Structures de Données
Algorithmes de tri

Licence IUP-MIAGE

Pascal PETIT

(sur la base d'un support de **Guillaume HUTZLER (2003-2004)**)

Méthodes de tri

- Un besoin classique (sortie écran, imprimante, ...), tri de fichiers, ...
- Utile pour certains algorithmes (recherche séquentielle versus recherche dichotomique)
- De nombreuses solutions sont connues
- Tri par comparaison: algorithme procédant uniquement par comparaison entre les éléments à trier
- On a un résultat d'optimalité: borne inf pour la complexité au pire en nb de comparaison

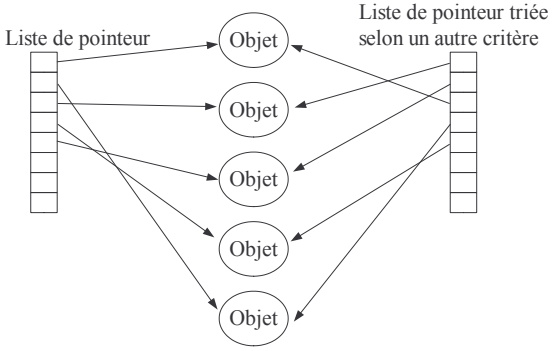
•Méthodes de tri

- Tri interne : en mémoire
- Tri externe: les données à trier ne tiennent pas en mémoire
- Tri sur place: on trie directement la structure de données en la modifiant
  - Aucune structure de données auxiliaire de taille proportionnelle à la taille des données n'est utilisée
- Tri avec recopie: on fournit une nouvelle structure de données, version triée de celle passée en argument (en général plus facile que « sur place »)

•Méthodes de tri: clef

- Chaque élément, éventuellement complexe, a une clef
- Les clefs sont comparables
- Des éléments différents peuvent éventuellement avoir la même clef
- Tri stable: deux éléments ayant la même clef restent dans leur ordre relatif initial
- Exemple d'utilisation: obtenir un tri par nom puis prénom en faisant d'abord un tri par prénom puis un tri stable par nom.

•Organisation de la mémoire



•Organisation de la mémoire

- Tri d'un tableau de pointeurs pointant vers les éléments
  - Possibilité d'avoir plusieurs version triées selon des critères différents
  - Transfert des pointeurs (peu coûteux) et pas des éléments complexes (coûteux)
- Dans la suite du cours, on travaillera avec des listes d'entiers sans perte de généralité

Méthodes de tri

Spécification (1)

- tri = construction d'une nouvelle liste
  - vérifiant la propriété d'être triée
  - contenant exactement les mêmes éléments que la liste initiale (c'est-à-dire obtenue par permutation des éléments)

```
opérations
  tri : liste → liste
axiomes
  ∀ l :liste
  est-triée(tri(l)) ∧ est-permut(l, tri(l))
```

Méthodes de tri

Spécification (2)

- Etant donnée une relation d'ordre, une liste triée répond à la spécification suivante :

```
extension type liste
avec
  _ ≤ _ : élément x élément → booléen
opérations
  est-triée : liste → booléen
sémantique
  est-triée(l) =
    si l = listevide alors vrai
    sinon soit l = cons(e, l') ;
      e ≤ premier(l') ∧ est-triée(l')
    fsi
```

Méthodes de tri

Spécification (3)

- Une permutation d'une liste peut être définie de la manière suivante :

```
extension type liste
opérations
  est-permut : liste x liste → booléen
propriétés
  est-permut(l, l') =
    // si l'une est vide, l'autre aussi
    si estvide(l) alors estvide(l')
    // le premier élément de l doit appartenir à l'
    sinon si (premier(l) ∈ l')=faux  alors faux
    // on supprime premier(l) des deux et
    // on vérifie que les listes obtenues sont
    // des permut l'une de l'autre
    sinon soit l'' = supprimer(premier(l), l')
      est-permut (fin(l), l'')
    fsi
```

