Cours d'algorithmique
Tris particuliers - EISTI - ING 1

Ecole Internationale des Sciences du Traitement de l'Information

Les tris abordés

2. Le tri Shell

4. Le tri pas tas

6. Les tris linéaires

La stabilité des tris

Propriété d'un tri de respecter l'ordre initial des valeurs égales

Taleau initial

Clef	Autre donnée
2	deuxième choix
3	troisième choix
1	premier choix
2	autre deuxième choix

Taleau trié stable

Clef	Autre donnée
1	premier choix
2	deuxième choix
2	autre deuxième choix
3	troisième choix

Tableau trié non stable

Clef	Autre donnée
1	premier choix
2	autre deuxième choix
2	deuxième choix
3	troisième choix

Retour sur le tri par insertion

- A) Le tri par insertion est efficace pour une liste à peu près triée
- B) En moyenne, il est peu efficace car il ne déplace les valeurs que d'une case par instruction

Le tri Shell : Sort Shell

- Son inventeur Donald Shell (1928)
- But : Améliorer le tri par insertion en tenant compte des remarques (A) et (B)
- Principe :

  Soit un tableau u de n cases

  Fixer une valeur entière m

  Pour m de n à 1 pas -1

  Tri insertion du sous tableau v(i) = u(m*(i-1)+1)
- Lors du tri d'un tableau v(i), on déplace des valeurs de plus d'une case par instruction (B)
- Les tris par insertion indicés par m sont de plus en plus efficaces (A).

Le tri Shell

- On ne sait calculer la complexité de ce tri qu'empiriquement.
- Ce tri n'est pas stable.
- Sur des petits tableaux, il est équivalent au tri rapide (quick sort).
- Suivant l'espacement initial n, cette complexité varie de $O(n^2)$ à $O(n\log_2(n))$ en passant par $O(n\log_2^2(n))$.
- Le calcul des espacements optimaux est empirique.
  Le début de la liste : 1, 4, 10, 23, 57, 132, 301, 701.
- Remarque :
  $10/4 \cong 23/10 \cong 57/23 \cong 32/57 \cong 301/132 \cong 701/301 \cong 2.3$

Arbre binaire

- Un tableau est une structure linéaire. Les cases sont placées les unes derrière les autres.
- Nous verrons précisément plus tard d'autres structures de données plus complexes. Nous introduisons l'une d'elle : les arbres binaires.
- Un arbre est une collection d'éléments reliés par des liaisons de type père/fils. Un élément a un et un seul père sauf un élément dit racine de l'arbre.

Arbre binaire

- Un élément qui n'a pas de fils est un nœud externe ou feuille.
- Un élément qui a au moins un fils est un nœud interne.
- Un arbre est un arbre binaire si tous les nœuds internes ont au plus deux fils.
- Un nœud ne peut pas être le père d'un des ces nœuds ancêtres.
- La profondeur d'un nœud est le nombre de liens père/fils entre ce nœud et le nœud racine.

Le tri par tas : Heap Sort

- Principe
  - Le tableau est vu comme un arbre binaire. Les fils de la case u(n) sont u(2n) et u(2n+1).
  - L'algorithme consiste à obtenir un tas. Un tas est un arbre binaire qui vérifie les propriétés suivantes :
    1. Les profondeurs de deux feuilles différent au plus de 1
    2. Les feuilles de plus grande profondeur sont placées à gauche
    3. La valeur d'un nœud est supérieure (resp inférieure) à celle de ces deux fils

Le tri par tas : la fonction tamiser

```
procedure tamiser(ES t tableau() : entier, m :entier, n : entier)
variables
j, k : entier
termine : booleen              // Hyp :les arbres t(2*m) et t(2*m+1) sont des tas.
    k← m                       // On descend t(m) à sa place, sans dépasser l'indice n.
    j← 2 * k                   // Conclusion : l'arbre t(m) est un tas.
    termine ← faux
    tant que j <= n et non termine Faire
        Si j < n et t[j] <  t[j+1] Alors
                j:=j+1
        fin si
        Si t(k) <  t(j) Alors
                swap(t(k), t(j))
                k ←  j
                j ← 2 * k
        sinon
                termine ← vrai
        fin si
    fin tant que
fin procedure
```

Le tri par tas : la fonction triParTas

```
procedure tripartas(ES t tableau() : entier, n : entier)
  // On construit  avec les éléments de t un premier tas
  pour i ← n à 1 pas -1
    tamiser(t, i, n)
  fin pour
  pour i ← n à 2  pas -1
    // t(1) est le plus grand élément de t(1) …. t(i)
    swap(t(i), t(1)) // On l'intervertit avec t(i)
    // On adapte le sous tableau t(1) … t(i-1) pour qu'il soit un
    tas. Seul t(1) est mal placé.
    tamiser(t,1, i -1)
  fin pour
fin procedure
```

