COMPTE RENDU ALGORITHMIQUE

Aurélien Paradis
Thomas Koenig
Guillaume Finot

Table des matières

Aurélien Paradis
Thomas Koenig
Guillaume Finot

Aurélien Paradis
Thomas Koenig
Guillaume Finot

RECHERCHE

Dans un premier temps, pour notre SAE, nous avons besoin d'un algorithme afin de trier les vols et les passagers. Les heures de décollages sont classées par ordre croissant au départ, cependant il se peut qu'il y ait des changements d'heures ou des reprogrammations, tout en restant occasionnel. Pour le tri des passagers, ils sont mélanges et il est necéssaire de les trier que le programme est lancé. Dans ce contexte, on privilégiera en conséquent un algorithme optimisé, c'est-à-dire pour les tableaux de petites tailles (10 éléments).

I - ALGORITHME DE TRI

1 - Algorithme tri par sélection

A – Algorithme

« « «

Entrée : tab [int]

Sortie : tab[int]

Pré-condition : len(tab) >= 1

Post-condition : Sélection de la valeur minimum du tableau puis on l'ajoute à la première place du tableau puis réitère l'opération sur l'indice d'après

» » »

Complexité théorique : $O(n^2)$

```
def tri_selection(tab):
    compteur = 0
    i = 0
    j=0
    compteur = 2+compteur
    while i < len(tab) - 1:
        min = i
        j=i+1
        compteur=5+compteur
        while j < len(tab):
            compteur+=1
            if tab[j] < tab[min]:
                min = j
                compteur=compteur+2
            j+=1
            compteur=2+compteur

        compteur=1+compteur
        tmp = tab[i]
        tab[i] = tab[min]
        tab[min] = tmp
        i = i + 1
        compteur =5+compteur

    compteur=1+compteur
    return compteur tab
```


Aurélien Paradis
Thomas Koenig
Guillaume Finot

B – Observation

Le tri par sélection nous donne des résultats en constantes augmentations de manière quadratique en termes d'opération élémentaire :

Taille tableau	Nombre moyen opérations élémentaires
10	261.6
500	383 163.6
5000	37 589 478

2 - Algorithme tri par insertion

A – Algorithme

« « «

Entrée : tab [int]

Sortie : tab[int]

Pré-condition : len(tab) >= 1

Post-condition : Trie des éléments successive du tableau, à au i^{ieme} indice on parcourt le tableau pour voir ou l'insertion se fais sur un tableau de i-1 valeur

» » »

Complexité théorique : $O(n^2)$

```
def tri_insert(tab):
    compteur = 0
    n=len(tab)
    i=1
    compteur = compteur + 2
    while i<n:

        k = tab[i]
        j = i - 1
        compteur = compteur + 4
        while j >= 0 and k < tab[j]:

            tab[j+1] = tab[j]
            j=j-1
            compteur = compteur + 6
        compteur = compteur + 1
        tab[j+1] = k
        i = i+1
        compteur = compteur + 4
    compteur = compteur + 1
    return compteur
```

B – Observation

Le tri par sélection nous donne des résultats en constantes augmentations de manière quadratique en termes d'opération élémentaire en étant légèrement meilleur que le tri par selection :

Aurélien Paradis
Thomas Koenig
Guillaume Finot

Taille tableau	Nombre moyen opérations élémentaires
10	218.4
500	373 674.0
5000	37 240 611.6

3 - Algorithme tri à bulles

A – Algorithme

« « «

Entrée : tab [int]

Sortie : tab[int]

Pré-condition : len(tab) >= 1

Post-condition : Le tableau de sortie est une permutation, il prend la valeur de la première occurrence et la permute avec la plus petite valeur du tableau, puis réitération avec les n-1 valeurs

» » »

Complexité théorique : $O(n^2)$

```
def tri_a_bulles(tab):
    compteur = 0
    n=len(tab)-1
    compteur = compteur + 2

    while n>0:
        i=0
        compteur = compteur + 2

        while i<n:
            compteur = compteur + 1
            if tab[i]>tab[i+1]:
                tmp=tab[i]
                tab[i]=tab[i+1]
                tab[i+1]=tmp
                compteur = compteur + 7
            i=i+1
            compteur = compteur + 3
        n=n-1
        compteur = compteur + 3
    compteur = compteur + 1
    return compteur
```

