LU2IN006

Algorithmes de tri

"Structures de données"

Pierre-Henri Wuillemin

2020-2021

1. Algorithmes de tri

Question : Comment trier rapidement une liste d'éléments
Le but de ce cours est de fournir un certain nombre d'algorithmes permettant de trier une liste (d'entiers).

Intérêts ?

Algorithmes régulièrement utilisés

- base de données,
- mails (différents classements),
- systèmes d'exploitation,
- bureautique,
- etc.

hypothèse : dans ce cours, nous trierons dans l'ordre croissant des tableaux d'entiers. Les généralisations à l'ordre décroissant et à des tableaux d'objets complexes ou de pointeurs d'objets complexes sont directes.

Tris

Il existe un grand nombre d'algorithmes de tri, nous allons voir ici les principaux :

- Tri bulle,
- Tri par insertion,
- Tri par sélection,
- Tri fusion,
- Tri rapide.

On supposera toujours qu'il existe quelquepart :

```
1  #define N 100
2  int tab[N];
3
4  void swap(int *t,int i,int j) {
5    int tmp=t[i];
6    t[i]=t[j];
7    t[j]=tmp;
8  }
```

Tri Bulle

Principe

Si une valeur dans le tableau est plus grande que sa voisine de droite, elle serait mieux à sa place.
On répète tant qu'une valeur était mal placée.

Peu à peu, les grandes valeurs migrent vers la fin du tableau.

```
tableau initial : 40 10 50 20 30
iteration 1     : 10 40 20 30 50
iteration 2     : 10 20 30 40 50
```

Analyse

- Durée de l'itération : n
- Nombre d'itération : au plus n

$O(n^2)$

En pratique, un mauvais tri : trop d'opérations (en moyenne : $\frac{n(n-1)}{4}$).

Tri bulle : implémentation

Faire une passe de l'algorithme :

```
1   int bulle_une_passe(int* t) {
2     int res=0,i;
3     for(i=0;i<N-1;i++) {
4       if (t[i]>t[i+1]) {
5         swap(t,i,i+1);
6         res=1;
7       }
8     }
9     return res;
10  }
```

Itérer tant que la dernière passe a servi à quelquechose :

```
1   void tri_bulle(int *t) {
2     while (bulle_une_passe(t)!=0);
3   }
```

Tri par insertion

Principe

À l'itération i, on suppose que les i premiers éléments du tableau sont triés. Alors, déplacer le $i+1$ élément à sa place dans le tableau des $i+1$ triés.

chaque itération peut déplacer les i éléments !

```
tableau initial : 40 (10 50 20 30)
iteration 1     : 10 40 (50 20 30)
iteration 2     : 10 40 50 (20 30)
iteration 3     : 10 20 40 50 (30)
iteration 4     : 10 20 30 40 50
```

Analyse

- Durée de l'itération : au plus i
- Nombre d'itération : $n-1$

$O(n^2)$

En pratique, un mauvais tri : trop d'opérations (en moyenne : $\frac{n(n-1)}{4}$).

Tri par insertion : implémentation

Placer le $i+1$ème élément au bon endroit :

```
1   void insertion_une_passe(int *t,int i) {
2     while (i>0) {
3       if (t[i]<t[i-1]) {
4         swap(t,i,i-1);
5         i--;
6       } else {
7         return;
8       }
9     }
10  }
```

PS : pas l'implémentation la plus rapide.

Itérer $n-1$ fois :

```
1   void tri_insertion(int *t) {
2     int i;
3     for(i=1;i<N;i++)
4       insertion_une_passe(t,i);
5   }
```

Tri par sélection

Principe

À l'itération i, on suppose que les i premiers éléments du tableau sont les plus petits éléments du tableau, triés. Alors, insérer le plus petit élément restant à la position $i+1$.

chaque itération doit rechercher dans $n-i$ éléments !

```
tableau initial : 40 10 50 20 30
iteration 0     : 10 (40 50 20 30)
iteration 1     : 10 20 (50 40 30)
iteration 2     : 10 20 30 (40 50)
iteration 3     : 10 20 30 40 50
```

Analyse

- Durée de l'itération : $n-i$
- Nombre d'itération : $n-1$

$O(n^2)$

En pratique, un mauvais tri : trop d'opérations (dans tous les cas : $\frac{n(n-1)}{2}$).

