CHAPITRE 17

TRI RAPIDE (QUICKSORT)

Ce type de tri est réputé, car il est un des plus efficaces grâce à son principe récursif, et donc implanté comme outil de base dans presque tous les problèmes impliquant le tri des données.

17-1 HISTORIQUE

Cet algorithme a été inventé dans les années 1960 par C.A.R. Hoare et publié dans la revue Computer Journal en 1962[1].Elle a été si efficace que beaucoup essaient encore de trouver une alternative à cet algorithme remarquable.

17-2 LE PRINCIPE DE FONCTIONNEMENT

Le principe de base est de choisir arbitrairement un élément pivot parmi l'ensemble et de balancer les valeurs plus petites d'un coté du pivot et les valeurs plus grandes de l'autre coté. On répète ensuite le même processus sur les 2 sous ensembles. Pour cela, la fonction fait deux appels récursifs sur elle-même, un pour le sous-ensemble de gauche, et un pour le sous-ensemble de droite.

Programmation en langage C:

La programmation exige 2 fonctions: **`Quiksort()`**, elle-même récursive et la fonction **`Partition()`** qui effectue le balancement des valeurs à la droite ou à la gauche du pivot.

[1] "Quicksort"C.A.R. Hoare, Computer Journal, **5**, 1 (1962).

```
//=====================================================================
//  GEN-3405    ALGORITHMES ET STRUCTURES DE DONNÉES    WINDOWS 95/2000
//                                                       VISUAL C++ 6.0
//                              TRI_VITE.CPP
//
//  Programme de démonstration du tri rapide (Quicksort). Codage selon
//  le manuel GEN-3405 de Robert Laganière, pages 138-139. Voir aussi
//  le codage selon Sedgewick dans TRI_RAP.CPP.
//
//  Création: LE René                   Date de création:     96-09-26.
//  Révision: LE René                   Date de modification: 98-09-24.
//=====================================================================
// zone des inclusions de fichiers et des définitions.
//
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
//=====================================================================
// Prototypes de fonctions
//
void Quicksort(int *, int, int);
int  partition(int *, int, int);
void AfficheTableau(int *, int);
//=====================================================================
//
void main()
    {
    FILE *fp;
    int result, i;
    int data[100];

    //construction du tableau par lecture dans un fichier
    fp = fopen("TRI.DAT", "r");
    if (fp == NULL)
        {
        printf("problème d'ouverture du fichier TRI.DAT");
        exit(0);
        }

    // remplir les éléments du tableau.
    i=0;
    while(1)
        {
        result = fscanf(fp, "%d", &data[i]);
        if ( (result == EOF) || (result == 0) ) break;
        i++;
        }
    --i;

    AfficheTableau(data, i);

    // appel à la fonction de tri rapide.
    Quicksort(data, 0, i);

    // affichage du tableau trié.
    AfficheTableau(data, i);
    }
```

```
//=====================================================================
//  Fonction:
//            Procéder au tri par division en deux sous-ensembles et
//            appel récursif sur ces sous-ensembles.
//  Arguments:
//            le pointeur au tableau à trier, l'index du début et
//            l'index de la fin des éléments dans le tableau à trier.
//  Retour:
//            aucun.
//
void Quicksort(int tableau[], int index_gauche, int index_droit)
    {
    int index_separation;

    //évite le travail inutile si moins de 2 éléments à ordonner.
    if (index_gauche < index_droit)
        {
        // c'est cette fonction qui fait l'ordonnance effective.
        index_separation = partition(tableau, index_gauche, index_droit);

        //appels récursifs sur les sous-ensembles gauche et droit.
        Quicksort(tableau, index_gauche, index_separation);
        Quicksort(tableau, index_separation+1, index_droit);
        }
    }
//=====================================================================
//  Fonction:
//            Choisit un élément qui sert de valeur pivot pour classer
//            les éléments en un sous-ensemble des valeurs plus petites
//            que le pivot, et un sous ensemble des valeurs plus grandes
//            que le pivot.
//  Arguments:
//            le pointeur au tableau à classer, l'index du début de zone
//            et l'index de la fin zone dans le tableau.
//  Retour:
//            index de l'élément pivot.
//
int partition(int data[], int inf, int sup)
    {
    int valeur_pivot;
    int tampon;
    int i, j;

    // on choisit un élément arbitaire qui sert de valeur pivot. Ici
    // c'est plus ou moins l'élément au milieu du tableau. Dans le
    // livre de Sedgewick, on utilise plutôt l'élément le plus à droite
    // pour valeur pivot. Cette valeur pivot sert à déterminer dans lequel
    // des deux sous-ensembles gauche ou droite vont aller les autres
    // valeurs.

    // Ici le choix de l'élément pivot est au milieu du tableau.
    valeur_pivot = data[(sup+inf)/2];

    // i et j servent d'index des éléments à permuter.
    i = inf;
    j = sup;

    // effectuer les permutations jusqu'à ce que les index se croisent.
    while (i < j)
        {
        while (valeur_pivot > data[i]) i++;
        while (valeur_pivot < data[j]) --j;
        if (i < j)
            {
            tampon  = data[i];
```

