Algorithmique et structures de données

Algorithmes de tri

Samar MOUCHAWRAB, PhD Eng

2A Cycle Préparatoire – Semestre 1
2017/2018

Algorithmes de tri

Un algorithme de tri est un algorithme qui permet d'organiser une collection d'objets selon une relation d'ordre déterminée.

Les objets à trier sont des éléments d'un ensemble muni d'un ordre total. Il est par exemple fréquent de trier des entiers selon la relation d'ordre usuelle « **est inférieur ou égal à** ».

Les algorithmes de tri sont utilisés dans de très nombreuses situations. Ils sont en particulier utiles à de nombreux algorithmes plus complexes dont certains algorithmes de recherche.

La collection à trier est souvent donnée sous forme de tableau, afin de permettre l'accès direct aux différents éléments de la collection, ou sous forme de liste, ce qui peut se révéler être plus adapté à certains algorithmes et à l'usage de la programmation fonctionnelle.

Algorithmes de tri

Bon nombre d'algorithmes de tri procèdent par comparaisons successives, et peuvent donc être définis indépendamment de l'ensemble auquel appartiennent les éléments et de la relation d'ordre associée.

Un même algorithme peut par exemple être utilisé pour trier des réels selon la relation d'ordre usuelle « est inférieur ou égal à » et des chaînes de caractères selon l'ordre lexicographique.

La classification des algorithmes de tri est très importante, car elle permet de choisir l'algorithme le plus adapté au problème traité, tout en tenant compte des contraintes imposées par celui-ci. Les principales caractéristiques qui permettent de différencier les algorithmes de tri, outre leur principe de fonctionnement, sont la complexité temporelle, la complexité spatiale et le caractère stable.

Complexité algorithmique

Complexité temporelle

La complexité temporelle mesure le nombre d'opérations élémentaires effectuées pour trier une collection d'éléments.

C'est un critère majeur pour comparer les algorithmes de tri, puisque c'est une estimation directe du temps d'exécution de l'algorithme.

Dans le cas des algorithmes de tri par comparaison, la complexité en temps est le plus souvent assimilable au nombre de comparaisons effectuées, la comparaison et l'échange éventuel de deux valeurs s'effectuant en temps constant.

Complexité algorithmique

Complexité spatiale

La complexité spatiale représente, quant à elle, la quantité de mémoire dont va avoir besoin l'algorithme pour s'exécuter.

Celle-ci peut dépendre, comme le temps d'exécution, de la taille de l'entrée.

Il est fréquent que les complexités spatiales en moyenne et dans le pire des cas soient identiques. C'est souvent implicitement le cas lorsqu'une complexité est donnée sans indication supplémentaire.

Caractère stable

Un algorithme est dit stable s'il garde l'ordre relatif des éléments égaux pour la relation d'ordre considérée, c'est-à-dire l'ordre de ces éléments avant l'exécution de l'algorithme.

Caractère stable

Exemple la relation d'ordre ≤ définie sur les couples d'entiers par (a,b) ≤ (c,d) ssi a ≤ c, qui permet de trier deux couples selon leur première valeur.

Soit L = { (4, 1) ; (3, 2) ; (3, 3) ; (5, 4) } une liste de couples d'entiers que l'on souhaite trier selon la relation ≤ préalablement définie.

Puisque (3,2) et (3,3) sont égaux pour la relation ≤, appeler un algorithme de tri avec L en entrée peut mener à deux sorties différentes :

```
L_1 = { (3, 2) ; (3, 3) ; (4, 1) ; (5, 4) }
L_2 = { (3, 3) ; (3, 2) ; (4, 1) ; (5, 4) }
```

L_1 et L_2 sont toutes les deux triées selon ≤, mais seule L_1 conserve l'ordre relatif. Dans L_2, (3,3) apparaît avant (3,2), d'où un algorithme de tri qui aurait pris L en entrée et renvoyé L_2 en sortie serait instable.