Méthodes de tri

Tris simples par sélection

- Principe
  - Recherche du plus petit élément de la liste
  - Déplacement de cet élément en début de liste
  - Tri du reste de la liste
- Famille d'algos en fonction de la manière dont sont effectuées les opérations de
  - sélection
  - déplacement

Méthodes de tri – tris simples par sélection

Exemple d'implantation

```
int placeMinimum(tableau  t, int debut){
  int cb, res=debut,valMin;
  assert(debut >=0);
  assert(debut < t.taille);
  valMin=t.tab[res];
  for(cb=debut+1; cb < t.taille; cb++)
    if (t.tab[cb] < valMin){
      valMin=t.tab[cb];
      res=cb;
    }
  return res;
}
void triSelection(tableau * pt){
  int cb1, pM;
  for(cb1=0; cb1 < pt->taille-1; cb1++){
    // on cherche le plus petit élément d'indice >= cb1
    pM=placeMinimum(*pt, cb1);
    // on échange les éléments d'indice pM et cb1
    echangeElemTab(pt,pM, cb1);
  }
}
```

Méthodes de tri – tris simples par sélection

Exemple

Rang de tête	Rang + petit	liste
1	4	[] 55, 40, 15, 30, 10
2	2	[10] 40, 15, 30, 55
3	3	[10, 15] 40, 30, 55
4	3	[10, 15, 30] 40, 55
5	Fin	[10, 15, 30, 40] 55

Méthodes de tri – tris simples par sélection

Complexité

- 2 opérations fondamentales :
  - la comparaison de deux éléments
    - Recherche dans une liste de longueur p : p-1
    - Au total : $\sum_{p=2}^{n}(p-1)=\sum_{p=1}^{n-1}p=\frac{n*(n-1)}{2}$
    - Complexité en $\Theta(n^2)$
  - le transfert d'un élément d'un emplacement à un autre
    - n - 1 appels à `Echanger` * 3 transferts = 3 * (n-1)
    - Complexité en $\Theta(n)$

Méthodes de tri

Tris simples par insertion

- Principe
  - Insertion des éléments de la liste les un après les autres dans une liste initialement vide
  - Utilisable avec toute structure de données pour laquelle on a une fonction d'insertion
  - Écriture générique de l'algorithme :

```
liste res=listeVide();
for j parcourant l
  ajouter(j,res)
return res;
```

- Famille d'algos en fonction de la manière dont l'élément est placé

Méthodes de tri - Tris simples par insertion
algorithme général sur les tableaux

```
void ajouter(int e, tableau * pt){
  int p, cb;
  assert(pt-> taille < pt->tailleMax);
  p=place(e, *pt);
  if ((p == pt->taille) || (pt->tab[p] != e)){
    for (cb = pt->taille; cb > p; cb--)
      pt->tab[cb]=pt->tab[cb-1];
    pt->tab[p]=e;
    pt->taille++;
  }
}
void triInsertion(tableau * pt){
  int taille=pt->taille, cb;
  pt->taille=1;
  for (cb=1; cb < taille; cb++){
    ajouter(pt->tab[cb], pt);
  }
}
```

Méthodes de tri - Tris simples par insertion

Insertion séquentielle

- L'algorithme de la fonction `place` détermine le type de recherche
- Parcours séquentiel de la liste triée jusqu'à trouver la bonne place

```
//  Nom :        place
//  retourne :   la place que devrait avoir e
//               dans le tableau
int place(int e, tableau t)
{
    int cb=0;
    while ((cb < t.taille) && (t.tab[cb] <= e))
      cb ++;
    return cb;
}
// ce tri est stable grace aux <= de
// (t.tab[cb] <= e)
```

Méthodes de tri – Insertion séquentielle

Exemple

Rang d'él'	Où le placer	liste
		[55] 40, 15, 30, 10
1	0	[40, 55] 15, 30, 10
2	0	[15, 40, 55] 30, 10
3	1	[15, 30, 40, 55] 10
4	0	[10, 15, 30, 40, 55]

