

Institut Préparatoire aux Études d'Ingénieurs de Tunis



Leila Ben Othman

Première année de classes préparatoires
MP-PC

Année universitaire: 2016 - 2017

Algorithmes de Recherche et de Tri:

Application en PYTHON

Plan

- ① Les tableaux unidimensionnels
- ② Recherche séquentielle
 - Recherche du maximum/minimum
 - Recherche de la première occurrence
 - Recherche de toutes les occurrences d'un élément
- ③ Recherche récursive
- ④ Recherche dichotomique dans un tableau trié
 - Version itérative
 - Version récursive
- ⑤ Recherche d'un mot dans un texte
 - Algorithme naïf
- ⑥ Algorithmes de tri
 - Tri par sélection
 - Tri Par bulles
 - Tri par insertion
 - Tri rapide - Quick Sort

Tableaux à une dimension

- Un tableau à une dimension ou encore unidimensionnel est une structure de données linéaire.
- Un tableau est une suite de cases (cellules) contenant des éléments de même type et directement accessibles par leurs indices ou index.
- Chaque élément du tableau est accessible en lecture ou écriture.

Tableaux à une dimension

Pour définir un tableau, il faut préciser:

- le nom (identifiant du tableau)
- l'indice début et l'indice fin (généralement de type entier)
- le type des éléments (entier, réel, caractère, etc.)

Déclaration d'un tableau

On déclare un tableau de la manière suivante (exemple):

VAR

T : Tableau [1..100] d'entier

ou encore

CONST

NMAX = 100

TYPES

Tab=Tableau[1..NMAX] d'entier

VAR

T: Tab

La notation $T[i]$ fait référence à l'élément d'indice i .

Remplissage d'un tableau

Procédure remplir_tab(var T:tab, n:entier)

VAR

i:entier

Début

Pour i de 1 à n **faire**

Ecrire("Donner un entier")

Lire (T[i])

Fin pour

Fin

Les tableaux en PYTHON

On va utiliser la bibliothèque numpy.

- 1 Importer le module numpy;

```
import numpy as np
```

- 2 Définir un tableau (vide) dont les éléments sont de type *int*;

```
T=np.array([], 'int')
```

- 3 Modifier la taille du tableau;

```
T.resize(n)
```

remplissage d'un tableau - PYTHON

```
import numpy as np
def remplir_tab(n):
    T=np.array([], 'int')
    T.resize(n)
    for i in range(len(T)):
        T[i]=int(input("Donner un entier"))
    return(T)
```

NB:

- La fonction `remplir_tab(n)` remplit et retourne un tableau de `n` entiers.
- En PYTHON, `T[i]` désigne la case d'indice `i`.
- L'index de la première case commence par 0.
- `len(T)` retourne la taille du tableau.
- `T[len(T)-1]` désigne la dernière case du tableau.

Recherche du min/max

```
Fonction recherche_min(T:tab, n:entier):entier
VAR
    i,min:entier
Début
    min ← T[1]
Pour i de 2 à n faire
    Si T[i] < min Alors
        min ← T[i]
    FinSi
Fin pour
retourner(min)
Fin
```

Recherche du min/max - PYTHON

```
def recherche_min(T):  
    m=T[0]  
    for i in range(1,len(T)):  
        if T[i]<m:  
            m=T[i]  
    return(m)
```

Recherche du min/max

```
Fonction recherche_max(T:tab, n:entier):entier
VAR
    i,max:entier
Début
    max ← T[1]
Pour i de 2 à n faire
    Si T[i] > max Alors
        max ← T[i]
    FinSi
Fin pour
retourner(max)
Fin
```

Recherche du min/max

```
def recherche_max(T):  
    m=T[0]  
    for i in range(1,len(T)):  
        if T[i]>m:  
            m=T[i]  
    return(m)
```

Recherche de la première occurrence d'un élément

Problème: La recherche de la position de la première occurrence de l'élément x dans un tableau T .