Tri par sélection : implémentation

Rechercher le $i+1$ème élément parmi les derniers :

```
1  void selection_une_passe(int *t,int i) {
2    int j,min=i;
3    for(j=i+1;j<N;j++) {
4      if (t[j]<t[min]) min=j;
5    }
6    if (min!=i) swap(t,i,min);
7  }
```

PS : pas l'implémentation la plus rapide.

Itérer $n-1$ fois :

```
1  int tri_selection(int *t) {
2    int i;
3    for(i=0;i<N-1;i++)
4      selection_une_passe(t,i);
5  }
```

Tri fusion

Principe

Trier un tableau, c'est trier 2 demi-tableaux puis les interclasser.Et récursivement.

Interclasser 2 tableaux triés est plus facile que trier un tableau...
On utilise le principe connu : "Diviser pour régner".

Algorithme naturellement récursif.

Analyse

Soit $C(n)$ la complexité pire cas du tri fusion de n éléments et $i(n)$ la complexité de l'interclassement.

- $i(n) = n - 1 \approx n$
- $C(n) = i(n) + 2C(\frac{n}{2}) = 1.n + 2C(\frac{n}{2})$
  $= n + 2\left(\frac{n}{2} + 2C(\frac{n}{4})\right) = 2.n + 4C(\frac{n}{4})$
  $\ldots$
  $\approx \log_2(n).n$

$O(n \log_2 n)$

Tri fusion : trace d'exécution


```
40 10 50 20 30
trier[0-4]
  trier[0-2]
    trier[0-1]
      trier[0-0]
      trier[1-1]
      interclasser : 10 40 (50 20 30)
    trier [2-2]
    interclasser : 10 40 50 (20 30)
  trier[3-4]
    trier[3-3]
    trier[4-4]
    interclasser : (10 40 50) 20 30
  interclasser : 10 20 30 40 50
10 20 30 40 50
```

Tri fusion : implémentation d'interclassement

Procédure d'interclassement des tableaux $[p, m]$ et $[m+1, g]$: nécessité d'un tableau temporaire.

```
1  void interclasser(int *t,int p,int m,int g) {
2    int tmp[N];
3    int i,j,k;
4    for(i=p;i<=g;i++) tmp[i]=t[i];
5
6    i=p;j=m+1;k=p;
7
8    for(k=p;k<=g;k++) {
9      if (i==m+1)                  /* lot1 vide      */
10       t[k]=tmp[j++];
11     else if (j==g+1)             /* lot2 vide      */
12       t[k]=tmp[i++];
13     else if (tmp[i]<tmp[j])      /* min dans lot1  */
14       t[k]=tmp[i++];
15     else                         /* min dans lot2  */
16       t[k]=tmp[j++];
17   }
18 }
```

PS : pas l'implémentation la plus rapide.

Tri fusion : implémentation

Procédure récursive : on divise en 2 jusqu'à ce qu'on ne puisse plus.

```
1 void tri_fusion(int *t,int deb,int fin) {
2   if (deb<fin) {
3     int milieu=(deb+fin)/2;
4     tri_fusion(t,deb,milieu);
5     tri_fusion(t,milieu+1,fin);
6     interclasser(t,deb,milieu,fin);
7   }
8 }
```

L'appel se fait donc par `tri_fusion(t,0,N);`

Tri Rapide

Principe

Trier un tableau, c'est juste trier 2 demi-tableaux si toutes les valeurs du premier sous-tableau sont plus petites que toutes les valeurs du second sous-tableau.Et récursivement.

Toujours le principe connu : "Diviser pour régner".
Il n'y pas d'interclassement, donc pas de besoin de tableau auxiliaire.

Algorithme naturellement récursif.

Analyse

Soit $C(n)$ la complexité pire cas du tri rapide de n éléments et $s(n)$ la complexité de la séparation en deux sous-tableaux.

Pour que l'analyse soit la même que celle de tri_fusion, il faut que la procédure de séparation des deux-sous tableau soit linéaire ($s(n) \approx n$) (et maligne). C'est possible (cf. transparent suivant).

$$O(n \log_2 n)$$

Tri rapide : trace

- Séparer le tableau consiste à choisir une valeur qui va servir de pivot :

on sépare en sous-tableau des plus petits et sous-tableau des plus grands que le pivot.

