

Chapitre 4

Algorithmes de tris

Trier un ensemble d'objets consiste à ordonner ces objets en fonction de clés et d'une relation d'ordre définie sur cette clé. Par exemple, chaque étudiant inscrit à l'université d'Aix-Marseille reçoit un numéro qui constitue la clé d'accès à son dossier. On trie des objets lorsqu'ils sont nombreux et qu'on doit pouvoir y accéder rapidement : rechercher un élément dans un tableau non trié est une opération de complexité linéaire dans le nombre d'objets, alors que si le tableau est trié, la recherche peut être effectuée en temps logarithmique. Le tri est donc une opération très fréquente.

On dit qu'un algorithme de tri est

- *en place*, s'il n'utilise pas d'espace mémoire supplémentaire pour stocker les objets,
- *par comparaison*, si le tri s'effectue en comparant les objets entre eux. Il existe des tris qui ne comparent pas les objets entre eux : lorsque les clés sont peu nombreuses, connues à l'avance, et peuvent être indexées facilement.
- Un tri est *stable* s'il préserve l'ordre d'apparition des objets en cas d'égalité des clés. Cette propriété est utile par exemple lorsqu'on trie successivement sur plusieurs clés différentes. Si l'on veut ordonner les étudiants par rapport à leur nom puis à leur moyenne générale, on veut que les étudiants qui ont la même moyenne apparaissent dans l'ordre lexicographique de leurs noms.

Dans ce cours nous distinguerons les tris en $O(n^2)$ (tri à bulle, tri par insertion, tri par sélection), les tris en $O(n \times \log n)$ (tris par fusion, tri par tas et tri rapide, bien que ce dernier n'ait pas cette complexité dans le pire des cas) et les autres (tris spéciaux instables ou pas toujours applicables). Il convient aussi de distinguer le coût théorique et l'efficacité en pratique :

certains tris de même complexité ont des performances très différentes dans la pratique. Le tri le plus utilisé est le tri rapide (*quicksort*).

En général les objets à trier sont stockés dans des tableaux indexés, mais ce n'est pas toujours le cas. Lorsque les objets sont stockés dans des listes chaînées, on peut soit les recopier dans un tableau temporaire, soit utiliser un tri adapté comme le tri par fusion.

4.1 Tri par sélection, tri par insertion, tri à bulle.

Le **tri par sélection** consiste simplement à *sélectionner* l'élément le plus petit de la suite à trier, à l'enlever, et à répéter itérativement le processus tant qu'il reste des éléments dans la suite. Au fur et à mesure les éléments enlevés sont stockés dans une pile. Lorsque la suite à trier est stockée dans un tableau on s'arrange pour représenter la pile dans le même tableau que la suite : la pile est représentée au début du tableau, et chaque fois qu'un élément est enlevé de la suite il est remplacé par le premier élément qui apparaît à la suite de la pile, et prends sa place. Lorsque le processus s'arrête la pile contient tous les éléments de la suite triés dans l'ordre croissant. Le tri par sélection est en $\Theta(n^2)$ dans tous les cas.

Algorithme 12: TriParSelection

entrée : $T[1, n]$ est un tableau d'entiers, $n \geq 1$.

résultat : les éléments de T sont ordonnés par ordre croissant.

début

<p>pour $i := 1$ à $n - 1$ faire</p> <p style="padding-left: 20px;">$j := \text{IndMin}(T[i, n])$ # indice du plus petit élément de $T[i, n]$</p> <p style="padding-left: 20px;">permuter $T[i]$ et $T[j]$</p>

Le **tri par insertion** consiste à *insérer* les éléments de la suite les uns après les autres dans une suite triée initialement vide. Lorsque la suite est stockée dans un tableau la suite triée en construction est stockée au début du tableau. Lorsque la suite est représentée par une liste chaînée on insère les maillons les uns après les autres dans une nouvelle liste initialement vide.

Algorithme 13: TriParInsertion_i

entrée : $T[1, n]$ est un tableau d'entiers, $n \geq 1$.
résultat : les éléments de T sont ordonnés par ordre croissant.
début
 pour $i := 2$ **à** n **faire**
 $j := i$ # indice de l'élément à insérer dans $T[1, i - 1]$
 $v := T[i]$ # valeur de l'élément à insérer
 tant que $j > 1$ *et* $v < T[j - 1]$ **faire**
 $T[j] := T[j - 1]$ # $j - 1$ est une case libre
 $j := j - 1$ # j est une case libre
 $T[j] := v$

Le tri effectue $n - 1$ insertions. A la i ème itération, dans le pire des cas, l'algorithme effectue $i - 1$ recopies. Le coût du tri est donc

$$\sum_{i=2}^n (i - 1) = \frac{n(n - 1)}{2} = O(n^2).$$

Dans le meilleur des cas le tri par insertion requiert seulement $O(n)$ traitements. C'est le cas lorsque l'élément à insérer reste à sa place, donc quand la suite est déjà triée.

Le **tri à bulle** consiste à parcourir le tableau, tant qu'il n'est pas trié, et à permuter les couples d'éléments consécutifs mal ordonnés. On sait que le tableau est trié si lors d'un parcours, aucun couple d'éléments n'a été permuté.