Méthodes de tri – Insertion séquentielle

Complexité - Comparaisons

- Représentation contiguë ou chaînée
  - au mieux (liste déjà triée en ordre décroissant)
    - n-1
    - complexité au mieux en O(n)
  - au pire (chaque élément est placé à la fin) :
    - k-1 comparaisons pour placer le $k^{ième}$
    - Au total : $\sum_{k=2}^{n}(k-1)=\sum_{k=1}^{n-1}k=\frac{n*(n-1)}{2}$
    - complexité en $\Theta(n^2)$
  - en moyenne :
    - complexité en $\Theta(n^2)$

Complexité - Transferts

- Représentation contigüe
  - au mieux (liste déjà triée) :
    - 1 (inutile d'ailleurs puisque l'élément est à sa place)
    - complexité au mieux en O(n)
  - au pire (chaque élément est placé en tête) :
    - k-1 affectations pour décaler les éléments
    - Plus une pour placer le $k^{ième}$ élément
    - Au total : $\sum_{k=2}^{n} k = \sum_{k=1}^{n} k - 1 = \frac{n(n+1)}{2} - 1$
    - complexité en $\theta(n^2)$
  - en moyenne :
    - complexité en $\theta(n^2)$
- Représentation chaînée: pas de décalages: complexité en $\theta(n)$

Bilan

- en moyenne moins bon que le tri par sélection ordinaire : $\theta(n^2)$ contre $\theta(n)$ en transferts
  - plus performant si liste presque triée
  - Les deux restent en $\theta(n^2)$ en nombre de comparaisons

Insertion dichotomique

- recherche dichotomique sur la liste en cours de construction pour trouver la bonne place
- impose une représentation contiguë de la liste
  - n'évite pas le décalage des éléments supérieurs pour insérer le nouvel élément

Exemple d'implantation

```
int place2(int e, tableau t, int d, int f)
{
  int milieu, valMilieu;
  while (f-d >= 0){
    milieu=(f+d)/2;
    valMilieu=t.tab[milieu];
    if (e == valMilieu)
      return milieu;
    else if (e < valMilieu)
      f=milieu-1;
    else d=milieu+1;
  }
  return d;
}

int place(int e, tableau t)
{
  return place2(e, t, 0, t.taille-1);
}
```


Complexité - Comparaisons

- au pire
  - recherche du kième demande $\lceil \log_2(k-1) \rceil$
  - au total : $n - 1 + \sum_{k=2}^{n-1} \lceil \log_2(k-1) \rceil$
  - complexité en $\theta(n.\log_2 n)$
- complexité en moyenne identique à la complexité dans le pire des cas

Complexité - Transferts

- comme pour le tri par insertion séquentielle
  - complexité au mieux en O(n)
  - complexités en moyenne et au pire en $O(n^2)$

Méthodes de tri – Insertion dichotomique

Bilan

- excellent pour le nb de comparaisons
  - ($O(n.\log_2 n)$ contre $O(n^2)$ pour les autres)
- pénalisé par le nb de transferts
  - $O(n^2)$
  - inévitable (représentation contiguë)

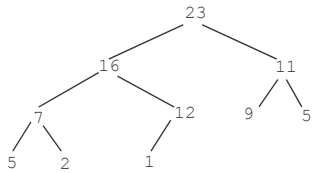
Méthodes de tri

Tri par tas (heapsort)

- S'appuie sur une structure de données adaptée appelée tas
- La SD tas peut aussi servir à implanter efficacement des files de priorité (cf TD)
- Tas:
  - un arbre binaire parfait tel que tout élément soit plus grand que tous ses descendants
  - liste contiguë = représentation par niveau d'un arbre binaire parfait dans un tableau

Méthodes de tri – tri par tas
Exemple de tas

- [23;16;11;7;12;9;5;5;2;1]