B – Observation

Le tri par sélection nous donne des résultats en constantes augmentations de manière quadratique en termes d'opération élémentaire tout en étant nettment moins meilleur que les 2 algorithme précédents:

Taille tableau	Nombre moyen opérations élémentaires
10	356.8
500	1 055 814.7
5000	94 706 849.8

4 - Algorithme tri fusion

A – Algorithme

« « «

Entrée : tab [int]

Sortie : tab[int]

Pré-condition : len(tab) >= 1

Post-condition : Tableau en sortie est une réunification de tableau coupé a plusieurs reprises pour obtenir des tableaux de 2 éléments triés. Puis réunification des tableaux tout en les triant via comparaison des éléments en partant des 2 premiers éléments des tableaux tout en avançant indépendamment de chacun des deux tableaux

» » »

Complexité théorique : O(n log(n))

```
def tri_fusion(tab):
    compteur = 0
    n = len(tab)
    compteur = compteur + 2
    if n > 1:
        millieu = n // 2

        tab1 = tab[:millieu]
        tab2 = tab[millieu:]
        compteur = compteur + 4

        tab1,compteur_rec = tri_fusion(tab1)
        compteur = compteur + compteur_rec
        tab2, compteur_rec = tri_fusion(tab2)
        compteur = compteur + compteur_rec

        compteur_rec = interclassement(tab1, tab2, tab)
        compteur = compteur + compteur_rec

    return tab,compteur


def interclassement(tab1, tab2, tab3):
    compteur = 0
    i, j, k = 0, 0, 0
    taillei, taillej = len(tab1), len(tab2)

    compteur = compteur + 8

    while i < taillei and j < taillej:
        compteur = compteur + 1
        if tab1[i] <= tab2[j]:
            tab3[k] = tab1[i]
            i = i + 1
            compteur = compteur + 3
        else:
            tab3[k] = tab2[j]
            j = j + 1
```


Aurélien Paradis
Thomas Koenig
Guillaume Finot

```
            compteur = compteur + 3
        k = k + 1
        compteur = compteur + 2

    compteur = compteur + 1
    while i < taillei:
        tab3[k] = tab1[i]
        i = i + 1
        k = k + 1
        compteur = compteur + 5

    compteur = compteur + 1
    while j < taillej:
        tab3[k] = tab2[j]
        j = j + 1
        k = k + 1
        compteur = compteur + 5

    return compteur
```

B – Observation

Le tri a fusion nous donne des résultats en constantes augmentations de manière logarithmique en termes d'opération élémentaire pour des tableaux de tailles variables :

Taille tableau	Nombre moyen opérations élémentaires
10	356.4
500	35 275.2
5000	454 171.2

5 – Conclusion

En somme, nous observons d'une part le tri a fusion qui est performants et pertinents pour les longs tableaux baissant radicalement le compteur des autres algorithme du fait de qu'il soit de O(n log(n)) pour sa complexité théorique. Toutefois, pour des tableaux plus petits ce n'est pas le plus adéquat et n'apparait pas très optimisé. Les algorithmes de complexité théorique de $O(n^2)$ seront plus efficaces pour des tableaux de plus petites tailles. L'algorithme par insertion est le plus optimisé, suivit de près par l'algorithme de tri par sélection d'après nos tests pour des tableaux de petites tailles et donc le plus adapté pour notre projet. Quant au tri à bulle, malgré une complexité de $O(n^2)$, nos tests démontrent une optimisation médiocre, même un compteur supérieur à l'algorithme de tri a fusion sur les tableaux de 10 entiers malgré une complexité théorique de $O(n^2)$.