Principe:

- Parcourir le tableau T;
- Comparer chaque élément à l'élément recherché (x);
- Si x est trouvé, **arrêter la recherche** et renvoyer sa position;
- Si x n'est pas trouvé (on est arrivé à la fin du tableau), retourner une valeur qui indique que x n'existe pas dans T (par exemple -99).

⇒ Quand l'élément est trouvé, il faut arrêter la recherche!

Fonction recherche_pos_Version1(T:tab, x, n:entier):entier

VAR

i,pos:entier trouver:boolean

Début

trouver $\leftarrow$ Faux i $\leftarrow$ 1

Tant que Non Trouver Et $i \leq n$ faire

Si T[i]=x **Alors**

 Trouver $\leftarrow$ Vrai

 pos $\leftarrow$ i

Sinon

 i $\leftarrow$ i+1

FinSi

Fin Tant que

Si Trouve **Alors**

 retourner(pos)

Sinon

 retourner(-99)

FinSi

Fin

Fonction recherche_pos_Version2(T:tab, x, n:entier):entier

VAR

 i,pos:entier

Début

pos $\leftarrow$ -99

i $\leftarrow$ 1

Tant que pos=-99 **Et** i $\leq$ n **faire**

Si T[i]=x **Alors**

 pos $\leftarrow$ i

Sinon

 i $\leftarrow$ i+1

FinSi

Fin Tant que

retourner(pos)

Fin

```
Fonction recherche_pos_Version3(T:tab, x, n:entier):entier
VAR
    i:entier
Début
    i ← 1
Tant que T[i] <> x Et i ≤ n faire
    i ← i+1
Fin Tant que
Si i = n+1 Alors
    retourner(-99)
Sinon
    retourner(i)
Fin
```

Recherche de la première occurrence - PYTHON

```
def recherche_pos_Version1(T,x):  
    i=0  
    trouve=False  
    while not trouve and i<len(T):  
        if T[i]==x:  
            trouve=True  
            pos=i  
        else:  
            i=i+1  
    if trouve:  
        return(pos)  
    else:  
        return(-99)
```

Recherche de la première occurrence - PYTHON

```
def recherche_pos_Version2(T,x):  
    i=0  
    pos=-99  
    while pos==-99 and i<len(T) :  
        if T[i]==x:  
            pos=i  
        else:  
            i=i+1  
    return(pos)
```

Recherche de la première occurrence - PYTHON

```
def recherche_pos_Version3(T,x):  
    i=0  
    while i<len(T) and T[i]!=x:  
        i=i+1  
    if i==len(T):  
        return -99  
    else:  
        return i
```

Recherche de la première occurrence - PYTHON

- L'instruction `break` dans une boucle `for` provoque la sortie de la boucle. Il devient alors possible d'employer une boucle `for` pour la recherche de la position de la première occurrence d'un élément.
⇒ Toute instruction après l'instruction `break` se trouvant dans la boucle ne sera pas exécutée.

```
def recherche_pos_Version4(T,x):
    pos=-99
    for i in range(len(T)):
        if T[i]==x:
            pos=i
            break
    return(pos)
```

Recherche de la première occurrence - PYTHON

- L'instruction `return` provoque l'arrêt de la fonction et le retour du résultat. Ainsi, une instruction `return` dans une boucle `for` provoque la sortie de la boucle et l'arrêt de la fonction.
⇒ Toute instruction après l'instruction `return` ne sera exécutée.

```
def recherche_pos_Version5(T,x):  
    pos=-99  
    for i in range(len(T)):  
        if T[i]==x:  
            return(i)  
    return(pos)
```

Recherche de toutes les occurrences d'un élément

Problème: La recherche de toutes les positions de x dans un tableau T .

Principe:

- Parcourir le tableau;
- Comparer chaque élément à l'élément recherché (x);
- Si l'élément est l'élément recherché, alors stocker son indice;
- Si l'élément n'est pas l'élément recherché, on continue la recherche

⇒ **Quand un élément est trouvé, il ne faut pas arrêter la recherche. Car on cherche toutes les positions de l'élément!**

Recherche de toutes les occurrences d'un élément

- Cette procédure construit un tableau composé des différentes positions de x dans T ainsi que sa taille.