Exemples d'algorithmes de tri

Voici une liste de quelques algorithmes de tri:

- Tri par sélection
- Tri par insertion
- Tri à bulles
- Tri de shell
- Tri fusion
- Tri rapide
- Tri par tas
- Introsort
- Tri arborescent
- Tri comptage
- Tri par base
- Tri par paquets

Tri par sélection

Le tri par sélection (ou tri par extraction) est un algorithme de tri par comparaison. Cet algorithme est simple, mais considéré comme inefficace, car il s'exécute en temps quadratique en le nombre d'éléments à trier.

L'idée est simple :

- ✓ rechercher le plus grand élément (ou le plus petit),
- ✓ le placer en fin de tableau (ou en début),
- ✓ recommencer avec le second plus grand (ou le second plus petit),
- ✓ le placer en avant-dernière position (ou en seconde position)
- ✓ et ainsi de suite jusqu'à avoir parcouru la totalité du tableau.

A chaque fois qu'on déplacera un élément en fin de tableau, on sera certain qu'il n'aura plus à être déplacé jusqu'à la fin du tri.

Tri par sélection

Exemple :

Soit le tableau d'entiers suivant :

6 2 8 1 5 3 7 9 4 0

L'élément le plus grand se trouve en 7ème position (si on commence à compter à partir de zéro) :

6 2 8 1 5 3 7 **9** 4 0

On échange l'élément le plus grand (en 7ème position) avec le dernier :

6 2 8 1 5 3 7 **0** 4 **9**

Le dernier élément du tableau est désormais forcément le plus grand. On continue donc en considérant le même tableau, en ignorant son dernier élément. On repère l'élément le plus grand en ignorant le dernier et on l'échange avec l'avant dernier :

6 2 **8** 1 5 3 7 0 4 **9** ➔ 6 2 **4** 1 5 3 7 0 **8** **9**

Tri par sélection

6 2 4 1 5 3 7 0 8 9

6 2 4 1 5 3 7 0 8 9 → 6 2 4 1 5 3 0 7 8 9

6 2 4 1 5 3 0 7 8 9 → 0 2 4 1 5 3 6 7 8 9

0 2 4 1 5 3 6 7 8 9 → 0 2 4 1 3 5 6 7 8 9

0 2 4 1 3 5 6 7 8 9 → 0 2 3 1 4 5 6 7 8 9

0 2 3 1 4 5 6 7 8 9 → 0 2 1 3 4 5 6 7 8 9

0 2 1 3 4 5 6 7 8 9 → 0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9 → 0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

On sait que le tableau est trié lorsque le nombre d'éléments non triés est égal à 1 (condition d'arrêt).

Exercice 5

Tri d’un tableau , ordre ascendant avec un algorithme de tri par sélection.

Solution

```
#include <stdio.h>
void main() {
   int array[100], n, c, d, position, swap;
   //demander nombre d'éléments et remplir le tableau avec données non triées
   printf("Enter number of elements\n");
   scanf("%d", &n);
   printf("Enter %d integers\n", n);
   for ( c = 0 ; c < n ; c++ )
      scanf("%d", &array[c]);
   //on cherche le plus petit élément et on le met au début du
   //tableau. On recommence avec le second plus petit
   for ( c = 0 ; c < ( n - 1 ) ; c++ ) {
      position = c; //on initialise à la position où on va insérer le
                    //plus petit élément
      for ( d = c + 1 ; d < n ; d++ ) {
         if ( array[position] > array[d]) //élément dans d plus petit
            position = d; //on change la valeur de position
      }
      if ( position != c ) { //si on a trouvé un élément plus petit
         swap = array[c];   //on échange pour placer à la bonne position
         array[c] = array[position];
         array[position] = swap;
      }
   }
```

Solution

```
printf("Sorted list in ascending order:\n");
   for ( c = 0 ; c < n ; c++ )
      printf("%d\n", array[c]);
}
```

Solution

Version du programme où on recherche le plus grand élément et on le met à la fin. Puis on recommence avec le second plus grand …
Les changements au programme concernent seulement les compteurs des boucles.

```
#include <stdio.h>
void main() {
   int array[100], n, c, d, position, swap;
   //demander nombre d'éléments et remplir le tableau avec données non triées
   printf("Enter number of elements\n");
   scanf("%d", &n);
   printf("Enter %d integers\n", n);
   for ( c = 0 ; c < n ; c++ )
      scanf("%d", &array[c]);
   //on cherche le plus grand élément et on le met à la fin du
   //tableau. On recommence avec le second grand petit
   for ( c = n-1 ; c > 0 ; c-- ) {
      position = c; //on initialise à la position où on va insérer le
                    //plus grand élément
```

