Les tris

I. Introduction

Définition : Un tri, c'est un algorithme qui permet d'organiser une collection d'objets selon un ordre prédéfini. (Croissant, décroissant, alphabétique, etc..)

Ils permettent entre autres d'optimiser les programmes, à travers la manipulation de données triées, et donc d'être en premier lieu user-friendly pour les différents intervenants sur le programme, ainsi qu'aux utilisateurs, en leur proposant des données classées.

II. Quel tri utiliser ?

Le choix d'un tri n'est pas à faire à la légère, en effet, trois notions sont à prendre en compte :

- La complexité en temps : Le premier point à prendre en compte est le temps que le programme mettra à s'exécuter. En effet, le nombre d'étapes de calcul doit être minime, on utilise alors une méthode appelée « l'analyse de la complexité des algorithmes », qui consiste à étudier l'utilisation des ressources nécessaires à l'utilisation d'un algorithme.
- La complexité en espace : Le deuxième point à prendre en compte est l'utilisation des cases mémoires simultanées lors d'un calcul.
- La stabilité : la stabilité est le caractère fiable du tri, compatibilité avec type de données.

III. Les tris

Voici donc les tris les plus utilisés, et leurs fonctions, ainsi que leurs points positifs, négatifs, et dans quelle(s) situation(s) ils s'appliquent.

1. Les résumés

Le tri par sélection

Le principe du tri par sélection/échange (ou tri par extraction) est d'aller chercher le plus petit élément du vecteur pour le mettre en premier, puis de repartir du second élément et d'aller chercher le plus petit élément du vecteur pour le mettre en second, etc...

Le tri par insertion

C'est le tri du joueur de cartes. On fait comme si les éléments à trier étaient donnés un par un, le premier élément constituant, à lui tout seul, une liste triée de longueur 1. On range ensuite le second élément pour constituer une liste triée de longueur 2, puis on range le troisième élément pour avoir une liste triée de longueur 3 et ainsi de suite...

Le principe du tri par insertion est donc d'insérer à la nième itération le nième élément à la bonne place.

Le tri Gnome

L'algorithme est similaire au tri par insertion, mais, au lieu d'insérer directement l'élément à sa bonne place, l'algorithme effectue une série de permutations, comme dans un tri bulle.

Le tri de Shell

Ce tri consiste à trier séparément des sous-suites de la table, formées par les éléments répartis de h en h. Empiriquement, Shell propose la suite d'incréments vérifiant $h_1 = 1$, $h_{n+1} = 3h_{n+1}$ en réalisant les tris du plus grand intervalle possible vers le plus petit.

Le tri fusion

Il s'agit à nouveau d'un tri suivant le paradigme diviser pour régner. Le principe du tri fusion (ou tri par interclassement) en est le suivant :

- On divise en deux moitiés la liste à trier (en prenant par exemple, un élément sur deux pour chacune des listes).
- On trie chacune d'entre elles.
- On fusionne les deux moitiés obtenues pour reconstituer la liste triée.

Le tri rapide

Le tri rapide - aussi appelé "tri de Hoare" ou "tri par segmentation" ou "tri des bijoutiers" ou, en anglais "quicksort" - est certainement l'algorithme de tri interne le plus efficace.

Le principe de ce tri est d'ordonner le vecteur T.(0)..T.(n) en cherchant dans celui-ci une clé pivot autour de laquelle réorganiser ses éléments. Il est souhaitable que le pivot soit aussi proche que possible de la clé relative à l'enregistrement central du vecteur, afin qu'il y ait à peu près autant d'éléments le précédant que le suivant, soit environ la moitié des éléments du tableau. Dans la pratique, comme dans la démo ci-dessous, on prend souvent le dernier élément du tableau.

On permute ceux-ci de façon que pour un indice j particulier tous les éléments dont la clé est inférieure à pivot se trouvent dans T.(0)...T.(j) et tous ceux dont la clé est supérieure se trouvent dans T.(j+1)...T.(n). On place ensuite le pivot à la position j.

On applique ensuite le tri récursivement à, sur la partie dont les éléments sont inférieurs au pivot et sur la partie dont les éléments sont supérieurs au pivot.

Le tri bulle

Le principe du tri à bulles (bubble sort ou sinking sort) est de comparer deux à deux les éléments e_1 et e_2 consécutifs d'un tableau et d'effecteur une permutation si $e_1 > e_2$. On continue de trier jusqu'à ce qu'il n'y ait plus de permutation.

Le tri à peigne

Le tri à peigne est un algorithme de tri par comparaison qui améliore de façon notable les performances du tri à bulle. Le principe est de ne plus comparer uniquement des éléments adjacents, mais de commencer par comparer des éléments plus lointains, puis de raccourcir progressivement l'intervalle entre les éléments pour aboutir finalement à un tri à bulle classique.