Procédure recherche_occurences(T :tab, n :entier, var Tab_pos :tab, var $taille$:entier)

VAR

i, j :entier

Début

$j \leftarrow 0$

Pour i de 1 à n **faire**

Si $T[i]=x$ **Alors**

$j \leftarrow j+1$

$Tab_pos[j] \leftarrow i$

FinSi

Fin pour

$taille \leftarrow j$

Fin

Recherche de toutes les occurrences d'un élément - PYTHON

- Cette fonction retourne un tuple composé des différentes positions de x dans T .

```
def recherche_occurrences_tuple(T,x):
    tup_pos=()
    for i in range(len(T)):
        if T[i]==x:
            tup_pos=tup_pos+(i,)
    return(tup_pos)
```

Recherche de toutes les occurrences d'un élément - PYTHON

- Cette fonction retourne un tableau composé des différentes positions de x dans T.

```
def recherche_occurences_tab(T,x):  
    tab_pos=np.array([], 'int')  
    j=0  
    for i in range(len(T)):  
        if T[i]==x:  
            tab_pos.resize(len(tab_pos)+1)  
            tab_pos[j]=i  
            j=j+1  
    return(tab_pos)
```

Recherche récursive

Problème: La recherche de la position de la première occurrence d'un élément x dans un tableau T de façon récursive.

Principe:

- L'idée est d'utiliser un paramètre d qui désigne l'indice début du tableau à partir duquel on va commencer la recherche.
- A chaque appel récursif, on teste si le premier élément du tableau $T[d]$ est égale à x ;
- Si la condition est vérifiée, **arrêter les appels récursifs** et renvoyer d comme position.
- Si la condition n'est pas vérifiée, deux cas peuvent se présenter:
 - si d dépasse la fin du tableau, cela veut dire que l'élément x ne se trouve pas dans le tableau. On **arrête les appels récursifs** en retournant alors une valeur qui indique que x n'existe pas dans T .
 - Sinon on continue la recherche récursive en incrémentant l'indice de début à partir duquel on effectue la recherche.

Recherche récursive

```

Fonction recherche_recursive(T:tab, x, n:entier, d:entier):entier
VAR
Début
Si d=n+1 Alors
    retourner(-99)
Sinon
    Si T[d]=x Alors
        retourner(d)
    Sinon
        retourner(recherche_recursive(T,x,n,d+1))
    FinSi
FinSi
Fin
    
```

Recherche récursive - PYTHON

```
def recherche_recursive(T,x,d):  
    if d==len(T):  
        return(-99)  
    elif T[d]==x:  
        return(d)  
    else:  
        return(recherche_recursive(T,x,d+1))
```

NB: Le premier appel à la fonction doit évidemment se faire avec un paramètre $d=0$ (pour que la recherche commence à partir du premier élément du tableau).

$n = 5$

$T = \text{remplir_tab}(n)$

$x = 10$

$pos = \text{recherche_recursive}(T, x, 0)$

$\text{print}(x, \text{"setrouvelaposition"}, pos)$

Recherche dichotomique dans un tableau trié

Principe: Considérons un tableau T constitué d'entiers **distincts triés** dans l'ordre croissant.

- Pour chercher x dans T , on commence par comparer x avec la valeur médiane de T . En notant m la valeur médiane, il s'agit de comparer x à $T[m]$:
 - si $x=T[m]$, on a terminé notre recherche: on retourne m comme position de x .
 - si $x>T[m]$, on continue notre recherche dans la partie droite du tableau ;
 - si $x<T[m]$, on continue notre recherche dans la partie gauche du tableau.
- La recherche dichotomique consiste à répéter ce procédé de façon **itérative** ou de façon **réursive**.
- **Il ne faut pas oublier d'arrêter la recherche lorsque x n'est pas trouvé.**

Recherche dichotomique dans un tableau trié

- Pour cela, il faudra employer deux indices d et f qui désignent respectivement l'indice début et l'indice fin à partir desquels on effectuera la recherche de x .
- La recherche de x s'effectue tant qu'on a au moins un élément dans le tableau délimité par les indices d et f .