Le programme comporte des commentaires explicites. Toutefois on doit porter une attention particulière au fonctionnement de la fonction **`Partition()`**.

```
            data[i] = data[j];
            data[j] = tampon;
            }
        }
    return j;
    }
//=========================================================================
// Sert à afficher à l'écran le contenu d'un tableau.
//
void AfficheTableau(int data[], int i)
    {
    int k;

    for (k=0; k<=i; k++)
        {
        printf("%d ", data[k]);
        }
    printf("\n");
    }
//=========================================================================
```

2ème codage:

Ici on a implémenté en choisissant l'élément pivot à mi-chemin dans le tableau à trier. Alors que le livre de Sedgewick implémente cette fonction en choisissant l'élément à l'extrémité droite du tableau comme valeur pivot. Voici donc cette version:

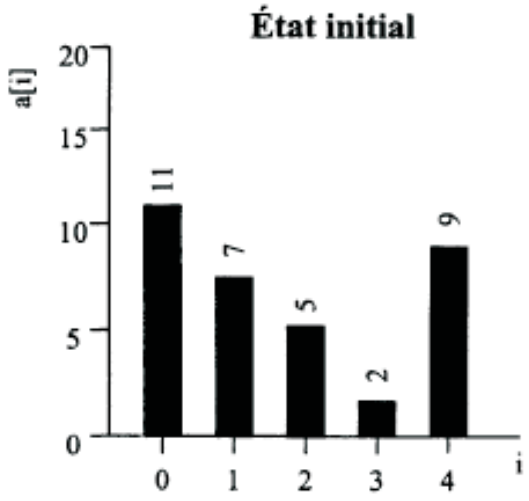
```
//=====================================================================
//  GEN-3405    ALGORITHMES ET STRUCTURES DE DONNÉES    WINDOWS 9x/2000
//                                                       VISUAL C++ 6.0
//                              TRI_RAP.CPP
//
//  Programme de démonstration du tri rapide (Quicksort), mais selon
//  le codage de la page 118 du Sedgewick: "Algorithms in C". Voir
//  aussi l'autre codage dans TRI_VITE.CPP.
//
//  Création: LE René                   Date de création:     96-09-26.
//  Révision: LE René                   Date de modification: 01-10-01.
//=====================================================================
// zone des inclusions de fichiers et des définitions.
//
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
//=====================================================================
// Prototypes de fonctions
//
void Quicksort(int *, int, int);
void AfficheTableau(int *, int);
//=====================================================================
//
void main()
    {
    FILE *fp;
    int result, i;
    int data[100];

    //construction du tableau par lecture dans un fichier
    fp = fopen("TRI.DAT", "r");
    if (fp == NULL)
        {
        printf("problème d'ouverture du fichier TRI.DAT");
        exit(0);
        }

    // remplir les éléments du tableau.
    i=0;
    while(1)
        {
        result = fscanf(fp, "%d", &data[i]);
        if ( (result == EOF) || (result == 0) ) break;
        i++;
        }
    --i;

    AfficheTableau(data, i);

    // appel à la fonction de tri rapide.
    Quicksort(data, 0, i);

    // affichage du tableau trié.
    AfficheTableau(data, i);
    }
```