Solution

```
   for ( d = c - 1 ; d >= 0 ; d-- ) {
          if ( array[position] < array[d]) //élément dans d plus grand
             position = d; //on change la valeur de position
       }
       if ( position != c ) { //si on a trouvé un élément plus grand
          swap = array[c];   //on échange pour placer à la bonne position
          array[c] = array[position];
          array[position] = swap;
       }
    }


printf("Sorted list in ascending order:\n");
   for ( c = 0 ; c < n ; c++ )
      printf("%d\n", array[c]);
}
```

Solution

En utilisant une fonction:

```
#include <stdio.h>

void tri_selection (int tab[], int taille) {
        int i, j, swap, position;
        for ( i =taille -1 ; i >0 ; i-- ) {
                position = i;
                for ( j = i - 1; j >= 0; j-- ) {
                        if ( tab[position] < tab[j] )
                                position = j;
                }
                if ( position != i ) {
                        swap = tab[i];
                        tab[i] = tab[position];
                        tab[position] = swap;
                }
        }
}
```

Solution

```
int main(void) {
    int i, n;
    int tableau[20] ;

    printf("Entrer le nombre d'elements\n");
    scanf("%d", &n);
    printf("Entrer %d entiers\n", n);
    for ( i = 0 ; i < n ; i++ )
      scanf("%d", &tableau[i]);

    printf("avant le tri : ");
    for(i = 0; i < n; i++) printf("%d ", tableau[i]);
    printf("\n");

    tri_selection(tableau, n);

    printf("apres le tri : ");
    for(i = 0; i < n; i++) printf("%d ", tableau[i]);
    printf("\n");

    return 0;
}
```

Solution

Une autre méthode en utilisant trois fonctions: recherche de max, échanger, et tri par sélection :

```
#include <stdio.h>
int max(int tab[], int taille){
    int i, position_max=0;
    for (i = 0; i < taille; i++)
        if(tab[i] > tab[position_max])
            position_max = i;
    return position_max;
}
void echanger(int tab[], int i, int j) {
    int tmp;
    tmp = tab[i];
    tab[i] = tab[j];
    tab[j] = tmp;
}
void tri_selection(int tab[], int taille) {
    int position_max, i;
    for(i=0; taille > 1 ; taille--, i--) {//réduire taille pour trier
//sans le plus grand élément déjà trouvé et inséré dans sa position
       position_max = max(tab, taille);
       echanger(tab, taille-1, position_max);
    }
}
```

Solution

```
int main(void) {
    int i, n;
    int tableau[20] ;

    printf("Entrer le nombre d'elements\n");
    scanf("%d", &n);
    printf("Entrer %d entiers\n", n);
    for ( i = 0 ; i < n ; i++ )
      scanf("%d", &tableau[i]);

    printf("avant le tri : ");
    for(i = 0; i < n; i++) printf("%d ", tableau[i]);
    printf("\n");

    tri_selection(tableau, n);

    printf("apres le tri : ");
    for(i = 0; i < n; i++) printf("%d ", tableau[i]);
    printf("\n");

    return 0;
}
```

Tri par insertion

Le tri par insertion est le tri le plus connu. C'est celui que la plupart utilisent intuitivement quand ils doivent trier une liste d'objets, par exemple quand on joue aux cartes.

L'algorithme principal du tri par insertion est un algorithme qui insère un élément dans une liste d'éléments déjà triés (par exemple, par ordre croissant).

Exemple : un joueur de cartes qui dispose de cartes numérotées. Il a des cartes triées de la plus petite à la plus grande dans sa main gauche, et une carte dans la main droite. Où placer cette carte dans la main gauche de façon à ce qu'elle reste triée ? Il faut la placer après les cartes plus petites, et avant les cartes plus grandes.