Méthodes de tri – tri par tas

Rappels

```
// différent de la formule du cours car on utilise le
   tableau depuis l'indice 0
// au lieu de 1 dans le cours
int pere(int n){
  return (n+1)/2-1;
}
int filsg(int n){
  return 2*(n+1)-1;
}
int filsd(int n){
  return 2*(n+1);
}
// détermine si un entier peut être l'indice d'un noeud
int estNoeudTas(int n, tableau t){
  return n < t.taille;
}
```

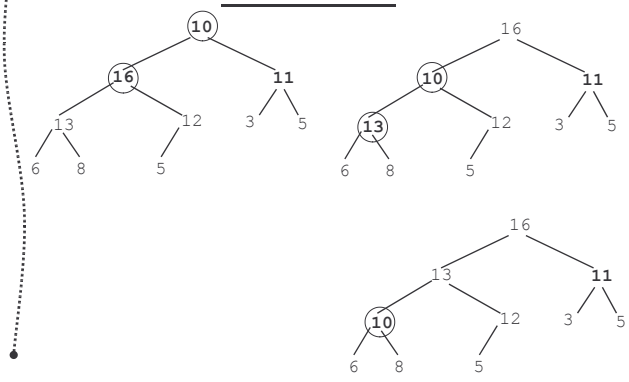
Méthodes de tri – tri par tas

Ordonner

- Préconditions:
  - Les deux sous-arbres de la racine sont des tas
  - La racine ne vérifie pas forcément la propriété des tas
- Action: transforme l'arbre en tas
- Principe de l'algorithme
  - On compare la racine à ses deux fils
  - On l'échange avec le plus grand des deux si nécesaire
  - On applique ordonner sur le sous-arbre où a eu lieu l'échange (qui n'est plus forcément un tas)

Méthodes de tri – tri par tas

Ordonner: exemple



Ordonner (3)

```
// préconditions: sag et sad sont des tas
// transforme l'arbre en tas en partant de noeud
void ordonner (tableau * pt, int noeud){
  int m=noeud;
  if (estNoeudTas(filsg(noeud), *pt)){
    m=indiceDuMax(m, filsg(noeud), *pt);
    if (estNoeudTas(filsd(noeud), *pt))
      m=indiceDuMax(m, filsd(noeud),*pt);
  }
  if (m != noeud){
    echangeElemTab(pt, m, noeud);
    ordonner(pt, m);
  }
}
```


Complexité - `ordonner`

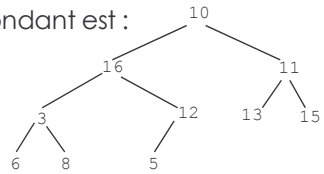
- Hauteur d'un arbre binaire parfait
  - $h(A) = \lfloor \log_2 \text{taille}(A) \rfloor$
- Nb de comparaisons de l'opération ordonner :
  - sur un nœud : de 2 à 3 comparaisons
  - au pire $3\log_2 p$ avec p la taille de l'arbre sur lequel on l'applique

construction d'un tas

- On applique ordonner à tous les noeuds :
  - En remontant de droite à gauche et de bas en haut:
    - on parcourt le tableau dans l'ordre décroissant des indices
  - En commençant au premier noeud qui a un fils
    - C'est le père du dernier élément du tableau

Ex. de construction d'un tas

- Soit la liste :
  - `10 16 11 3 12 13 15 6 8 5`
- L'arbre correspondant est :



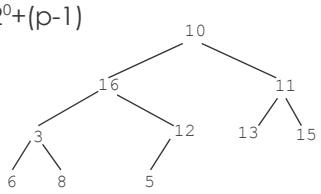
- Le tas correspondant est :
  - `16 12 15 8 10 13 11 6 3 5`

construction d'un tas

```
// entrée: un tableau quelconque
// fait: transforme sur place le tableau  en tas
void construitTas(tableau * pt){
  int i;
  // on part du premier noeud qui a au moins un fils
  // c'est le père du dernier noeud.
  for (i=pere(pt->taille -1); i>=0; i--)
    ordonner(pt, i);
}
```


construction : complexité

- La complexité d'ordonner en nb de comparaisons/transferts : 3*h(noeud)
- Chaque niveau a au plus deux fois plus de noeuds que celui du dessus
- $3*(1*2^2+2*2^1+3*2^0)$
- Généralisons: $3*(p*2^0+(p-1)*2^1+...+1*2^{p-1})=O(n)$