TRI DE BATCHER

TRI TOPOLOGIQUE

Le tri shaker

Le tri "Shaker", également appelé tri "Cocktail", tri "Shuttle", tri "boustrophédon" ou tri à bulles bidirectionnel est une variante du tri à bulles.

Son principe est identique à celui du tri à bulles, mais il change de direction à chaque passe. C'est une légère amélioration car il permet non seulement aux plus grands éléments de migrer vers la fin de la série mais également aux plus petits éléments de migrer vers le début.

Tri par impair

C’est un algorithme de tri simple, basé sur le tri à bulles, avec lequel il partage quelques caractéristiques. Il opère en comparant tous les couples d'éléments aux positions paires et impaires consécutives dans une liste et, si un couple est dans le mauvais ordre (le premier élément est supérieur au second), il en échange les deux éléments.

FIFO

La méthode du premier entré, premier sorti, aussi désignée par son acronyme PEPS ou leurs équivalents en anglais First In, First Out ou FIFO est employée en gestion pour désigner une méthode de gestion des stocks. La première référence entrée, est la première à sortir de la pile.

LIFO

Last In, First Out, souvent abrégé par l'acronyme LIFO, signifie « dernier arrivé, premier sorti ». En informatique pour décrire la structure de pile. La dernière donnée enregistrée est ainsi la première à être retirée (sauf commande particulière).

2. Les algorithmes

1. Le tri par sélection

```
PROCEDURE tri_Selection ( Tableau a[1:n])
        POUR i DE 1 A n - 1 FAIRE
            TROUVER [j] LE PLUS PETIT ELEMENT DE [i + 1:n];
            ECHANGER [j] ET [i];
FIN PROCEDURE
```

2. Le tri par insertion

```
PROCEDURE tri_Insertion ( Tableau a[1:n])
        POUR i VARIANT DE 2 A n FAIRE
            INSERER a[i] à sa place dans a[1:i-1];
FIN PROCEDURE
```

3. Le tri Gnome

```
PROCEDURE tri_Gnome(tableau[])
    pos ← 1
    TANT QUE pos < longueur(tableau)
        SI (tableau[pos] >= tableau[pos-1])
            pos ← pos + 1
        SINON
            echanger tableau[pos] ET tableau[pos-1]
            SI (pos > 1)
                pos ← pos - 1
            FIN SI
        FIN SI
    FIN TANT QUE
FIN PROCEDURE
```

4. Le tri de Shell

```
PROCEDURE tri_Insertion ( Tableau a[1:n],gap,debut)
    POUR i VARIANT DE debut A n AVEC UN PAS gap FAIRE
        INSERER a[i] à sa place dans a[1:i-1];
FIN PROCEDURE
```

5. Le tri fusion

```
PROCEDURE tri_fusion ( TABLEAU a[1:n])
FAIRE
    SI TABLEAU EST VIDE RENVOYER TABLEAU
    gauche = partie_gauche de TABLEAU
    droite = partie_droite de TABLEAU
    gauche = tri_fusion gauche
    droite = tri_fusion droite
    renvoyer fusion gauche droite
    POUR i VARIANT DE 1 A n - 1 - passage FAIRE
        SI a[i] > a[i+1] ALORS
            echanger a[i] ET a[i+1]
            permut ← VRAI
        FIN SI
    FIN POUR
    passage ← passage + 1
FIN PROCEDURE
```

6. Le tri rapide

```
PROCEDURE tri_rapide (tableau [1:n], gauche, droit )
   //mur marque la separation entre les elements plus petits
   //et ceux plus grands que pivot
   mur ← gauche;
   // On prend comme pivot l element le plus a droite
   pivot ← tableau[droit];
   placer a gauche de mur tout les elements plus petits
   placer a droite de mur tout les element plus grands
   // On place correctement le pivot
   placer le pivot a la place de mur
   // On poursuit par recursivite
   SI (gauche<mur-1)
   ALORS tri_rapide(tableau,gauche,mur-1);

   SI (mur+1<droit)
   ALORS tri_rapide(tableau,mur,droit);
FIN PROCEDURE
```

7. Le tri bulle

```
PROCEDURE tri_bulle ( TABLEAU a[1:n])
    passage ← 0
    REPETER
        permut ← FAUX
        POUR i VARIANT DE 1 A n - 1 - passage FAIRE
                SI a[i] > a[i+1] ALORS
                    echanger a[i] ET a[i+1]
                    permut ← VRAI
                FIN SI
        FINPOUR
        passage ← passage + 1
    TANT QUE permut = VRAI
FIN PROCEDURE
```