Tri par insertion

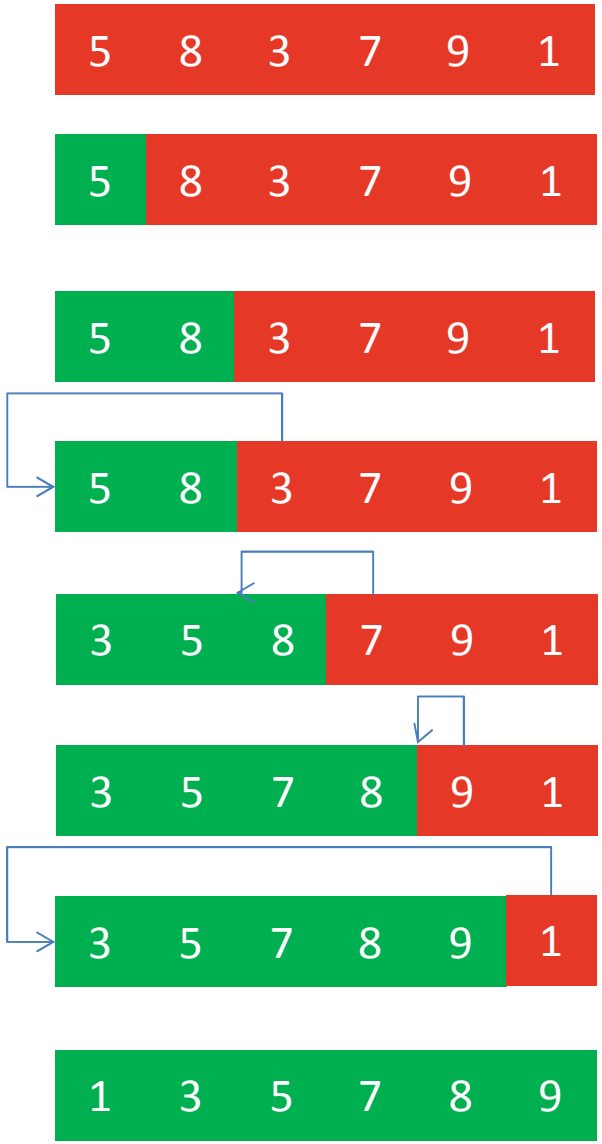
Par exemple, si la main gauche porte (1 3 6 8), et que j'ai la carte 5 dans la main droite, il faut la placer après (1 3) et avant (6 8). Si l'on fait ça, on se retrouve avec la main gauche (1 3 5 6 8), qui est encore triée.

Pour trier entièrement un ensemble de cartes dans le désordre, il suffit alors de placer toutes ses cartes dans la main droite (la main gauche est donc vide), et d'insérer les cartes une à une dans la main gauche, en suivant la procédure ci-dessus.

Au départ, la main gauche est vide, donc elle bien triée. À chaque fois que l'on insère une carte depuis la main droite vers la main gauche, la main gauche reste triée, et la main droite (l'ensemble des cartes non triées) perd une carte. Ainsi, si la main droite comprenait au départ N cartes, en N insertions, on se retrouve au final avec 0 carte dans la main droite, et N cartes, triées, dans la main gauche.

Tri par insertion

Illustration :



Exercice 6

Tri d’un tableau , ordre ascendant avec un algorithme de tri par insertion.

Solution

```
#include <stdio.h>

void tri_insertion(int tab[], int taille)
{
    int i, j; //i pour parcourir main droite, et j main gauche
    for (i = 1; i < taille; ++i) {
        int elem = tab[i];
        for (j = i; j > 0 && tab[j-1] > elem; j--)
            tab[j] = tab[j-1];
        tab[j] = elem;
    }
}
```

- Le premier élément du tableau est dans la main gauche (partie triée par défaut)
- La première boucle parcourt la partie droite, à chaque itération, on prend un élément à insérer dans la partie gauche
- La partie gauche est alors parcourue dans la deuxième boucle, parcourue à l'envers, du plus grand élément au plus petit. On décale vers la droite les éléments, un à un. On boucle jusqu'à trouver un élément dans le tableau <= l'élément à insérer
- On s'arrête et on insère l'élément