Principe de l'algo

- Après avoir transformé l'arbre en arbre partiellement ordonné, l'élément maximum se retrouve à la racine (tête de liste)
- Echange de l'élément max avec la feuille du dernier niveau la plus à droite
- Application de `ordonner(a)` sur la racine du nouvel arbre (les deux sous-arbres sont déjà partiellement ordonnés)

Exemple

Tri de la liste

55 40 15 30 10 5

Algorithme

```
void triTas(tableau * pt){
  int taille=pt->taille;
  int cb;
  construitTas(pt);
  for (cb=pt->taille -1; cb > 0; cb --){
    echangeElemTab(pt, 0, pt->taille-1);
    pt->taille --;
    ordonner(pt, 0);
  }
  pt->taille=taille;
}
```


Complexité

- Construction: O(n)
- Chaque itération de la boucle
  - 3 transferts (echangerElemTab)
  - Au pire : 3 * hauteur de l'arbre traité (ordonner)
  - Au pire 3+3h(a) transferts et 3h(a) comparaisons
  - Au total, pour l'ensemble des itérations: au pire 3(n+nh(a)) =O(n $\log_2$ n) car l'arbre est parfait
  - complexité en terme de comparaisons aussi en O(n.$\log_2$n)
- Complexité globale en O(n.$\log_2$n)

Tri fusion

- Principe :
  - Découper la liste à trier en deux listes l1 et l2 de taille aussi égale que possible
  - Trier l1 et l2
  - Fusionnner l1 et l2
- Complexite au pire:
  - En nb de comparaisons: θ(n.$\log_2$n)
  - En nb de transferts: θ(n.$\log_2$n)
- Avec des tableaux: de nombreux tableaux auxiliaires
  - Efficace et simple avec une représentation chaînée

Exemple

Opération	liste
Liste initiale	[55 40 15 30 10 5 25 35]
Coupe en deux	[55 40 15 30][10 5 25 35]
Coupe en deux	[55 40][15 30][10 5][25 35]
Coupe en deux	[55][40][15][30][10][5][25][35]
Fusion	[40 55][15 30][5 10][25 35]
Fusion	[15 30 40 55][5 10 25 35]
Fusion	[5 10 15 25 30 35 40 55]

Méthodes de tri - tri fusion

Algorithme

```
liste triFusion(liste l){
  // 0 ou 1 éléments: la liste est déjà triée
  // ce sont les cas d'arrêt
  if ((estVide(l)) || (estVide(l->suivant)))
    return l;
  else { // au moins deux éléments
    liste l1, l2;
    coupeEnDeux(l, &l1, &l2);
    l1=triFusion(l1);
    l2=triFusion(l2);
    return fusion(l1, l2);
  }
}
```

Méthodes de tri - tri fusion

Fusion de listes triées

```
liste fusion(liste l1, liste l2){
  if (estVide(l1))
    return l2;
  else if (estVide(l2))
    return l1;
  else {
    if (l1->valeur < l2->valeur){
      l1->suivant=fusion(l1->suivant, l2);
      return l1;
    } else {
      l2->suivant=fusion(l1, l2->suivant);
      return l2;
    }
  }
}
```

Méthodes de tri - tri fusion

CoupeEnDeux

```
// coupe l à la moitié
void coupeEnDeux(liste l, liste * l1, liste * l2){
  int t=taille(l), cb;
  if ((estVide(l))){ // 0 éléments
    *l2=listeVide();
    *l1=l;
  } else{ // au moins un élément
    liste ltmp=l;
    *l1=l;
    for (cb=0; cb < t/2-1; cb++)
      ltmp=ltmp->suivant;
    *l2=ltmp->suivant;
    ltmp->suivant=listeVide();
  }
}
```

