Université de Sciences et de la Technologie Houari Boumediene

Faculté Informatique

Rapport Projet 2

TP 2 Algorithmique et complexité

La recherche d'éléments

Mahmoud Bacha rabah sami G_02

Saadallah Abderahim G_03

M1 RSD 2023/2024

Environnement de travail :

Les algorithmes seront tous exécuter sur le même échantillon de taille de tableau et aussi sur les mêmes machines pour avoir une comparaison équitable

La bibliothèque de python **numpy** sera utilisé pour la création des tableaux.

- Langage : python 3.9
- CPU: i5 8365U 3.9 GHZ max frequency
- Ram: 8 GB
- OS: windows 11
- Logical: VScode Jupiter notebook

A. Recherche d'un element

Algorithme 1 : Recherche séquentiel dans un tableau non trié

Implémentation :

```
def recherche_NONtrie(tableau,valeur):
    end= len(tableau)
    for i in range(end):
        if tableau[i]==valeur:
            return i
    return None
```

Explication:

- L'algorithme prend comme argument le tableau et la valeur recherché
- Parcourir tous les éléments du tableau et vérifier pour éléments s'il est égal à la valeur
  On retourne l'indice du première apparition s'elle existe et NULL sinon.

Complexité :

- Meilleur cas : **O (1)** si la valeur recherchée se retrouve à la première case du tableau, on ne fera qu'une seule itération

- Pire cas : **O(N)** si la valeur ne se trouve pas dans le tableau
  Dans ce cas on va parcourir tous les éléments du tableau, la boucle finie quand on dépasse la taille du tableau, donc **N** itération

Algorithme 2 : Recherche séquentielle dans un tableau trié

Implémentation :

```
def recherche_trie(tableau , valeur):

    if(valeur > tableau[-1]):
        return None

    end= len(tableau)
    i = 0

    while(tableau[i] < valeur):
            i +=1

    if(tableau[i]==valeur):
        return i
    return None
```

Explication :

- L'algorithme prend comme argument le tableau et la valeur recherché
- Le tableau est trié par ordre croissant donc si la valeur dépasse le dernier élément, on retour NULL
  Sinon :
- Parcourir les éléments du tableau jusqu'à ce que l'élément du tableau dépasse la valeur recherchée, dans ce cas, soit l'élément du tableau est égal à la valeur, sinon la valeur n'existe pas

0	1	2	3	4	5	6	7	8	9
23	36	100	187	211	324	891	1002	2911	9233

Supposons Valeur = 200 : On va parcourir jusqu'à indice 4 , puis on test si 200 est égal tab[4]

Supposons Valeur = 10000 : On va sortir dans le premier test car on dépasse la borne sup de tableau

Supposons Valeur = 100 : On va parcourir jusqu'à indice 2 , puis on test si 100 est égal tab[2]

Complexité :

- Meilleur Cas : O(1) Quand la valeur se trouve dans la première case ou bien quand la valeur est supérieure a la dernier cas du tableau O(1) car on fait un seul test
- Pire Cas : O(N) Quand la valeur est le dernier élément du tableau , dans ce cas on doit parcourir tout le tableau et donc N itération

Algorithme 3 : La Recherche Dichotomique (Binary Search)

Implémentation :

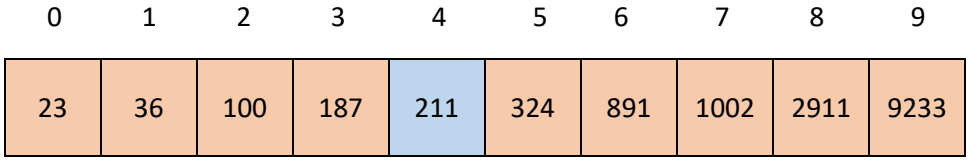
```
def recherche_dichotomique(tableau , valeur):

        deb = 0
        fin = len(tableau)-1

        while( deb <= fin):

            mid = int((fin+deb)/2)

            if valeur==tableau[mid]:
                return mid
            elif valeur > tableau[mid]:
                deb = mid+1
            else:
                fin= mid-1

        return None
```