Solution

```
int main(void)
{
    int i, n;
    int tableau[20] ;

       printf("Entrer le nombre d'elements\n");
    scanf("%d", &n);
    printf("Entrer %d entiers\n", n);
    for ( i = 0 ; i < n ; i++ )
      scanf("%d", &tableau[i]);

    printf("avant le tri : ");
    for(i = 0; i < n; i++) printf("%d ", tableau[i]);
    printf("\n");

    tri_insertion(tableau, n);

    printf("apres le tri : ");
    for(i = 0; i < n; i++) printf("%d ", tableau[i]);
    printf("\n");

    return 0;
}
```

Tri à bulles

Le tri à bulles (Bubble Sort) ou tri par propagation est un algorithme de tri qui consiste à faire remonter progressivement les plus grands éléments d'un tableau, comme les bulles d'air remontent à la surface d'un liquide.

Le principe du tri à bulles est de comparer deux valeurs adjacentes et d'inverser leur position si elles sont mal placées.

- Si un premier nombre x est plus grand qu'un deuxième nombre y et que l'on souhaite trier l'ensemble par ordre croissant, alors x et y sont mal placés et il faut les inverser.
- Si, au contraire, x est plus petit que y, alors on ne fait rien et l'on compare y à z, l'élément suivant.
- Et on parcourt ainsi la liste jusqu'à ce qu'on ait réalisé n-1 passages (n représentant le nombre de valeurs à trier) ou jusqu'à ce qu'il n'y ait plus rien à inverser lors du dernier passage.

Tri à bulles

Illustration:

Prenons la liste de chiffres « 5 1 4 2 8 » et trions-la de manière croissante en utilisant l'algorithme de tri à bulles. Pour chaque étape, les éléments comparés sont écrits en bleu.

Première étape:

(5 1 4 2 8) → (1 5 4 2 8) Les éléments 5 et 1 sont comparés, et comme 5 > 1, l'algorithme les intervertit.

(1 5 4 2 8) → (1 4 5 2 8) Interversion car 5 > 4.

(1 4 5 2 8) → (1 4 2 5 8) Interversion car 5 > 2.

(1 4 2 5 8) → (1 4 2 5 8) Comme 5 < 8, les éléments ne sont pas échangés.

Tri à bulles

Deuxième étape:

(1 4 2 5 8) → (1 4 2 5 8) pas de changement.
(1 4 2 5 8) → (1 2 4 5 8) Interversion car 4 > 2.
(1 2 4 5 8) → (1 2 4 5 8) pas de changement.
(1 2 4 5 8) → (1 2 4 5 8) pas de changement.
À ce stade, la liste est triée, mais pour le détecter, l'algorithme doit effectuer un dernier parcours.

Troisième étape:

(1 2 4 5 8) → (1 2 4 5 8) pas de changement.
(1 2 4 5 8) → (1 2 4 5 8) pas de changement.
(1 2 4 5 8) → (1 2 4 5 8) pas de changement.
(1 2 4 5 8) → (1 2 4 5 8) pas de changement.
Comme la liste est triée, aucune interversion n'a lieu à cette étape, ce qui provoque l'arrêt de l'algorithme.

Exercice 7

Tri d’un tableau , ordre ascendant avec un algorithme de tri à bulles.

Solution

```
#include<stdio.h>
#include<stdlib.h>

void tab_trier(int tab[], int n) {
    int i, tmp, permutation = 1;
    while (permutation == 1) {
        permutation = 0;
        for (i=0; i<n-1; i++) {
                if (tab[i] > tab[i+1]) {
                        /* Il faut une permutation */
                        tmp = tab[i];
                        tab[i] = tab[i+1];
                        tab[i+1] = tmp;
                        permutation = 1;
                }
        }
    }
}
```

Solution

```
int main() {
    int i, tab[10],n;

    printf("Taille du tableau = ");
    scanf("%d",&n);

    for (i = 0 ; i < n ; i++) {
                printf("Saisir element %d = ",i+1);
                scanf("%d",&tab[i]);
    }
    printf("Tableau avant tri\n");
    for (i = 0 ; i < n; i++) {
                printf("case %d = %d \n",i+1,tab[i]);
    }
    tab_trier(tab ,n);
    printf("\nTableau apres tri\n");
    for (i = 0 ; i < n; i++) {
                printf("case %d = %d \n",i+1,tab[i]);
    }
    return 0;
}
```

Tri rapide

Le tri rapide (en anglais quicksort) est un algorithme de tri inventé par C.A.R. Hoare en 1961 et fondé sur la méthode de conception **diviser pour régner**. Il est généralement utilisé sur des tableaux, mais peut aussi être adapté aux listes.