Méthodes de tri - tri fusion

CoupeEnDeux (2)

```
// coupe l en deux: premier dans l1, second dans l2, ...
// on évite le calcul de la taille de la liste
void coupeEnDeux(liste l, liste * l1, liste * l2){
  // 0 ou 1 élément: cas d'arrêt
  if ((estVide(l)) || (estVide(l->suivant))) {
    *l2=listeVide();
    *l1=l;
  } else { // au moins deux éléments
    *l1=l;
    l=l->suivant;
    *l2=l;
    l=l->suivant;
    coupeEnDeux(l,&((*l1)->suivant), &((*l2)->suivant));
  }
}
```

Méthodes de tri

Tri rapide (quicksort)

- Principe
  - Choix d'un élément pivot
  - Partage de la liste en 2 sous-listes contenant
    - Les éléments inférieurs au pivot
    - les éléments supérieurs au pivot
  - Tri des 2 sous-listes
- Famille d'algorithmes en fonction
  - du choix du pivot
  - de la construction des deux sous-listes

Méthodes de tri - tri rapide

Choix du pivot

- But = séparer la liste initiale en 2 sous-listes de taille identique
- Exemple de choix du pivot :
  - Le premier élément
  - Examiner le 1er, le dernier, et l'élément du milieu et choisir l'élément médian comme pivot
  - Choisir le pivot au hasard

Méthodes de tri - tri rapide

Tri des 2 sous-listes

- on ne connaît pas la place finale du pivot
  - on construit la liste des éléments inférieurs à partir du début de la liste
  - on construit la liste des éléments supérieurs à partir de la fin de la liste
- Les éléments qui sont
  - en début de liste et inférieurs au pivot
  - en fin de liste et supérieurs au pivot

  … restent en place
- On échange deux éléments qui ne sont pas à leur place
- A l'issue du tri des sous-listes, le pivot est à sa place définitive

Méthodes de tri - tri rapide

Exemple

Opération	liste
Liste initiale	[55 40 15 30 10 5 25 35]
Choix pivot	[**30** 40 15 35 10 5 25 55]
Echange	[30 25 15 35 10 5 40 55]
Echange	[30 25 15 5 10 35 40 55]
Listes obtenues	[30 25 15 5 10] 35 [40 55]
Liste à 2 éléments	[30 25 15 5 10] 35 40 55
Choix pivot	[10 25 15 5 30] 35 40 55
Echange	[10 5 15 25 30] 35 40 55
Listes obtenues	[10 5] 15 [25 30] 35 40 55
Liste à 2 éléments	5 10 15 [25 30] 35 40 55
Liste à 2 éléments	5 10 15 25 30 35 40 55

Méthodes de tri - tri rapide

Complexité

- Comparaisons :
  - Au mieux et en moyenne : $n.\log_2 n$
  - Au pire : max = n^2 (ràf)
- Transferts :
  - Au mieux : n
  - En moyenne : $n.\log_2 n$
  - Au pire : n^2
- Mémoire :
  - Mémorisation des bornes des sous-listes restant à traiter
  - Pile dont la taille est bornée par $\log_2 n$
  - Complexité en mémoire en O(n)

Méthodes de tri - tri rapide

Complexité

- En pratique, pour n ≤ 10, une version itérative d'un tri par sélection sera plus rapide qu'un tri rapide (nb comparaisons << coût appels récursifs)
  - Dans l'algo, on utilise un tri sélection si n ≤ 10
- En pratique, le code du tri rapide est beaucoup plus léger que celui des tris comme le tri par ta
  - Quand on n'est pas dans le cas le pire, le tri rapide est plus rapide (la constante qui est devant le log n est plus petite)
  - C'est un algorithme très utilisé

