Algorithmes de tri

Exo 1

1/1

```
Fonction TAB_est_trié (TAB : tableau [1..N] entier) : booléen
variable
        i : entier
début
    i ← 1
    tantque (i<N et TAB[i] ≤ TAB[i+1]) faire
        i ← i+1
    Fintq
    Si
    alors retourne true
    sinon retourne false
    Finsi
Fin
```

2/1

```
Fonction nb_fois (TAB : tableau (1..n) entier) : entier
variable
        i, j : entier
début
    i ← 1   j ← 0
    tantque i < n faire
        Si TAB[i] > TAB[i+1] alors
            j ← j+1
        Finsi
        i ← i+1
    Fintq
    retourne j
Fin
```

EXO2

Algorithme Tri_insertion

Var : i, j, tmp : entier

TAB : tableau [1..N] : entier

```
début
    pour (i ← 2, i ≤ N, i++) faire
        j ← i-1
        tmp ← T[i]
        tant que (j ≥ 1 et tmp < TAB[j]) faire
            TAB[j] ⟷ TAB[j+1]
            j ← j-1
        fin tant que
    fin pour
fin
```

meilleur cas : c'est lorsque le tableau est trié déjà dans l'ordre croissant

$O(N)$

Dans le cas: on a $N-1$ comparaisons

0 échanges

pire cas : c'est lorsque le tableau est trié dans l'ordre décroissant

$O(N^2)$

Dans le cas: on a $\left[\frac{N(N-1)}{2}\right]$ comparaisons

et $\left[\frac{N(N-1)}{2}\right]$ échanges

Algorithme Tri-sélection

Variables

TAB : tableau [1..N] entier

i, j, index_min : entier

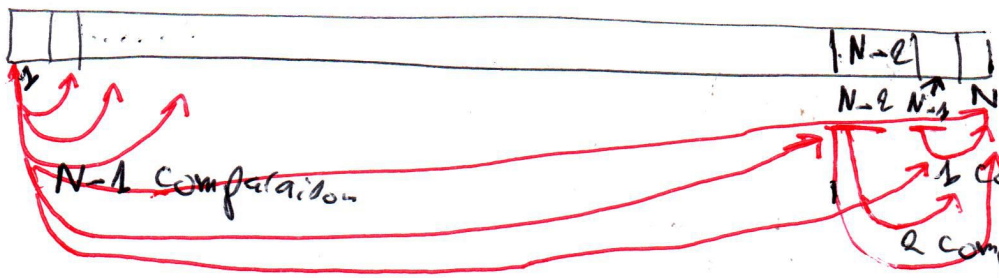
```
début
    pour (i←1, i≤N-1, i++) faire
        index_min ← i
        pour (j←i+1, j≤N, j++) faire
            si TAB[j] < TAB[index_min] alors
                index_min ← j
                //c'est à dire: index_min c'est l'indexe qui peut aller de
            finsi  i à n et qui dont la case contient la valeur
                                                    minimale//
        Finpour
        Si (i ≠ index_min) alors
            TAB[i] <échange> TAB[index_min]
            // c'est à dire: l'échange se fait seulement lorsque
        finsi  la case de l'indexe "i" ne contient
                    pas la valeur minimale //
    Finpour
Fin
```

meilleur cas dans le cas: on a 0 échange et $\frac{N(N-1)}{2}$ comparaisons

pire cas dans ce cas : on a N-1 échange et $\frac{N(N-1)}{2}$ comparaisons

- donc dans tous les cas : la complexité est $O(N^2)$

on a toujours $\frac{N(N-1)}{2}$ comparaisons puisque :



c'est à dire :

l'indexe 1 nécessite $(N-1)$ comparaisons

// 2 // $(N-2)$ comparaisons

⋮ ⋮ ⋮

l'indexe $N-2$ // 2 comparaisons

l'indexe $N-1$ // 1 comparaison

en total : on a besoin $1+2+\dots+(N-2)+(N-1)$ comparaisons

cela revient à la somme d'une suite arithmétique

et qui est égale à $\frac{N(N-1)}{2}$

Algorithme Tri_bulle

Variables

TAB : tableau [1..N] entier

i, passage : entier

échanger : booléen

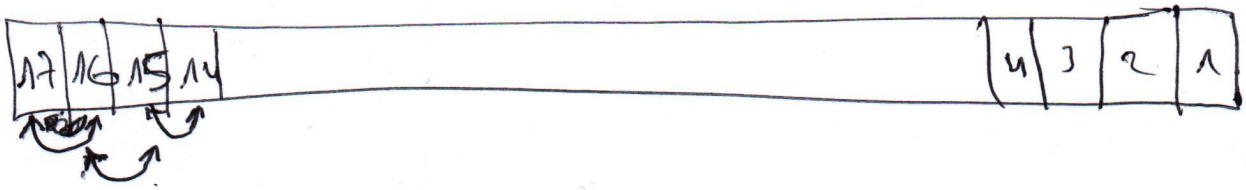
```
début
    passage ← 0
    échanger ← vrai
    tant que (échange = vrai) faire
        échanger ← faux
        pour (i:=1, i≤(N-1-passage), i++) faire
            si TAB[i] > TAB[i+1] alors
                TAB[i] ⟷ TAB[i+1]
                échanger ← vrai
            finsi
        finpour
        passage ← passage + 1
    fin tant que
fin
```

meilleur cas : pas d'échange et (N−1) comparaisons

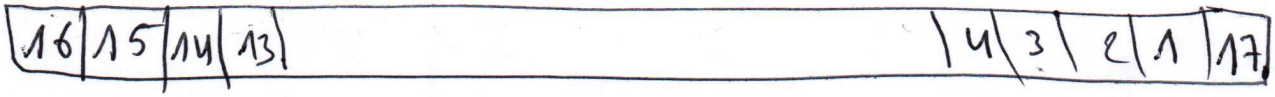
pire cas : dans ce cas : on a un échange par chaque comparaison

en total : on a besoin $\frac{N(N-1)}{2}$ comparaisons et $\frac{N(N-1)}{2}$ échanges.