L'idée, c'est de séparer le tableau en deux. Pour cela, on choisit une valeur du tableau de base, qu'on appelle pivot. Le pivot est souvent la valeur de la première case du tableau. On construit alors deux « sous-tableaux » : l'un contient toutes les valeurs du premier tableau qui sont inférieures ou égales au pivot, l'autre contient les valeurs supérieures au pivot.

Tri rapide

Afin de terminer le tri, il suffira de trier à nouveau chacun des deux sous-tableaux avec quicksort, puis de concaténer les deux tableaux (cela signifie les mettre bout à bout, comme strcat le fait avec les chaînes de caractères en C).

L'algorithme du tri rapide s'utilise lui-même. On dit que c'est un **algorithme récursif**. On arrête l'algorithme quand les sous-tableaux restants sont soit des singletons, soit vides.

Tri rapide

Illustration:

Prenons cette liste : 42 5 38 37 21
Si on choisit 37 comme pivot. En effet, il y a deux valeurs au-dessus (38 et 42) et deux valeurs au-dessous (5 et 21). C'est donc une valeur médiane, ce qui convient très bien pour un pivot (le principal problème de l'implémentation de QuickSort est le choix du pivot).

On prend donc la case 37 comme pivot et on fait passer de l'autre côté du pivot les valeurs qui ne sont pas à leur place :
21 5 37 42 38

Ici, on voit la liste initiale séparée en deux : au milieu le pivot, 37. À gauche, les valeurs inférieures à 37 ont été insérées. De même à droite pour les valeurs supérieures à 37 (>37).

Tri rapide

Bon, maintenant on est sûr que le 37 est à sa position définitive : il ne bougera plus. En effet, toutes les valeurs qui lui sont inférieures sont à sa gauche, et les autres à droite.

Vous pouvez donc recommencer à appliquer le tri rapide pour les deux sous-listes restantes : (21, 5) et (42, 38), avec 21 pivot pour la première liste et 42 pour la deuxième.

Une fois qu'on a fini le tri des deux sous-listes restantes, on obtient donc ceci :

5 21 37 38 42

On remarque que les groupes de nombres entre les pivots sont soit des singletons, soit vides. QuickSort est donc terminé.

Tri rapide

Implémentation :

```
#include <stdio.h>

void quickSort( int[], int, int);
int partition( int[], int, int);

void main() {
        int a[] = { 7, 12, 1, -2, 0, 15, 4, 11, 9};

        int i;
        printf("\n\nUnsorted array is:  ");
        for(i = 0; i < 9; ++i)
                printf(" %d ", a[i]);

        quickSort( a, 0, 8);

        printf("\n\nSorted array is:  ");
        for(i = 0; i < 9; ++i)
                printf(" %d ", a[i]);

}
```

Tri rapide

```
void quickSort( int a[], int l, int r) {
   int j;
   if( l < r )  {
        // diviser pour régner
         j = partition( a, l, r);
       quickSort( a, l, j-1);
       quickSort( a, j+1, r);
   }
}

int partition( int a[], int l, int r) {
   int pivot, i, j, t;
   pivot = a[l];
   i = l; j = r+1;

   while(1) {
        do ++i; while( a[i] <= pivot && i <= r );
        do --j; while( a[j] > pivot );
        if( i >= j ) break;
        t = a[i]; a[i] = a[j]; a[j] = t;
   }
   t = a[l]; a[l] = a[j]; a[j] = t;
   return j;
}
```

Exercice 8

Faites une recherche sur l'algorithme de tri par fusion, son principe de fonctionnement et son implémentation