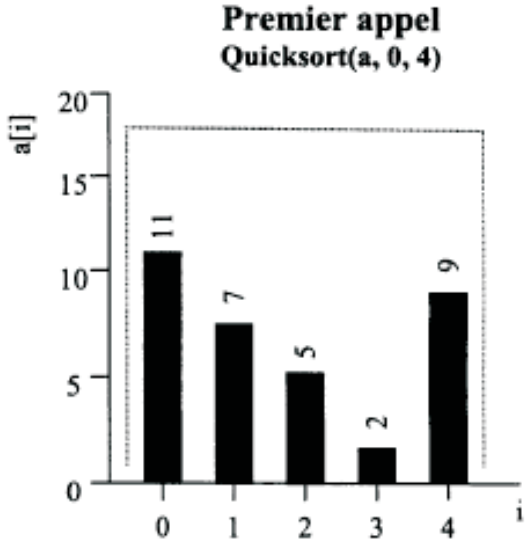
```
//=====================================================================
//  Fonction:
//            Procéder au tri par division en deux sous-ensembles et
//            appel récursif sur ces sous-ensembles.
//  Arguments:
//            le pointeur au tableau à trier, l'index du début et
//            l'index de la fin des éléments dans le tableau à trier.
//  Retour:
//            aucun.
//
void Quicksort(int data[], int index_gauche, int index_droit)
    {
    int valeur_pivot, tampon;
    int i, j;

    //évite le travail inutile si moins de 2 éléments à ordonner.
    if (index_gauche < index_droit)
        {
        // Ici le choix de l'élément pivot est à droite du tableau.
        valeur_pivot = data[index_droit];

        // i et j servent d'index des éléments à permuter.
        i = index_gauche -1;
        j = index_droit;

        // effectuer les permutations jusqu'à ce que les index se croisent.
        for (;;)
            {
            while (data[++i] < valeur_pivot);
            while (data[--j] > valeur_pivot);
            if (i >= j) break;
            tampon = data[i]; data[i] = data[j]; data[j] = tampon;
            }
        tampon = data[i];
        data[i] = data[index_droit];
        data[index_droit] = tampon;

        //appels récursifs sur les sous-ensembles gauche et droit.
        Quicksort(data, index_gauche, i-1);
        Quicksort(data, i+1, index_droit);
        }
    }
//=====================================================================
// Fonction d'affichage d'un tableau.
//
void AfficheTableau(int data[], int i)
    {
    int k;

    for (k=0; k<=i; k++)
        {
        printf("%d ", data[k]);
        }
    printf("\n");
    }
//=====================================================================
```

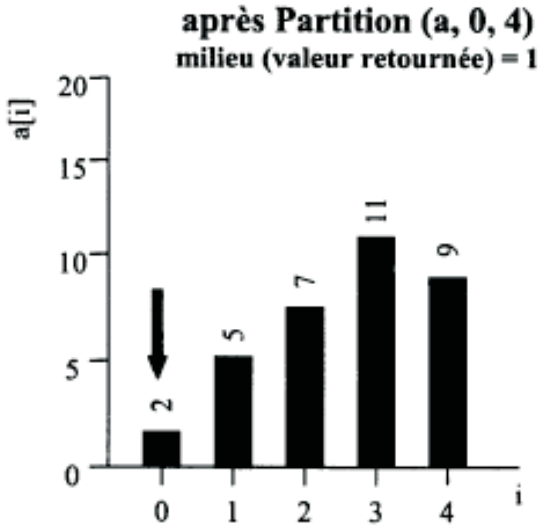
Cela ne change pas grand chose puisque ce n'est pas l'index du pivot mais la valeur de l'élément pivot qui détermine quels éléments permuter pour effectuer un ordonnancement partiel.

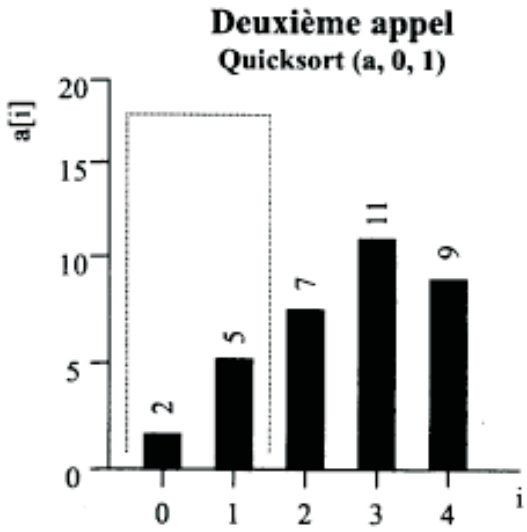
Exemple d'éxécution:

