NSI - Première

Algorithmique : Recherche dichotomique dans un tableau trié

Le tableau `T` contient-il `x` ? À quelle position ?

*C'est un problème déjà rencontré lors des parcours séquentiels. Nous allons étudier un algorithme beaucoup plus rapide… mais qui ne s'applique qu'aux tableaux **triés**.*

Introduction : recherche par balayage

On a déjà abordé lors des *parcours séquentiels*

```
x = 5
T = [11,  7,  9,  5, 15, 13,  3,  1]
      0   1   2   3   4   5   6   7
```

```
def recherche(T, x):
    Pour i allant de 0 à len(T) - 1:
        Si x == T[i]:
            renvoyer i
    renvoyer -1
```

Déroulé

```
i = 0           5 != 11
i = 1           5 != 7
i = 2           5 != 9
i = 3           5 == 5
```

Remarques

1. Termine toujours (boucle bornée…)
2. **Au pire `len(T)` étapes**. La coût calculatoire d'un parcours séquentiel est *linéaire*.

Recherche dichotomique

Présentation

On suppose maintenant que le tableau `T` est **trié** par ordre croissant

```
x = 5
T = [ 1,  3,  5,  7,  9, 11, 13, 15]
      0   1   2   3   4   5   6   7
```

Stratégie du Jeu de "+ ou -" : viser le centre des éléments restant et éliminer la moitié des nombres à chaque étape.

Donc : comparer la valeur centrale à `x` et **éliminer la moitié des valeurs restantes**

Déroulé

1. Premier tour

```
x = 5
T = [ 1,  3,  5,  7,  9, 11, 13, 15]
      g           ^                d
```

```
g = 0, d = 7
m = (g + d) // 2 = (0 + 7) // 2 = 3
    x = 5 < T[3] = 7    =>   Chercher à gauche
```

   On recommence avec

   - `d = m - 1 = 2`
   - `g` inchangé

2. Second tour

```
x = 5
T = [ 1,  3,  5,  7,  9, 11, 13, 15]
      g   ^   d
```

```
g = 0, d = 2
m = (g + d) // 2 = (0 + 2) // 2 = 1
    x = 5 > T[1] = 3    =>   Chercher à droite
```

   On recommence avec

   - `d` inchangé
   - `g = m + 1 = 2`

3. Troisième tour

```
x = 5
T = [ 1,  3,  5,  7,  9, 11, 13, 15]
         g=d
```

```
g = 2, d = 2
m = (g + d) // 2 = (2 + 2) // 2 = 2
    x = 5 == T[2] = 5    =>   Trouvé !
```

   On peut renvoyer `2`.

Construction de l'algorithme

- **Précondition** ⇒ `T` un tableau trié par ordre croissant
- **Plusieurs étapes** ⇒ Il faut une boucle
- **Nombre inconnu d'étapes** ⇒ Boucle non bornée (`while`)
- **Arrêt** ⇒ `g > d` ⇒ `while g <= d:`

Corps de la boucle

- `m = (g + d) // 2`
- 3 cas :
  - Si `x == T[m]` ⇒ `return m`
  - Si `x < m` ⇒ `d = m - 1`

    – Si `x > m` ⇒ `d = m + 1`

- **Et si la boucle termine ?**
    – `T` ne contient pas `x` ⇒ `return -1`

Algorithme

```python
def recherche_dichotomique(T: list, x: list) -> int:
    """
    Renvoie l'indice de `x` dans `T`.
    Renvoie -1 si `x` n'est pas dans `T`.

    Précondition : `T` est trié par ordre croissant
    """
    g = 0
    d = len(T) - 1
    while g <= d:
        m = (g + d) // 2
        if x == T[m]:
            return m
        elif x > T[m]:
            g = m + 1
        else:
            d = m - 1
    return -1
```

Déroulé de l'algorithme

```python
T = [ 1,  3,  5,  7, 11, 13, 15]

def recherche_dichotomique(T, x):
    g = 0                               |       1. g = 0 <= d = 7
    d = len(T) - 1                      |          m = (0 + 7) // 2 = 3
    while g <= d:                       |         5 < 7  => d = 3 - 1 = 2
        m = (g + d) // 2                |
        if x == T[m]:                   |       2. g = 0 <= d = 2
            return m                    |          m = (0 + 2) // 2 = 1
        elif x > T[m]:                  |         5 > 3  => g = 1 + 1 = 2
            g = m + 1                   |
        else:                           |       3. g = 2 <= d = 2
            d = m - 1                   |          m = (2 + 2) // 2 = 2
    return -1                           |         5 == 5  => return 2
```

Remarques

Précondition

`T` doit être trié par ordre croissant

Terminaison

- `while d <= g:`
- `d - g` décroit strictement à chaque étape
- En nombre fini d'étapes, on arrive à `d > g` et l'algorithme s'arrête

Coût

- `d - g` est *grossièrement* divisé par deux à chaque étape

- Ex. Si `len(T)` $= 16 = 2^4$, il faut ~**4 étapes**.

Conclusion

- La recherche dichotomique permet de gagner beaucoup d'étapes par rapport au parcours séquentiel du tableau.
- Elle nécessite d'avoir un tableau **trié** sans quoi on ne peut l'appliquer.

Remarques sur le coût

- Si on ne souhaite l'appliquer qu'une seule fois, il n'est pas intéressant de trier de le tableau pour chercher. C'est généralement trop long.
- Mais si on doit souvent effectuer des recherches dans le tableau, alors c'est indispensable.

Nombre d'étapes

- **parcours séquentiel** : autant que d'éléments dans le tableau dans le pire des cas.

  Le parcours séquentiel prend (dans le pire des cas) n étapes.

- **recherche dichotomique** : $\log_2 n$ étapes.

  $\log_2 n \approx$ le nombre de divisions entières de n par 2 qu'on peut effectuer avant de trouver un quotient nul

  $\log_2 n \approx$ le nombre de bits de n en binaire.