Explication :

Le principe de ce algorithme est de réduire la taille du tableau dans lequel on recherche par 2 pour chaque itération

- L'algorithme prend comme argument le tableau et la valeur recherché
- On teste la valeur avec la case du moitié du tableau **tab[n/2]**
    - Si elle est égal a la valeur alors on retourne l'indice
    - Si elle est supérieur alors on va recherché dans la moitié droite du tableau

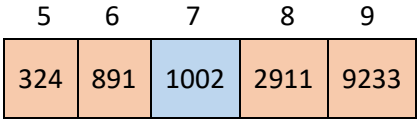
- Si elle est inférieur alors on va recherché dans la moitié gauche du tableau



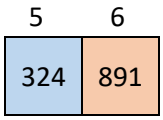
Milieu = 9+1 /2

Supposons que valeur = 221

1) Itération 1 : 221 > 211



2) Itération 2 : 221 < 1002



3) Itération 3 : 221 < 324
   Fin = 5-1 = 4
   Deb = 5 <=4 est fausse donc on arrête l'itération

Complexité :

- $Meilleur\ cas : \boldsymbol{O(1)}\ si\ valeur\ se\ trouve\ au\ milieu\ du\ tableau\ ,on\ arrete\ a\ la\ premiere\ itérat$
- Pire cas : **O(Log(N))** si la valeur ne se trouve pas dans le tableau
  Démonstration :
  Dans la dernier itération la taille du tableau devient 1
  et la taille du tableau diminue par division sur 2 dans chaque itération
  donc après **K** itération , $\frac{N}{2^K} = 1$ donc $N = 2^K$
  en appliquant la fonction $\log_2 N = K$

  Donc On fera $\log_2 N$ iteration

Etude expérimentale et représentation de résultats:

Meilleur cas :

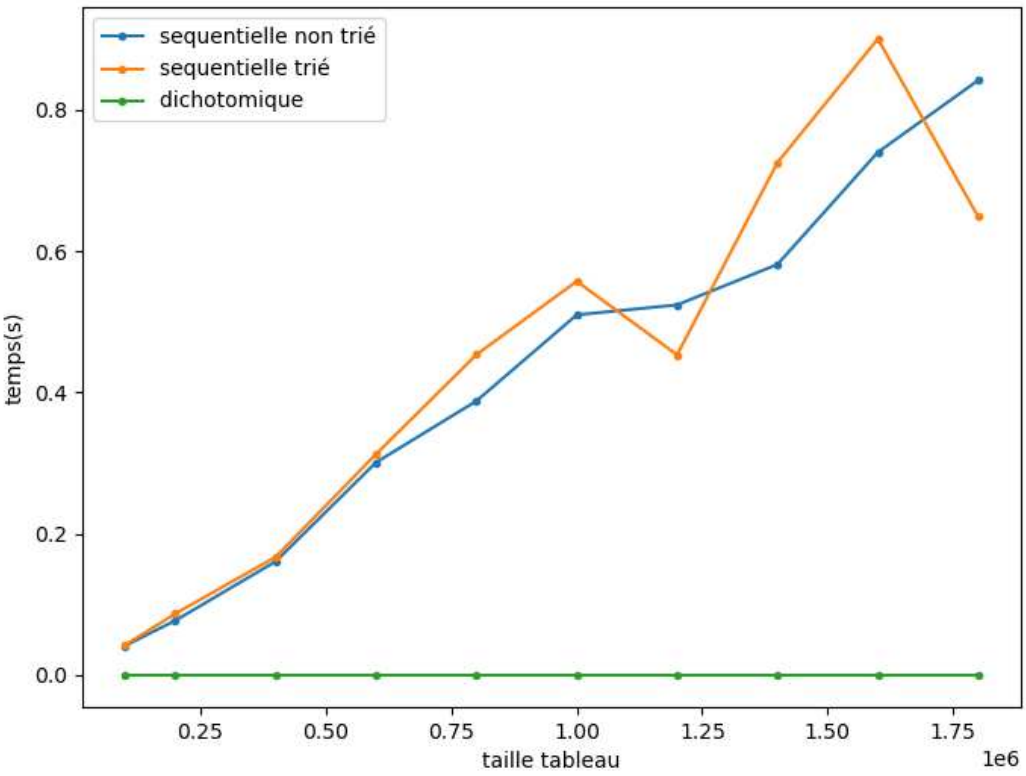
	100000	200000	400000	600000	800000	1000000	1200000	1400000	1600000	1800000
Non trié	0	0	0	0	0	0	0	0	0	0
Trié	0	0	0	0	0	0	0	0	0	0
Dichotomique	0	0	0	0	0	0	0	0	0	0