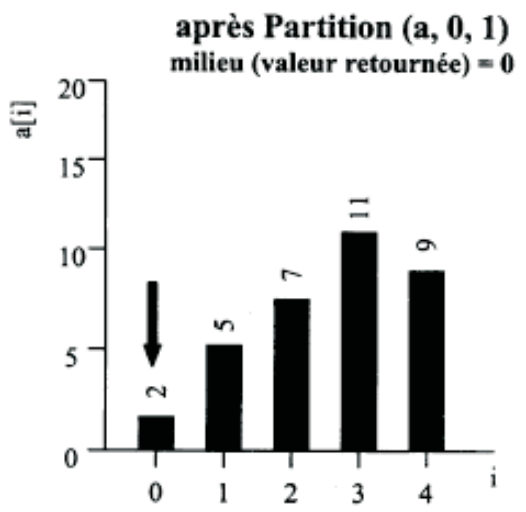
Voici la trace de cette exécution.

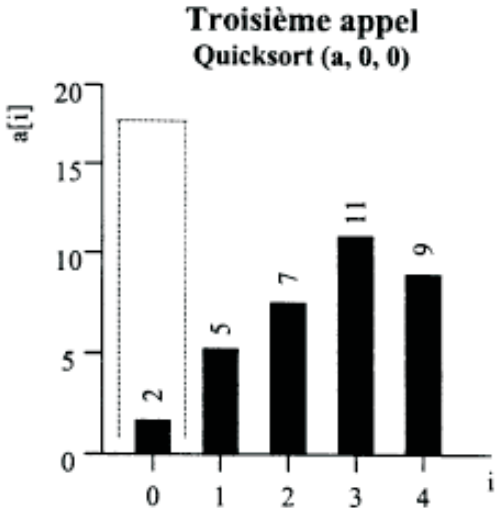


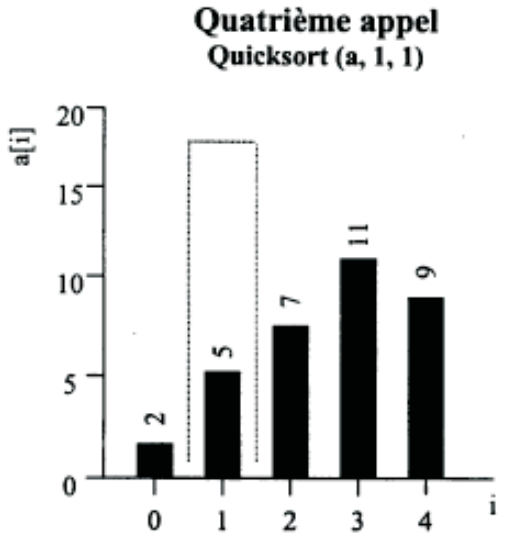


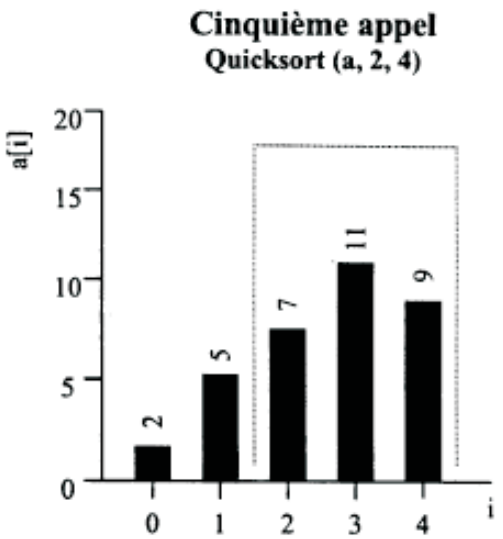


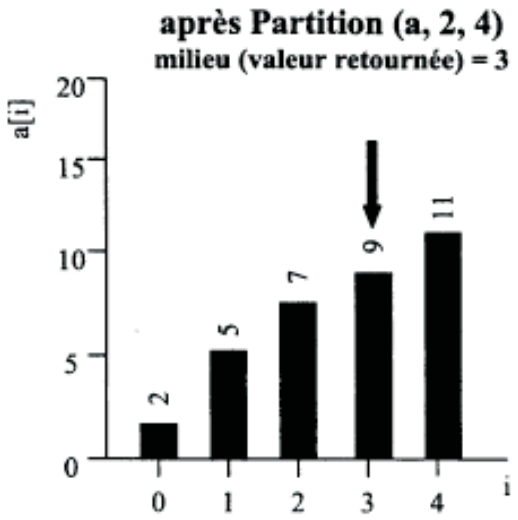


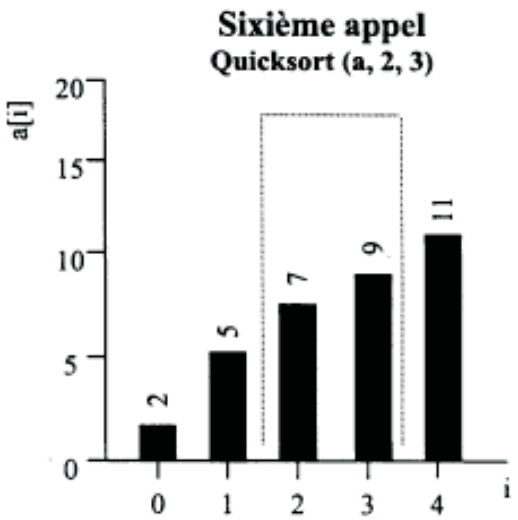

après Partition (a, 0, 1)
milieu (valeur retournée) = 0
a[i]
20
15
10
5
0
2
5
7
11
9
0
1
2
3
4
i