- Comment choisir le pivot ?

On prend l'élément de gauche du tableau (il y a bien plus malin à faire).

```
30 10 50 20 40
trier [0-4] : pivot 2  => 20 10 30 - 50 40
  trier [0-2] : pivot 1  => 10 20 - 30 (50 40)
    trier [0-1] : pivot 0  => 10 - 20 (30 50 40)
    trier [2-2]
  trier  [3-4] : pivot 4  => (10 20 30) 40 - 50
    trier [3-4] : pivot 3 => (10 20 30) 40 50
    trier [4-4]
10 20 30 40 50
```

Tri rapide : implémentation de la séparation

- La valeur pivot est au début du tableau (`t[deb]`).
- On se déplace alors dans le reste du tableau et on échange les éléments pour s'assurer que les élements d'indice $\leq$ `sep` sont tous plus petits que `t[deb]`.
- Il suffit alors de mettre `t[deb]` au milieu.

```
1  int tri_rapide_separation(int *t,int deb,int fin) {
2    int i,sep=deb+1;
3    for(i=deb+1;i<=fin;i++)
4      if (t[i]<t[deb]) {
5        if (i!=sep) swap(t,i,sep);
6        sep++;
7      }
8
9    if (sep!=deb+1) {
10     swap(t,deb,sep-1);
11   }
12   return sep-1;
13 }
```

Tri rapide : implémentation

Le tri rapide est très simple après cela :

```
1 int tri_rapide(int *t,int deb,int fin) {
2   if (deb<fin) {
3     int milieu=tri_rapide_separation(t,deb,fin);
4     tri_rapide(t,deb,milieu);
5     tri_rapide(t,milieu+1,fin);
6   }
7 }
```

Retour sur l'analyse

- La procédure de séparation est cruciale et nécessite d'être linéaire mais aussi de séparer 'correctement' le tableau. Dans le pire de cas, on pourrait avoir une complexité en $O(n^2)$.
- Néanmoins, le tri rapide tend à être en pratique significativement plus rapide que d'autres algorithmes en $O(n \log n)$.

2. Théorèmes du tri

Question : Peut-on produire un algorithme de tri de meilleure complexité que $O(n \log n)$?

Théorèmes du tri

- Un algorithme nécessite au moins $\lceil \log_2 n! \rceil$ comparaisons dans le pire des cas pour trier n éléments.
- Un algorithme nécessite au moins $\lceil \log_2 n! \rceil$ comparaisons en moyenne des cas pour trier n éléments.

Stirling : $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

La complexité d'un algorithme de tri basé sur la comparaison des éléments est au mieux $O(n \log n)$.

Idée de la preuve des théorèmes

- L'opération élémentaire de ces algorithmes est la comparaison de 2 éléments (et leur permutation si besoin) : décision.
- On peut donc toujours tracer un tri comme un chemin dans un arbre des décisions.
- L'arbre des décisions : arbre binaire, dont les feuilles sont des permutations : il y a donc au moins $n!$ feuilles.
- La hauteur de l'arbre est la complexité au pire des cas de l'algorithme.
- Un arbre binaire de hauteur c a au plus 2^c feuilles.

$$n! \leq \text{nombre de feuilles} \leq 2^c \Rightarrow c \geq \log_2(n!) \Rightarrow O(n \log n)$$

3. Tri par dénombrement (Seward, 1954)

On ne doit pas utiliser de comparaison entre éléments. Il faut une autre propriété sur les valeurs à trier.
Supposons : Les valeurs à trier sont n entiers dans $[0, K]$. On peut alors faire un tri en $O(n + K)$.

Idée : Créer un tableau de taille $K + 1$ dans lequel on place à l'indice k le nombre de valeurs k dans le tableau à trier.

```
1  int tri_denombrement(int *t,int max) {
2    int tmp[max+1];
3    int k,i;
4    for(k=0;k<=max;k++) tmp[k]=0;
5    for(i=0;i<N;i++) tmp[t[i]]++;
6
7    i=0;
8    for(k=0;k<max;k++) {
9      while(tmp[k]>0) {
10       t[i++]=k;
11       tmp[k]--;
12     }
13   }
14 }
```