La complexité de tous ses algorithmes au meilleurs cas est **O(1)** donc le temps de calcul est très minim par rapport a la machine

Pire Cas :

	100000	200000	400000	600000	800000	1000000	1200000	1400000	1600000	1800000
Non trié	0.0416	0.0773	0.1609	0.3011	0.388	0.50993	0.52405	0.58137	0.74006	0.84195
Trié	0.0427	0.0869	0.1673	0.3127	0.4536	0.55752	0.45331	0.7255	0.90049	0.64873
Dichotomique	0	0	0	0	0	0	0	0	0	0

Representation Graphique de T(N) :



Remarque :

- On constate que la recherche dichotomique est très puissant par rapport au deux autres algorithmes et donc une complexité de **O($\log_2 N$)** est très optimal par rapport a **O(N)**
- L'application de l'algorithme 1 sur un tableau trié ou non trié reste la même car il fait dans les deux cas les mêmes test sur tous le tableau
- L'algorithme 2 est une petite amélioration de l'algorithme 1 car , le pire cas peut être détecté dés la première itération et donc on évite le cas ou la valeur n'existe pas
  mais si la valeur existe dans les derniers éléments du tableau , le temps de calcul sera plus grand qu'a l'algorithme 1 , car l'algo 2 fais plus de test

B. Max et Min :

Algorithme 1 : MaxminA

Implementation :

```
def maxmin(tableau):

    cpt = 0
    max = tableau[0]
    min = tableau[0]
    end = len(tableau)

    for i in range(1,end):
        cpt+=2
        if max < tableau[i]:
            max=tableau[i]
        if min > tableau[i]:
            min = tableau[i]


    return {'min' : min , 'max' : max , 'compteur' : cpt}
```

Complexité :

- L'algorithme fait N-1 itération , dans chaque itération on a deux a test a faire
- **Nombre de test = 2N – 2**
- **Complexité : O(N)**

Algorithme 2 : MaxminB

Implémentation :

```
def maxmin_2(tableau):

    cpt=0
    end= len(tableau)-1


    for i in range(0,end,2):
        cpt+=1
        if tableau[i]<tableau[i+1]:
            tableau[i] , tableau[i+1] = tableau[i+1] , tableau[i]



    min = tableau[1]
    max = tableau[0]

    for i in range(2,end,2):
        cpt+=2
        if tableau[i] > max:
            max = tableau[i]
        if tableau[i+1] < min :
            min = tableau[i+1



    # treaiter le cas ou taille tableau est impair et comaprer avec la
derniere case
    cpt+=1
    if (len(tableau)%2==1):
        cpt+=2
        if(min > tableau[-1]):
            min = tableau[-1]
        if(max < tableau[-1]):
            max = tableau[-1]


    return {'min' : min , 'max' : max , 'compteur' : cpt }
```