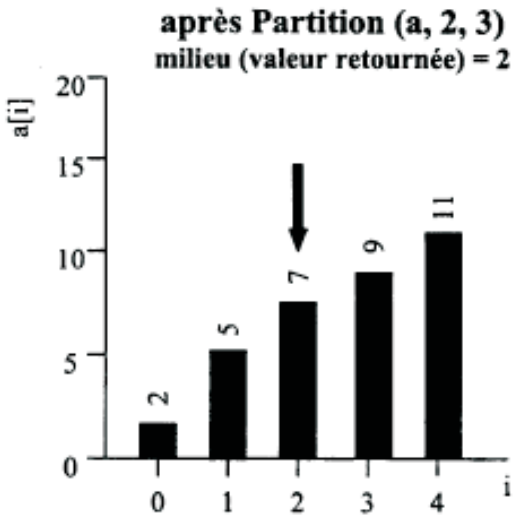

Troisième appel
Quicksort (a, 0, 0)
a[i]
20
15
10
5
0
2
5
7
11
9
0
1
2
3
4
i

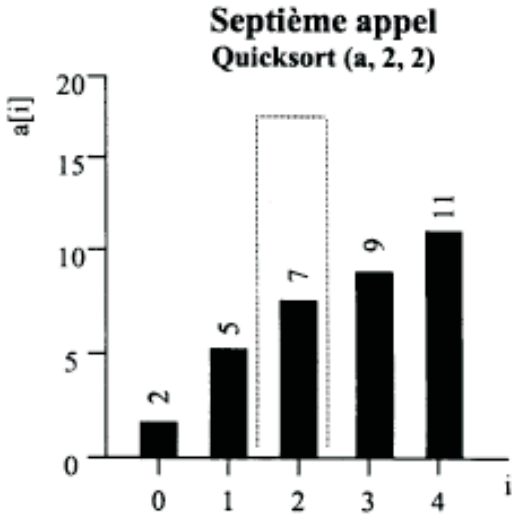

Quatrième appel
Quicksort (a, 1, 1)
a[i]
20
15
10
5
0
2
5
7
11
9
0
1
2
3
4
i

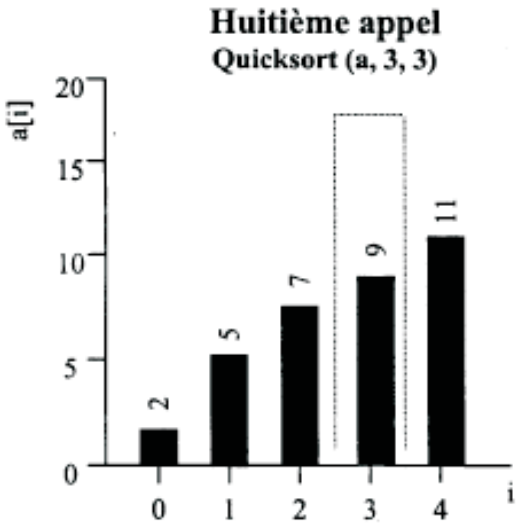

Cinquième appel
Quicksort (a, 2, 4)
a[i]
20
15
10
5
0
2
5
7
11
9
0
1
2
3
4
i

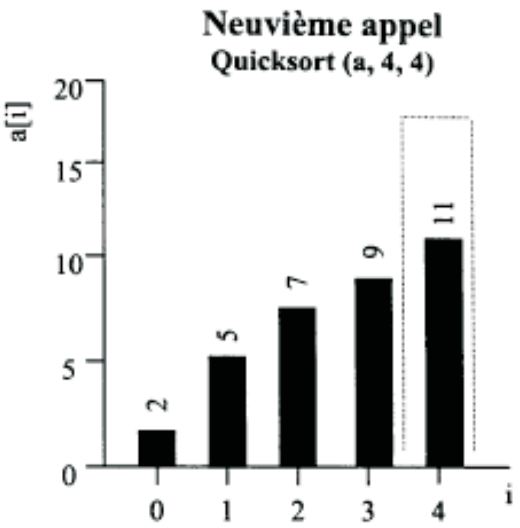

après Partition (a, 2, 4)
milieu (valeur retournée) = 3
a[i]
20
15
10
5
0
2
5
7
9
11
0
1
2
3
4
i

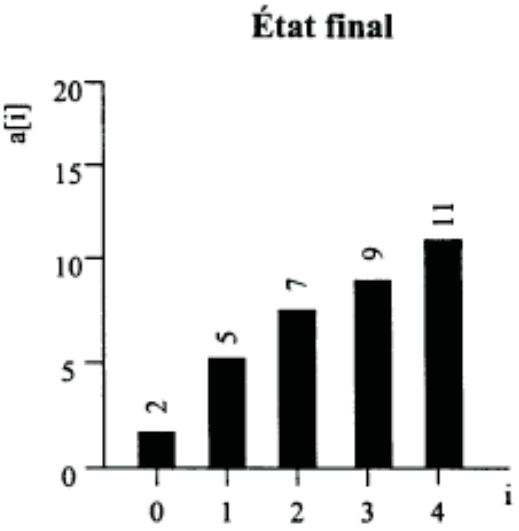

Sixième appel
Quicksort (a, 2, 3)
a[i]
20
15
10
5
0
2
5
7
9
11
0
1
2
3
4
i


après Partition (a, 2, 3)
milieu (valeur retournée) = 2
a[i]
20
15
10
5
0
2
5
7
9
11
0
1
2
3
4
i


Septième appel
Quicksort (a, 2, 2)
a[i]
20
15
10
5
0
2
5
7
9
11
0
1
2
3
4
i


Huitième appel
Quicksort (a, 3, 3)
a[i]
20
15
10
5
0
2
5
7
9
11
0
1
2
3
4
i


Neuvième appel
Quicksort (a, 4, 4)
a[i]
20
15
10
5
0
2
5
7
9
11
0
1
2
3
4
i


État final
a[i]
20
15
10
5
0
2
5
7
9
11
0
1
2
3
4
i

17-3 ANALYSE DE COMPLEXITÉ:

Analyse du meilleur cas:

La meilleure chose qui puisse arriver, c'est qu'à chaque fois que la fonction **`Partition()`** est appelée, elle divise exactement le tableau (ou le sous-tableau) en 2 parties égales. Ce qui rend effectif la philosophie de "diviser pour mieux régner".

À la première passe, les N éléments du tableau sont comparés avec la valeur pivot pour les balancer à la droite ou à la gauche. Il y a donc N comparaisons.

À la seconde passe il y a 2 fonctions **`Partition()`** qui effectuent leur rôle chacune sur leur moitié de tableau. Chaque fonction doit comparer les N/2 éléments du sous-tableau pour effectuer le balancement. Donc de fait, il y a encore N comparaisons pour cette passe.

Il en sera ainsi de même pour la troisième passe, puisqu'il y aura 4 fonctions **`Partition()`** pour comparer chacune N/4 éléments que comporte leur sous-sous-tableau.