Recherche dichotomique dans un tableau trié

Version Itérative

```
def recherche_dico_iterative(T,x):  
    d=0  
    f=len(T)-1  
    while(d<=f):  
        m=(d+f)//2  
        if x==T[m]:  
            return (m)  
        elif x>T[m]:  
            d=m+1  
        elif x<T[m]:  
            f=m-1  
    return(-99)
```

NB: les indices d et f sont des variables locales à la fonction recherche_dico_iterative.

Recherche dichotomique dans un tableau trié

Version récursive

```
def recherche_dico_recursive(T,x,d,f):  
    m=(d+f)//2  
    if d>f:  
        return -99  
    elif T[m]==x:  
        return m  
    elif x>T[m]:  
        d=m+1  
        return(recherche_dico_recursive(T,x,d,f))  
    else:  
        f=m-1  
        return(recherche_dico_recursive(T,x,d,f))
```

NB: Les indices d et f sont des paramètres de la fonction recherche_dico_recursive. Le premier appel à la fonction doit se faire avec d=0 et f=len(T)-1 (la totalité du tableau).

Recherche d'un mot dans un texte

- Nous nous intéressons à la problématique de recherche de sous-chaîne de caractères dans une chaîne de caractères, connue sous le nom de recherche de motifs.
- Un motif est une sous-chaîne de caractères, qu'on notera P . La chaîne à partir de laquelle se fera la recherche est un texte noté T .
- Nous nous intéressons à la position de la première occurrence du motif P dans le texte T s'il en existe une. Sinon, nous retournons la valeur -1.

Recherche d'un mot dans un texte

- En considérant que le texte est représenté par un tableau de caractères et le motif par un tableau de caractères.
- Le problème de la recherche de motifs consiste donc à trouver une sous-chaîne de T égale à P .
- Pour se faire, nous allons employer l'algorithme naïf. L'idée de l'algorithme naïf consiste à réaliser une comparaison caractère par caractère du texte T et du motif P jusqu'à trouver la séquence des caractères du motif P consécutivement dans T si elle existe.

Recherche d'un mot dans un texte

```
def recherche_motif_naive(T,P):  
    pos=-1  
    i=0  
    while pos==-1 and i<=len(T)-len(P):  
        j=0  
        while j<=len(P)-1 and T[i+j]==P[j]:  
            j=j+1  
        if j==len(P):  
            pos=i  
        else:  
            i=i+1  
    return(pos)
```

Tri par sélection

Principe

Le tri par sélection consiste à :

- chercher l'élément minimal et le placer en première position du tableau par permutation avec l'élément courant;
- recommencer avec le second plus petit élément, le placer en seconde position et ainsi de suite jusqu'à la fin du tableau.

NB: on peut également travailler avec le maximum et le placer en dernière position.

Tri par sélection - Exemple

9	3	10	2	5
----------	---	----	----------	---

2	3	10	9	5
----------	---	----	----------	---

2	3	10	9	5
---	----------	----	---	---

2	3	10	9	5
---	---	-----------	---	----------

2	3	5	9	10
---	---	----------	---	-----------

2	3	5	9	10
---	---	---	----------	----

Tri par sélection -PYTHON

```
def minimum(T, i):  
    pos=i  
    for j in range(i+1, len(t)):  
        if T[j] < T[pos]:  
            pos= j  
    return pos  
  
def tri_selection(T):  
    for j in range (len(T)-1):  
        i = minimum(T, j)  
        if i != j:  
            T[i], T[j] = T[j], T[i]
```

Tri par bulles

Principe

Le tri par bulles consiste à:

- comparer deux éléments consécutifs $T[i]$ et $T[i+1]$, les permuter si $T[i] > T[i+1]$;
- on recommence le traitement jusqu'à ce qu'il n'y ait plus de permutation.