Le tri par tas

- Dans la procédure tamiser, l'indice j ne doit pas dépasser n et il est multiplié au moins par deux à chaque itération. L'algorithme est donc en $O(\log_2(n))$
- Dans la procédure tripartas, la procédure tamiser est appelée 2*n – 1 fois. L'algorithme est donc en $O(n * \log_2(n))$
- Ce tri est en place : il ne nécessite aucun espace mémoire autre que le tableau lui-même
- Une url http://formation.enst.fr/SDA/tri_tas.html de l'ENST pour mieux comprendre

Optimalité des tris

- Dans un tableau de taille n, on a n! arrangements possibles des valeurs du tableau.
- Tous les algorithmes vus précédemment sont basés sur des comparaisons entre deux éléments. A chaque comparaison, on divise l'ensemble des arrangements en deux sous ensembles.
- Tous ces algorithmes sont donc au mieux en $\log_2(n!)$
- $(n/2)^{(n/2)} \leq n! \Rightarrow n/2 * \log_2(n/2) \leq \log_2(n!)$
- Les algorithmes de comparaison en $n * \log_2(n)$ sont donc optimaux

Les tris linéaires

- Pour obtenir une complexité en O(n), il faut donc inventer des tris qui ne sont pas basés simplement sur les comparaisons des cases.

- Quand le domaine des valeurs possibles est assez petit, on va pouvoir utiliser des techniques de comptage.

- Le principe général consiste à utiliser des tableaux intermédiaires contenant autant de cases que de valeurs du domaine

Le tri par dénombrement

// Nous supposons que le tableau à trier A contient des valeurs comprises entre 1 et k

```
fonction triDenombrement(E tableau A(N) : entier , E n : entier , E k : entier ) : tableau(N) : entier
variables
tableau C( ), B(N) : entier
i , j : entier
    creertableau(C, k )
    Pour i ← 1 à k pas 1
        C( i ) ← 0
    f in pour
    Pour j ← 1 à n pas 1          // les fréquences d'apparition d'une même valeur
        C(A( j ) )  ← C(A( j ) ) + 1
    fin  pour
    Pour i ←  2 à n pas 1         // les fréquences cumulées d'apparition d'une même valeur
        C( i )  ←  C( i ) + C( i −1)
    f in pour
    Pour j ←   n à 1 pas −1       // placement des valeurs de A dans B en fonction des fréquences
        B(C(A( j ) ) )  ←  A( j )
        C(A( j ) )  ←  C(A( j ) )−1
    fin  pour
    retourner B
fin fonction
```

Le tri par dénombrement : Exemple

A	3	6	4	1	3	4	1	4

La première boucle calcule les fréquences dans C

Ind	1	2	3	4	5	6
Fré	2	0	2	3	0	1

La deuxième boucle cumule les fréquences dans C

Ind	1	2	3	4	5	6
Fré	2	2	4	7	7	8

La troisième boucle place les valeurs de A dans le tableau B en fonction des fréquences cumulées calculées dans C

B								
C	2	2	4	7	7	8		

B							**4**	
C	2	2	4	**6**	7	8		

B		**1**					4	
C	**1**	2	4	6	7	8		

B		1				**4**	4	
C	2	2	4	**5**	7	8		

B		1		**3**		4	4	
C	2	2	**3**	5	7	8		

Le tri par dénombrement : Résultats

- 1ere boucle : O(k)
- 2eme boucle : O(n)
- 3eme boucle : O(n)
- Au total : O(k + n)
- En pratique, on utilise ce tri quand k = O(n)
- On obtient alors un tri en temps linéaire :O(n)

- Ce tri est stable.
- Ce tri n'est pas en place.

Le tri par base

```
// A est un tableau de nombres à d chiffres.
// On tri A par dénombrement chiffre par chiffre en
// commençant par les plus faibles puissances
procedure TriBase (ES tableau A(N) : entier , E n :
entier , E d : entier )
variables
i : entier
    pour i  ← 1 a d pas 1
        A ← triDenombrement(A, n , i )
    fin pour
fin procedure
```

Le tri par base : Exemple

T	unité	dizaine	centaine
3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Le tri par base : Complexité

Complexité

- Soient n nombres de d chiffres dans lequel chaque chiffre peut prendre k valeurs possibles. La complexité du tri par base est alors de

  $O(d * (n + k))$

- Soient n nombres de b bits et un entier positif $r < b$. La complexité du tri par base est alors de

  $O((b/r) * (n + 2^r))$