- **la terminaison** : nous allons utiliser un **variant** de boucle, c'est-à-dire une expression (le plus souvent une variable) dont la valeur prend à un moment une valeur satisfaisant la condition d'arrêt de la boucle.

  Ici, prenons la valeur $d - g$ comme variant de boucle. On notera $d_k - g_k$ la valeur de cet écart au bout de k étapes (au départ, on a ainsi $d_0 = 0$ et g_0= `len(liste)-1`).

  La boucle s'arrête dès que $d_k - g_k < 0$. Il suffit donc de montrer que $d_k - g_k$ est une valeur entière strictement décroissante.

  À l'instant k, on calcule m_k qui est la partie entière de $\dfrac{d_k + g_k}{2}$.

  - si x est égal à la valeur d'index m_k, alors on sort de la boucle et l'algorithme se termine ;
  - si x est inférieur à la valeur d'index m_k, alors d_{k+1} vaut $m_k - 1$ et $g_{k+1} = g_k$. Comme $m_k - 1 < d_k$, on obtient $d_{k+1} - g_{k+1} < d_k - g_k$.

– si x est supérieur à la valeur d'index m_k, alors g_{k+1} vaut $m_k + 1$ et $d_{k+1} = d_k$. Comme $m_k \geqslant g_k$, on obtient $g_{k+1} > g_k$ et $d_{k+1} - g_{k+1} < d_k - g_k$.

Donc l'algorithme se terminera dans tous les cas de figure.

- **La correction** : il faut déterminer un invariant de boucle.

  À chaque étape $0 \leqslant g$; $d <=$ `len(liste)-1` et x ne peut se trouver qu'entre les index g et d.

  – Au départ, on s'assure (`assert` placé au début du programme) que x est bien entre les valeurs extrêmes du tableau : `liste[0]`$\leqslant x \leqslant$ `liste[len(liste)-1]`. Mais sans cette condition, la phrase x ne peut être qu'entre les index 0 et `len(liste)-1` reste correcte car la recherche se fait sur tout le tableau.

  – Si on suppose que `liste[g]`$\leqslant x \leqslant$ `liste[d]` à un certain instant k, alors on peut montrer que ce sera vrai à l'instant suivant $(k + 1)$. D'après les notations précédentes :

    – si x est la valeur d'indice m_k, alors on sort de la boucle et x est bien entre les index $g_{k+1} = g_k$ et $d_{k+1} = d_k$.

    – si x est inférieur à la valeur d'index m_k, alors deux cas de figure sont possibles :

      – si $g_k = d_k = m_k$, alors d_{k+1} vaudra $g_k - 1$ et g_{k+1} restera g_k. Dans ce cas, on sort de la boucle et on renvoie la valeur None : c'est le cas où x n'est pas présent (ce qui est bien le cas car il ne restait plus qu'une valeur possible).

      – Sinon, la plus grande valeur que pourrait prendre x est la valeur d'index $m_k - 1$, donc x ne pourra se trouver qu'entre les index $g_{k+1} = g_k$ et $d_{k+1} = m_k - 1$.

    – si x est supérieur à la valeur d'index m_k, alors on obtient le même type de raisonnement que précédemment (soit $g_k = d_k = m_k$ et on sort de la boucle, soit $g_{k+1} = m_k + 1$ et x se trouvera entre g_{k+1} et d_{k+1}).

  Nous avons donc bien un invariant de boucle.

Exercice : Un algorithme de recherche dichotomique met un temps t pour effectuer cette recherche dans un tableau de taille n et effectue k comparaisons.

1. Combien de comparaisons effectuera-t-il pour rechercher dans un tableau de $2n$ valeurs ?
2. Combien de valeurs contiendra un tableau pour lequel l'algorithme mettrait un temps égal à $2t$ pour effectuer une recherche ?
3. Si on met 15 ms pour rechercher dans un tableau de 1000 valeurs, combien en faudra-t-il pour un tableau d'un million de valeurs.