II - ALGORITHME DE RECHERCHE

1 - Algorithme de recherche dichotomique

A- Algorithme

```
def recherche_dicho(tab:[int],e:int):
    compteur=0

    deb=0
    fin=len(tab)-1
    ind= -1

    compteur = 4 + compteur

    while deb < fin:
```


Aurélien Paradis
Thomas Koenig
Guillaume Finot

```
        millieu = (fin + deb) //2

        compteur = 4 + compteur

        if tab[millieu] < e:

            deb = millieu + 1
            compteur = 3 + compteur

        elif tab[millieu] > e:

            fin = millieu - 1
            compteur = 4 + compteur

        else:

            ind = millieu
            compteur = 3 + compteur
            return ind,compteur

    compteur = 1 + compteur

    return ind,compteur
```

B – Observation

La recherche dichotomique nous donne des résultats en constantes augmentations en termes d'opération élémentaire pour des tableaux de tailles variables :

Taille tableau	Nombre moyen opérations élémentaires
10	21
500	55,1
5000	56,7

2 - Algorithme de recherche dichotomique récursif

A – Algorithme

```
def recherche_dicho_rec(tab,e):
    compteur = 0
    n=len(tab)
    millieu = n//2
    ind = 0

    compteur = compteur + 4

    if n == 0:
        ind = ind - 1

        compteur = compteur + 3
    else:
        compteur = compteur + 1

        if tab[millieu] == e:

            ind = millieu
            compteur = compteur + 2

        elif tab[millieu] > e:

            ind ,compteur_rec = recherche_dicho_rec(tab[:millieu],e)
            compteur = compteur + 3 + compteur_rec

        else:

            ind, compteur_rec = recherche_dicho_rec(tab[millieu+1:],e)
            compteur = compteur + 2 + compteur_rec

            if ind != -1:

                ind = ind + millieu + 1
                compteur = compteur + 4

            compteur = compteur + 1
    return ind,compteur
```

B – Observations

La recherche dichotomique récursive nous donne des résultats en constantes augmentations en termes d'opération élémentaire pour des tableaux de tailles variables, bien qu'ils soient plus haut que la version non-récursive :

Taille tableau	Nombre moyen opérations élémentaires
10	31
500	75,4
5000	80

3 - Algorithme de recherche de première occurrence

A – Algorithme

```
def recherche_prem_occ(tab:[int],e:int):
    compteur = 0
    n=len(tab)
    ind=0
    i=0
    compteur = compteur + 3
    while i < n and tab[ind] != e:

        ind = ind + 1
        i = i + 1
        compteur = compteur + 6

    compteur = compteur + 1
    return compteur
```

B – Observations

La recherche de la première occurrence nous donne des résultats en augmentations rapides en termes d'opération élémentaire pour des tableaux de tailles variables :

Taille tableau	Nombre moyen opérations élémentaires
10	60
500	2969,8
5000	29690,8

4 - Algorithme de recherche du nombre d'occurrence

A – Algorithme

```
def nbocc(tab:[int],e:int):
    compteur = 0
    i=0
    nb=0
    compteur = compteur + 2
    while i < len(tab):
        compteur = compteur + 1
        if e==tab[i]:
            nb = nb + 1
            compteur = compteur + 2
        i=i+1
        compteur = compteur + 3
    return compteur
```

B – Observations

La recherche du nombre d'occurrences d'une valeur nous donne des résultats en augmentations rapides en termes d'opération élémentaire pour des tableaux de tailles variables :

Taille tableau	Nombre moyen opérations élémentaires
10	42
500	2011,2
5000	20098,2

5 - Algorithme de recherche du nombre d'occurrence récursive

A – Algorithme

```
def nbocc_rec(tab:[int],e:int,i:int = 0,occ:int=0,compteur:int = 2):
    compteur = compteur + 1
    if i == len(tab):
        return occ,compteur
    compteur = compteur + 1
    if tab[i] == e:
        occ = occ + 1
        compteur = compteur + 2
    return nbocc_rec(tab,e,i+1,occ,compteur)
```

B – Observations

La recherche du nombre d'occurrences récursive d'une valeur nous donne des résultats en augmentations exponentielles en termes d'opération élémentaire pour des tableaux de tailles variables :

Taille tableau	Nombre moyen opérations élémentaires
10	21
500	55,1
5000	56,7

6 – Conclusion

Pour conclure sur ces algorithmes de recherches, on voit une distinction stricte entre eux. À savoir que la recherche dichotomique qu'elle soit récursive ou non est la recherche la moins demandeuse en ressource quelle que soit la taille du tableau. Cependant, si l'on se réfère uniquement aux valeurs moyennes récupérer lors de nos tests, l'algorithme de recherche dichotomique de base serait le plus intéressant.