Tri par bulles - Exemple

3	2	5	1
---	---	---	---

permutation

2	3	5	1
---	---	---	---

2	3	5	1
---	---	---	---

pas de permutation

2	3	5	1
---	---	---	---

permutation

2	3	1	5
---	---	---	---

2	1	3	5
---	---	---	---

on refait le traitement
permutation

1	2	3	5
---	---	---	---

1	2	3	5
---	---	---	---

pas de permutation

1	2	3	5
---	---	---	---

pas de permutation

2	3	1	5
---	---	---	---

on refait le traitement
pas de permutation

2	3	1	5
---	---	---	---

permutation

2	1	3	5
---	---	---	---

2	1	3	5
---	---	---	---

pas de permutation

Tri par bulles - Exemple

2	1	3	5
---	---	---	---

On refait le traitement
permutation

1	2	3	5
---	---	---	---

1	2	3	5
---	---	---	---

pas de permutation

1	2	3	5
---	---	---	---

pas de permutation

1	2	3	5
---	---	---	---

on refait le traitement
pas de permutation

1	2	3	5
---	---	---	---

pas de permutation

1	2	3	5
---	---	---	---

pas de permutation
arrêt - tableau trié
tout un parcours
sans aucune permutation

Tri par bulles

```
def tri_bulle(T):  
    trie=False  
    while (not trie):  
        permut=False  
        for i in range(len(T)-1):  
            if T[i]>T[i+1]:  
                T[i],T[i+1]=T[i+1],T[i]  
                permut=True  
        trie=not permut
```

Tri par insertion

Principe

- Le tri par insertion consiste à parcourir le tableau et à insérer chaque élément à sa bonne position (comme dans un jeu de carte).
- Pour savoir si un élément est à sa bonne position, il faut le comparer avec celui qui le précède. Si l'élément est inférieur à celui qui le précède, il faudra les permuter. Ce traitement est répété jusqu'à ce l'élément soit bien placé (c'est-à-dire supérieur à celui qui le précède).

Tri par insertion - Exemple

9	3	10	2	5
----------	---	----	---	---

9	3	10	2	5
---	----------	----	---	---

3	9	10	2	5
----------	---	----	---	---

3	9	10	2	5
---	---	-----------	---	---

3	9	10	2	5
---	---	----	----------	---

2	3	9	10	5
----------	---	---	----	---

2	3	9	10	5
---	---	---	----	----------

2	3	5	9	10
---	---	----------	---	----

NB: Attention, cet exemple ne montre pas les étapes intermédiaires des différentes permutations effectuées.

Tri par insertion - PYTHON

```
def tri_insertion(T):  
    for i in range(1,len(T)):  
        j=i  
        while j>0 and T[j]<T[j-1]:  
            T[j-1],T[j]=T[j],T[j-1]  
            j=j-1
```

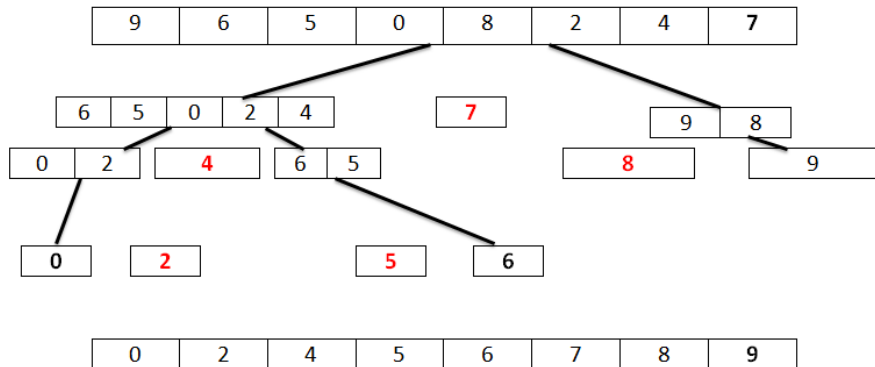
Tri rapide (Quick Sort)

- Le tri rapide consiste à appliquer le principe **diviser pour régner**.
- Cet algorithme consiste à segmenter (partitionner) le tableau autour d'un pivot (élément du tableau) en plaçant les éléments qui lui sont inférieurs à sa gauche et les éléments qui lui sont supérieurs à sa droite.
⇒ Ainsi, le pivot est placé à sa bonne position (sa place définitive).
- La partie gauche et la partie droite sont ensuite triées en appliquant le même principe récursivement.