En fait, il y aura donc N comparaisons à chaque passe.

Il reste maintenant à déterminer le nombre de passes. L'algorithme s'arrête du moment que les sous-tableaux ne comportent plus qu'un seul élément. Comme à chaque passe, les sous-tableaux sont divisés en deux. Il y a donc K passes tels que 2^K=N. D'où on en déduit que:

$K = \log_2(N)$

Et la complexité est donc le nombre de passes, que multiplie le nombre de comparaissons à chaque passe:

$O(N*\log_2 N)$ = meilleur cas de Quicksort.

Analyse du pire cas:

La pire chose qui puisse arriver, c'est qu'à chaque appel à la fonction **`Partition()`**, à cause des circonstances de la disposition des données, celle-ci place

la totalité du sous-tableau à droite ou à gauche (excluant bien sûr l'élément pivot qui est alors à sa place définitive). Dès lors le tri rapide se transforme en un tri en bulle.

À la première passe, il y aura donc N comparaisons. À la seconde passe il y a déjà une valeur ordonnée et un sous-tableau de N-1 éléments, il y aura donc N-1 comparaisons effectuées par la fonction **`Partition()`**. À la 3ème passe, il y aura N-2 comparaisons, etc.

Pour effectuer le tri au complet, il aura donc fallu en tout N passes. D'où la complexité:

N+(N-1)+...+2+1 = N(N+1)/2

Soit donc $O(N^2)$ = pire cas de Quicksort.

Analyse du cas moyen:

Pour l'analyse du cas moyen il faut faire intervenir les probabilités. Utilisons la formule de récurrence suivante qui indique que le nombre total de comparaisons pour trier N éléments est:

$$C_N = (N+1) + \frac{1}{N}\left(\sum_{k=1}^{k=N} C_{k-1} + C_{N-k}\right) \quad \text{avec:} \quad C_1 = C_0 = 1.$$

Cette relation exprime qu'il y a $N+1$ comparaisons pour effectuer le premier partage (il y a $N-1$ comparaisons entre l'élément pivot avec le reste des éléments, plus 2 comparaisons pour savoir si les indices **`i`** et **`j`** se croisent).

Ensuite, C_{k-1} et C_{N-k} représentent respectivement le nombre de comparaisons pour les sous-tableaux de $k-1$ et de $N-k$ éléments. Comme il y a une chance égale pour que ce cas de figure se présentent sur les N cas de figure possibles, on les a multiplié par le facteur d'**équiprobabilité** $\frac{1}{N}$. Et donc $\frac{1}{N}\left(\sum_{k=1}^{k=N} C_{k-1} + C_{N-k}\right)$ représente la moyenne des comparaisons dans les sous-tableaux.

Comme de fait, C_{k-1} et C_{N-k} sont complémentaires, on arrive à la formule de récurrence suivante:

$$C_N = (N+1) + \frac{2}{N}\left(\sum_{k=1}^{k=N} C_{k-1}\right)$$

Si on effectue les opérations suivantes:

$$N.C_N = N(N+1) + 2\left(\sum_{k=1}^{k=N} C_{k-1}\right) = N(N+1) + 2\left(\sum_{k=1}^{k=N-1} C_{k-1}\right) + 2C_{N-1}$$

$$\text{donc } (N-1).C_{N-1} = (N-1)(N-1+1) + 2\left(\sum_{k=1}^{k=N-1} C_{k-1}\right) = N(N+1) + 2\left(\sum_{k=1}^{k=N-1} C_{k-1}\right) - 2N$$

De sorte qu'en les combinant, on simplifie la formule de récurrence:

$$N.C_N = (N+1)C_{N-1} + 2N$$

Ou encore en divisant les 2 membres par $N(N+1)$:

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} = \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} = \ldots = \frac{C_2}{3} + \sum_{k=3}^{k=N} \frac{2}{k+1}$$

Et la réponse approximée est alors:

$$\frac{C_N}{N+1} \approx \sum_{k=1}^{k=N} \frac{2}{k+1} \approx 2\int_1^N \frac{dx}{x} = 2\ln N$$

Lorsque N devient très grand. D'où le résultat:

$$C_N \approx 2(N+1)\ln N$$

Et alors: O(N*ln(N))= cas moyen.

QUESTIONS ET PROBLÈMES

17.1.