Méthodes de tri - Comparaison des méthodes

Complexité

	comparaisons			transferts		
	min	moy	max	min	moy	max
Sélection ordinaire	n^2			n		
Tri bulle	n^2			0	n^2	
Insertion séquentielle	n	n^2		n	n^2	
Insertion dichotomique	n	$n.\log_2 n$		0	n^2	
Tri par tas	$n.\log_2 n$			$n.\log_2 n$		
Tri rapide	$n.\log_2 n$		n^2	n	$n.\log_2 n$	n^2
Tri fusion	$n.\log_2 n$			$n.\log_2 n$		

- Longueur de la liste
  - Méthodes simples → évolution en n^2
  - Méthodes efficaces → évolution en $n.\log_2 n$

Méthodes de tri - Comparaison des méthodes

Temps d'exécution

	500 éléments		2000 éléments		8000 éléments	
	Non triés	triés	Non triés	triés	Non triés	triés
Sélection ordinaire	0,34 s.	0,33 s.	5,4 s.	5,5 s.	106 s.	105 s.
Tri bulle	1,5 s.	0,36 s.	26 s.	6,2 s.	600 s.	129 s.
Insertion séquentielle	0,73 s.	0,0 s.	13 s.	0,06 s.	290 s.	0,22 s.
Insertion dichotomique	0,52 s.	0,0 s.	9,2 s.	0,01 s.	232 s.	0,04 s.
Tri par tas	0,05 s.	0,07s.	0,35 s.	0,38 s.	2,0 s.	1,9 s.
Tri rapide	0,03 s.	0,02 s.	0,24 s.	0,07 s.	1,2 s.	0,5 s.
$n^2/500^2$	1		16		256	
$n.\log_2 n/500.\log_2 500$	1		5		24	

Tri par dénombrement

- Le tri par dénombrement n'est pas un tri par comparaisons
- Les nombres à trier sont compris entre 0 et k
- Pour tout nombre e à trier, on détermine le nombre d'éléments nbInf inférieur à e
- La place de e dans une liste triée, c'est nbInf + 1 (les places commencent à 1 au contraire des indices de tableaux)
- Complexité en nb d'affectations: θ(k+n)
- Aucune comparaison

nb d'éléments inférieurs à e pour tout e

- Calcul à faire pour tous les éléments du tableau
  - Méthode optimisée dont la complexité sera linéaire O(n+k)
- Etapes du calcul:
  - nInf[0..k] tableau initialisé à 0
  - Pour tout e du tableau
    - nInf[e]=nInf[e]+1;
  - nInf[e] contient le nombre de fois où e apparaît dans le tableau d'origine
  - For (cb=1; cb < k; cb++)
    - nInf[e]=nInf[e]+nInf[e-1]
  - nInf[e] contient maintenant le nombre d'éléments inférieurs à e

Algorithme

```
tableau triDenombrement(tableau t, int valMax){
  tableau res=cree_tableau(t.taille);
  tableau nInf=cree_tableau(valMax);
  int cb;
  for (cb=0; cb < valMax; cb ++)
    nInf.tab[cb]=0;
  for (cb=0; cb<t.taille; cb++)
    nInf.tab[t.tab[cb]] ++;
  for (cb=1; cb < valMax; cb ++)
    nInf.tab[cb]+=nInf.tab[cb -1];
  // parcours depuis la fin pour préserver la stabilité
  for (cb=t.taille-1; cb >=0; cb --){
    res.tab[nInf.tab[t.tab[cb]]]=t.tab[cb];
    nInf[t.tab[cb]] --;
  }
  res.taille=t.taille;
  return res;
}
```


Bilan

- Le tri par dénombrement n'est pas un tri par comparaisons
- Les nombres à trier sont compris entre 0 et k
- Pour tout nombre e à trier, on détermine le nombre d'éléments nbInf inférieur à e
- La place de e dans une liste triée, c'est nbInf + 1 (les places commencent à 1 au contraire des indices de tableaux)
- Complexité en nb d'affectations: θ(k+n)
- Aucune comparaison