NB: L'algorithme tri rapide présente plusieurs implémentations: la différence concerne essentiellement le choix du pivot ainsi que la fonction qui effectue la segmentation (partitionnement).

Tri rapide (Quick Sort)

Choix du pivot: le pivot est le dernier élément du tableau.



Tri rapide (Quick Sort)

- Nous aurons besoin de deux fonctions:
 - la fonction `quick_sort`: effectue le tri
 - la fonction `segmentation`: sélectionne un pivot (dernier), place les éléments inférieurs au pivot à sa gauche et les éléments supérieurs à sa droite et retourne la nouvelle position du pivot.
- Les deux fonctions utilisent deux indices `d` (début) et `f` (fin) pour délimiter les parties du tableau à traiter.
- La fonction `quick_sort` est récursive et se déroule que si le tableau comporte au moins deux éléments ($d < f$)
- La fonction `quick_sort` fait appel à la fonction `segmentation`, qui effectue le placement des éléments comme indiqué et retourne la nouvelle position du pivot `pos_pivot`. Ensuite, la fonction `quick_sort` effectue le tri récursivement sur la partie droite `quick_sort(T, pos_pivot+1, f)` ainsi que sur la partie gauche du tableau `quick_sort(T, d, pos_pivot-1)`.

Tri rapide (Quick Sort)

La fonction `quick_sort` est alors définie comme suit:

```
def quick_sort(T,d,f):  
    if d < f:  
        pos_pivot = segmentation(T, d, f)  
        quick_sort(T, d, pos_pivot-1)  
        quick_sort(T, pos_pivot + 1, f)
```

Tri rapide (Quick Sort)

La segmentation (le partitionnement)

- Le pivot est initialisé par le dernier élément du tableau
`pivot=T[f];`
- Une variable `a` est initialisée par `d`;
- On parcourt le tableau à partir de `d` (avec un compteur `b`) et on compare `t[b]` avec le pivot :
- Si `T[b]<pivot` on permute `t[a]` et `t[b]` et on incrémente la variable `a`.
- Une fois arrivé à la fin du tableau, on permute `t[f]` et `t[a]` : le tableau est alors partitionné comme souhaité. On retourne à la fin la valeur de la variable `a`, qui représente la nouvelle position du pivot.

Tri rapide (Quick Sort)

La fonction segmentation est alors définie comme suit:

```
def segmentation(T,d,f):  
    pivot=T[f]  
    a=d  
    for b in range(d,f):  
        if T[b]<pivot:  
            T[b],T[a]=T[a],T[b]  
            a=a+1  
    T[a],T[f]=T[f],T[a]  
    return(a)
```

Tri rapide (Quick Sort)

Segmentation - Exemple

9	6	5	0	8	2	4	7
----------	---	---	---	---	---	---	----------

$a=0, b=0$

9	6	5	0	8	2	4	7
---	---	---	---	---	---	---	----------

$a=0 \quad b=1$

6	9	5	0	8	2	4	7
---	---	---	---	---	---	---	----------

$a=1 \quad b=2$

6	5	9	0	8	2	4	7
---	---	---	---	---	---	---	----------

$a=2 \quad b=3$

Tri rapide (Quick Sort)

Segmentation - Exemple

6	5	0	9	8	2	4	7
---	---	---	---	---	---	---	----------

a=3 b=4

6	5	0	9	8	2	4	7
---	---	---	---	---	---	---	----------

a=3 b=5

6	5	0	2	8	9	4	7
---	---	---	---	---	---	---	----------

a=4 b=6

6	5	0	2	4	9	8	7
---	---	---	---	---	---	---	----------

a=5 arrêt