8. Le tri à peigne

```
PROCEDURE tri_peigne ( TABLEAU a[1:n])
    gap ← n
    REPETER
        permut ← FAUX
        gap ← gap / 1.3
        SI gap < 1 ALORS gap ← 1
        POUR i VARIANT DE 1 A n AVEC UN PAS gap FAIRE
            SI a[i] > a[i+gap] ALORS
                echanger a[i] et a[i+gap]
                permut ← VRAI
            FIN SI
        FINPOUR
    TANT QUE permut = VRAI OU gap > 1
FIN PROCEDURE
```

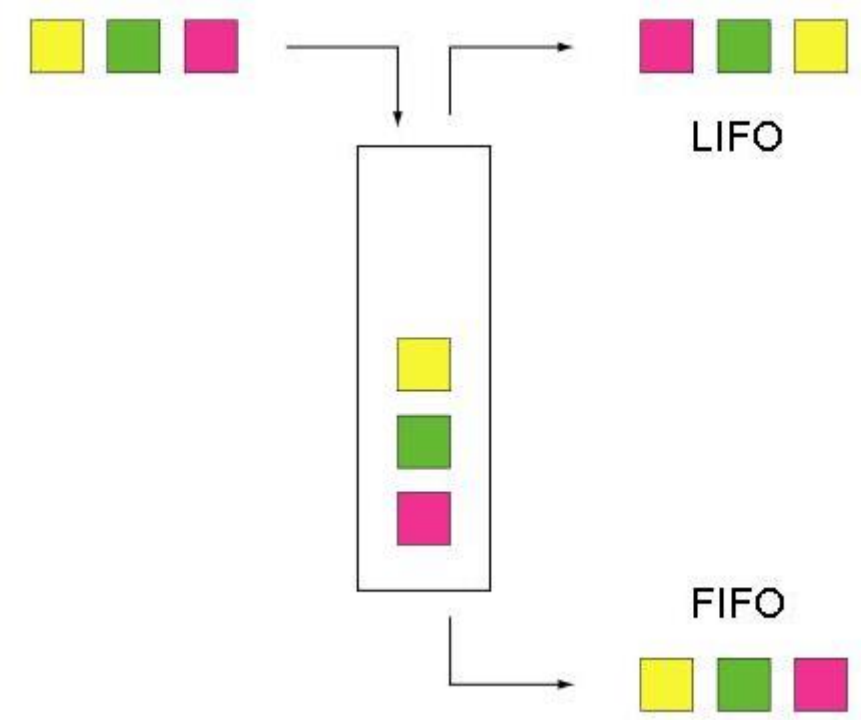
9. Le tri shaker

```
PROCEDURE tri_shaker ( TABLEAU a[1:n])

    sens ← 'avant', debut ← 1, fin ← n-1, en_cours ← 1

    REPETER
       permut ← FAUX
       REPETER
           SI a[en_cours] > a[en_cours + 1] ALORS
               echanger a[en_cours] et a[en_cours + 1]
               permut ← VRAI
           FINSI
           SI (sens='avant') ALORS
               en_cours ← en_cours + 1
           SINON
               en_cours ← en_cours - 1
           FINSI
       TANT QUE ((sens='avant') ET (en_cours<fin)) OU ((sens='arriere') ET (en_cours>debut))
       SI (sens='avant') ALORS
           sens ← 'arriere'
           fin ← fin - 1
       SINON
           sens ← 'avant'
           debut ← debut + 1
       FINSI
    TANT QUE permut = VRAI

FIN PROCEDURE
```

10. Tri pair impair

```
PROCEDURE tri_pair_impair(n)
POUR i:=0 a n-1
      SI i est pair
      ALORS
         POUR j:=0 a n-1 par 2
               compare_echange(a[j],a[j+1]);
         FINPOUR
      SINON
          POUR j:=1 a n-1 par 2
                compare_echange(a[j],a[j+1]);
          FINPOUR
      FINSI
FIN PROCEDURE
```

11. FIFO / LIFO



3. Le code en C

1. Le tri par sélection

```c
void tri_selection(int *tableau, int taille)
{
     int en_cours, plus_petit, j, temp;

     for (en_cours = 0; en_cours < taille - 1; en_cours++)
     {
          plus_petit = en_cours;
          for (j = en_cours 1; j < taille; j++)
               if (tableau[j] < tableau[plus_petit])
                    plus_petit = j;
           temp = tableau[en_cours];
           tableau[en_cours] = tableau[plus_petit];
           tableau[plus_petit] = temp;
     }
}
```

2. Le tri par insertion

```c
void inserer(int element_a_inserer, int tab[], int taille_gauche)
{
    int j;
    for (j = taille_gauche; j > 0 && tab[j-1] > element_a_inserer; j--)
      tab[j] = tab[j-1];
    tab[j] = element_a_inserer;
}
```