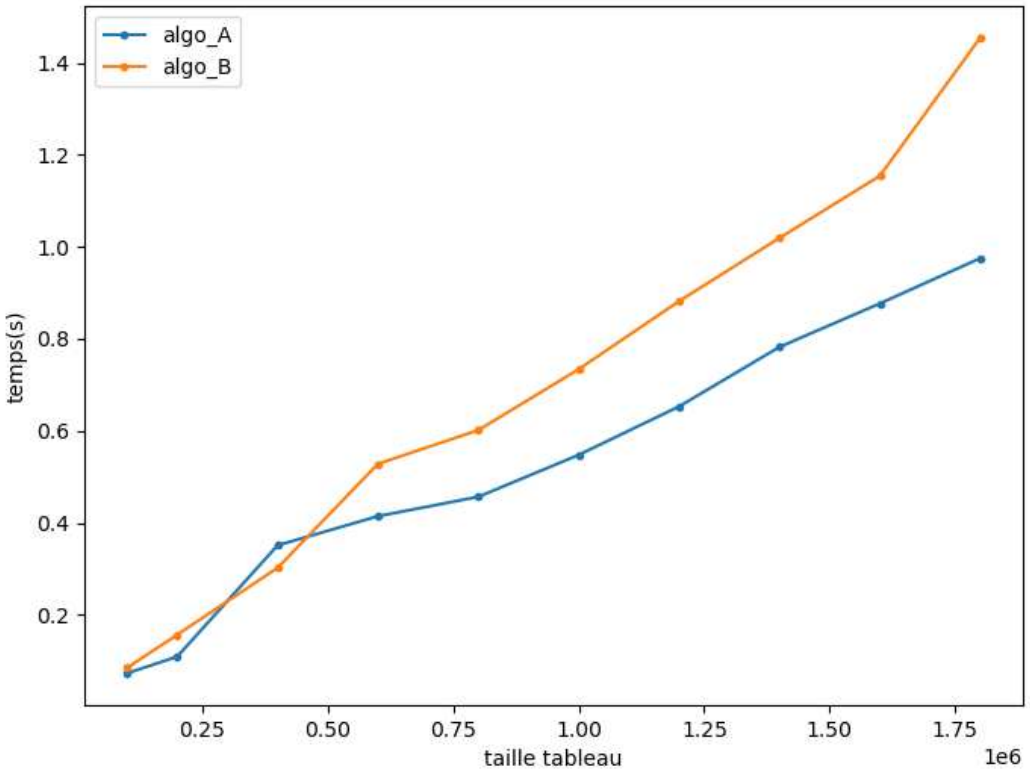
Complexité :

- L'algorithme fait un parcours de $\frac{N}{2}$ pour faire la permutation deux a deux
  donc : $\frac{N}{2}$ test
- Le deuxième parcours est de $\frac{N-2}{2}$ pour trouver le max et le min
  donc : N-2 test
- Deux derniers test sont faite si N est impair pour tester avec la dernier case du tableau
- **Nombre de test** : $\frac{3}{2}\ N - 1$ si N est pair et $\frac{3}{2}\ N + 1$ sinon
- **Complexité : O(N)**

Etude expérimentale et représentation de résultats:

	Algorithm A		Algorithm B	
	temps (s)	nbr_comparaison	temps (s)	nbr_comparaison
100000	0.07288456	199998	0.084832191	149999
200000	0.10932827	399998	0.156754255	299999
400000	0.351104021	799998	0.302281857	599999
600000	0.414398432	1199998	0.528259516	899999
800000	0.456614256	1599998	0.601560354	1199999
1000000	0.547736645	1999998	0.733939886	1499999
1200000	0.652977467	2399998	0.881943703	1799999
1400000	0.782326937	2799998	1.019279718	2099999
1600000	0.876785278	3199998	1.154253244	2399999
1800000	0.975199223	3599998	1.454521894	2699999

Representation graphique de T(N) :



Remarque :

- Le nombre de comparaison de l'algorithme B est moins que l'algorithme A car on a moins d'itérations dans le deuxième algorithme
- L'algorithme B a un temps d'exécution plus grand car il fait plusieurs affectations en effet les instructions d'affectations en python prend du temps car elles nécessitent des accès mémoires et des tests de compatibilité de type et d'autres test caché
- La deuxième approche reste une optimisation car le nombre de tests est réduit et dans d'autre langage, elle peut donner un temps bien moins que la première

END