exemple :

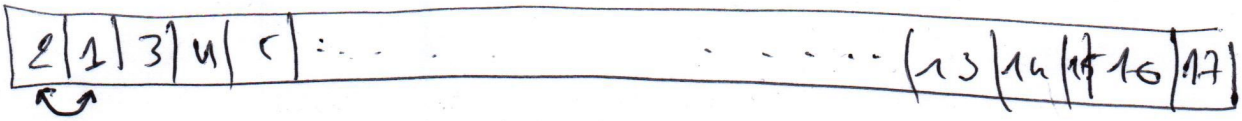


après (N-1) comparaisons et (N-1) échange : on obtient



- Ensuite, on a besoin (N-2) comparaisons et (N-2) échanges pour que la valeur 16 atteint l'avant dernière case
- ensuite, " " (N-3) " " (N-3) " " la valeur 15 " l'avant avant dernière case

et ainsi de suite jusqu'on arrive au dernier échange



qui nécessite une comparaison et un échange

c-à-d : $(N-1)+(N-2)+\dots+1 = \frac{N(N-1)}{2}$ comparaisons

$(N-1)+(N-2)+\dots+1 = \frac{N(N-1)}{2}$ échanges

```
Fonction tri_fusion (T : tableau, Deb : entier, Fin : entier)
    variables : milieu : entier
    Si (Deb < Fin) alors
        milieu ← (Deb + Fin) div 2
        tri_fusion (T, Deb, milieu)
        tri_fusion (T, milieu+1, Fin)
        Fusion (T, Deb, milieu, Fin)
    finsi
fin fonction
```

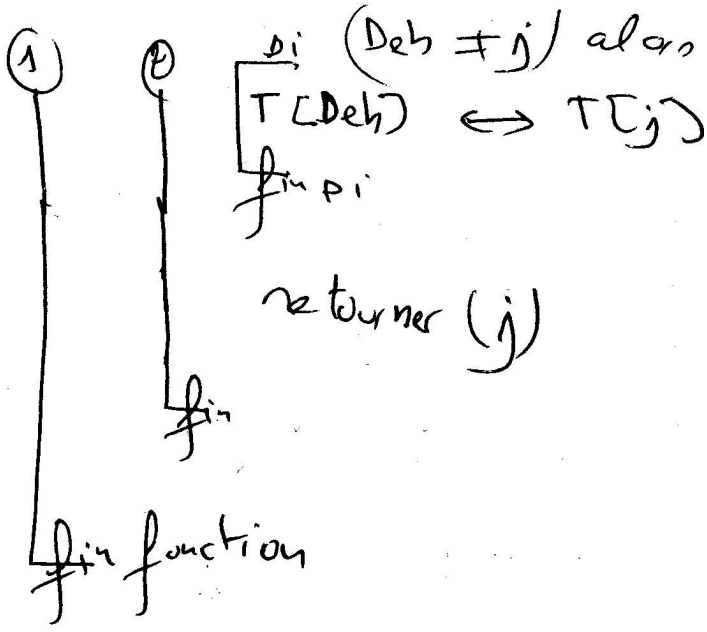
```
Fonction fusion (T : tableau, Deb : entier, milieu : entier, Fin : entier)
    variables : i, j, k : entier
                aux : tableau [1..100]
    i ← Deb , j ← milieu +1     k ← Deb
    tantque (i ≤ milieu) et (j ≤ Fin) faire
        si T[i] < T[j] alors
            aux[k] ← T[i]
            i ← i+1
        else
            aux[k] ← T[j]
            j ← j+1
        finsi
        k ← k+1
    fintantque
    tantque (i ≤ milieu) faire
        aux[k] ← T[i]
        i ← i+1
        k ← k+1
    fintantque
    tantque (j ≤ Fin) faire
        aux[k] ← T[j]
        j ← j+1
        k ← k+1
    fintantque
```

```
K ← Deb
tant que (K ≤ Fin) faire
    T[K] ← aux[K]
    K ← K + 1
fin tant que
fin fonction
```

```
fonction tri_rapide (T: tableau, Deb : entier, Fin : entier)
    var : position : entier
    si (Deb < Fin) alors
        position ← partition (T, Deb, Fin)
        tri_rapide (T, Deb, position - 1)
        tri_rapide (T, position + 1, Fin)
    fin si
fin fonction
```

```
fonction partition (T: tableau, Deb : entier, Fin : entier) : entier
var : i, j, pivot : entier
début
    i ← Deb + 1
    j ← Fin
    Pivot ← T[Deb]
    tant que i ≤ j faire
        tant que (T[i] ≤ Pivot) et (i ≤ Fin) faire
            i ← i + 1
        fin tant que
        tant que ( T[j] > Pivot ) faire
            j ← j - 1
        fin tant que
        si i < j alors
            T[i] ↔ T[j]
            i ← i + 1
            j ← j - 1
        fin si
    fin tant que
```

① ②


1
2
si (Deb ≠ j) alors
T[Deb] ⟺ T[j]
fin si
retourner (j)
fin
fin fonction