3. Le tri Gnome

```
void bulle(int* tableau, int p) {
    int i_b = p;
    while ((i_b>0) && (tableau[i_b]<tableau[i_b - 1])) {
        int t = tableau[i_b -1];
        tableau[i_b - 1]= tableau[i_b];
        tableau[i_b ] = t;
        i_b --;
    }
}

void tri_gnome (int* tableau) {
    for (int i_i=0;i_i<20;i_i++)
    {
        bulle(tableau,i_i);
    }
}
```

4. Le tri de Shell

```
void tri_insertion(int* t, int gap, int debut)
{
    int j,en_cours;
    for (int i = gap + debut; i < 20; i+=gap) {
        en_cours = t[i];
        for (j = i; j >= gap && t[j - gap] > en_cours; j-=gap) {
            t[j] = t[j - gap];
        }
        t[j] = en_cours;
    }
}

void tri_shell(int* t) {
    int intervalles[5]={6,4,3,2,1};
    for (int ngap=0;ngap<5;ngap++) {
        for (int i=0;i<intervalles[ngap];i++)
            tri_insertion(t,intervalles[ngap],i);
    }
}
```

5. Le tri fusion

```c
void fusion (int *a, int n, int m) {
    int i, j, k;
    int *x = malloc(n * sizeof (int));
    for (i = 0, j = m, k = 0; k < n; k++) {
        x[k] = j == n       ? a[i++]
             : i == m       ? a[j++]
             : a[j] < a[i] ? a[j++]
             :               a[i++];
    }
    for (i = 0; i < n; i++) {
        a[i] = x[i];
    }
    free(x);
}

void tri_fusion (int *liste, int taille) {
    if (taille < 2) return;
    int milieu = taille / 2;
    tri_fusion(liste, milieu);
    tri_fusion(liste milieu, taille - milieu);
    fusion(liste, taille, milieu);
}
```

6. Le tri rapide

```c
void tri_rapide (int *tableau, int taille) {
    int mur, courant, pivot, tmp;
    if (taille < 2) return;
    // On prend comme pivot l element le plus a droite
    pivot = tableau[taille - 1];
    mur   = courant = 0;
    while (courant<taille) {
        if (tableau[courant] <= pivot) {
            if (mur != courant) {
                tmp=tableau[courant];
                tableau[courant]=tableau[mur];
                tableau[mur]=tmp;
            }
            mur ++;
        }
        courant ++;
    }
    tri_rapide(tableau, mur - 1);
    tri_rapide(tableau + mur - 1, taille - mur + 1);
}
```

7. Le tri bulle

```
void tri_bulle(int* tableau)
{
    int passage = 0;
    bool permutation = true;
    int en_cours;

    while ( permutation) {
        permutation = false;
        passage ++;
        for (en_cours=0;en_cours<20-passage;en_cours++) {
            if (tableau[en_cours]>tableau[en_cours+1]){
                permutation = true;
                // on echange les deux elements
                int temp = tableau[en_cours];
                tableau[en_cours] = tableau[en_cours+1];
                tableau[en_cours+1] = temp;
            }
        }
    }
}
```

8. Le tri à peigne

```
typedef int bool;
enum { false, true };

void tri_peigne(int* tableau)
{
    int gap = 20;
    bool permutation = true;
    int en_cours;

    while (( permutation) || (gap>1)) {
        permutation = false;
        gap = gap / 1.3;
        if (gap<1) gap=1;
        for (en_cours=0;en_cours<20-gap;en_cours++) {
            if (tableau[en_cours]>tableau[en_cours+gap]){
                permutation = true;
                // on echange les deux elements
                int temp = tableau[en_cours];
                tableau[en_cours] = tableau[en_cours+gap];
                tableau[en_cours+gap] = temp;
            }
        }
    }
}
```

9. Le tri shaker

```
typedef int bool;
enum { false, true };

void tri_shaker(int* tableau) {
    bool permutation;
    int en_cours=0, sens=1;
    int debut=1, fin=19;
    do {
        permutation=false;
        while (((sens==1) && (en_cours<fin)) || ((sens==-1) && (en_cours>debut))) {
            en_cours += sens;
            if (tableau[en_cours]<tableau[en_cours-1]) {
                int temp = tableau[en_cours];
                tableau[en_cours]=tableau[en_cours-1];
                tableau[en_cours-1]=temp;
                permutation=true;
            }
        }
        if (sens==1) fin--; else debut++;
        sens = -sens;
    } while (permutation);
}
```

10. Tri pair impair

```
void tri_pair_impair(int n)
{
    for(int i = 0; i < n-1; i++)
    {
        if(i%2 == 0)
        {
            for (int j = 0; j < n-1; j+2)
            {
                compare_echange(a[j],a[j+1]);
            }
        else
        {
            for (int i = 1; i < n-1; i+2)
            {
                compare_echange(a[j],a[j+1]);
            }
        }
        }
    }
}
```