Algorithme 14: TriBulle

entrée : $T[1, n]$ est un tableau d'entiers, $n \geq 1$.
résultat : les éléments de T sont ordonnés par ordre croissant.
début
 $M := n - 1$
 tant que *le tableau n'est peut-être pas trié* **faire**
 pour $i := 1$ **à** M **faire**
 si $T[i] > T[i + 1]$ **alors**
 permuter $T[i]$ et $T[i + 1]$
 $M = M - 1$

Exercice 1 Dans l'algorithme du tri à bulle,

1. montrez qu'après k parcours du tableau (boucle interne), au moins k éléments sont à leur place,
2. déduisez-en que la complexité dans le pire des cas est en $\Theta(n^2)$,
3. montrez qu'après chaque parcours, le plus petit élément du tableau reste à sa place ou recule au plus d'une case,
4. déduisez-en que la complexité moyenne est en $\Theta(n^2)$ (pour tout entier $1 \leq i \leq n$, avec une probabilité $1/n$, l'élément minimal du tableau est situé à l'indice i),
5. montrez que dans le meilleur des cas, l'algorithme est linéaire,
6. montrez que le tri est stable et identifiez précisément la raison de cette propriété.

4.2 Tri par fusion et tri rapide

4.2.1 Tri par fusion (ou par interclassement)

Le tri par fusion (*merge sort* en anglais) implémente une approche de type diviser pour régner très simple : la suite à trier est tout d'abord scindée en deux suites de longueurs égales à un élément près. Ces deux suites sont ensuite triées séparément avant d'être fusionnées (ou interclassées). L'efficacité de ce tri vient de l'efficacité de la fusion : le principe consiste à parcourir simultanément les deux suites triées dans l'ordre croissant de leur éléments, en extrayant chaque fois l'élément le plus petit. La fusion peut-être effectuée en conservant l'ordre des éléments de même valeur (le tri par fusion est stable)

Le tri par fusion est bien adapté aux listes chaînées : pour scinder la liste il suffit de la parcourir en liant les éléments de rangs pairs d'un côté et les éléments de rangs impairs de l'autre. La fusion de deux listes chaînées se fait facilement. Inversement, si la suite à trier est stockée dans un tableau il est nécessaire de faire appel à un tableau annexe lors de la fusion, (au moins dans ses implémentations les plus courantes).

Algorithme 15: TriParFusion

entrée : $T[m, n]$ est un tableau d'entiers, $m \leq n$.
résultat : les éléments de T sont ordonnés par ordre croissant.
début
 si $m < n$ **alors**
 TriFusion($T[m, \lfloor \frac{m+n}{2} \rfloor]$)
 TriFusion($T[\lfloor \frac{m+n}{2} \rfloor + 1, n]$)
 Interclassement($T[m, n]$)

Algorithme 16: Interclassement

entrée : $T[m, n]$ est un tableau d'entiers, $m \leq n$, $T[m, \lfloor \frac{m+n}{2} \rfloor]$ et $T[\lfloor \frac{m+n}{2} \rfloor + 1, n]$ sont triés par ordre croissant.
résultat : les éléments de T sont ordonnés par ordre croissant.
début
 $i := m, j := \lfloor \frac{m+n}{2} \rfloor + 1, k := m$
 tant que $i \leq \lfloor \frac{m+n}{2} \rfloor$ **ET** $j \leq n$ **faire**
 si $T[i] \leq T[j]$ **alors**
 $S[k] := T[i], i := i + 1$
 sinon
 $S[k] := T[j], j := j + 1$
 $k := k + 1$
 # un des deux tableaux est vide
 tant que $i \leq \lfloor \frac{m+n}{2} \rfloor$ **faire**
 $S[k] := T[i], i := i + 1, k := k + 1$
 # recopie de la fin du premier tableau ; la fin du second est en place
 pour $i := 1$ **à** $k - 1$ **faire**
 $T[i] := S[i]$ # recopie de S dans T

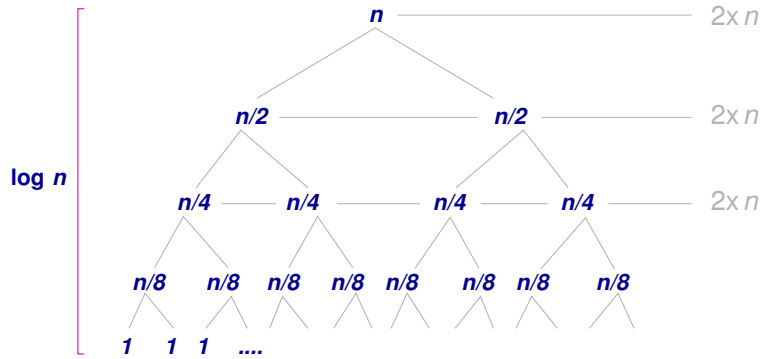
La complexité de l'interclassement est $O(n)$ dans tous les cas. La complexité du tri par fusion vérifie donc l'équation réursive suivante :

$$T(n) = 2 \times T(n/2) + O(n).$$

On en déduit que le tri par fusion est en $O(n \log n)$.

On le vérifie en cumulant les nombres de comparaisons effectuées à chaque niveau de l'arbre qui représente l'exécution de la fonction (voir figure

ci-dessous) : chaque noeud correspond à un appel de la fonction, ses fils correspondent aux deux appels récursifs, et son étiquette indique la longueur de la suite. La hauteur de l'arbre est donc $\log_2 n$ et à chaque niveau le cumul des traitements locaux (scission et fusion) est $O(n)$, d'où on déduit un coût total de $O(n) \times \log_2 n = O(n \log n)$.



Exercice 2

1. Qu'est-ce qui, dans l'algorithme d'interclassement, garantit la stabilité du tri fusion ?
2. Écrivez l'algorithme de fusion lorsque les données sont stockées dans une liste chaînée.

4.2.2 Tri rapide

Le *tri rapide* (*quicksort*) est une méthode de type *diviser pour régner* qui consiste, pour trier un tableau T , à le partitionner en deux sous-tableaux T_1 et T_2 contenant respectivement les plus petits et les plus grands éléments de T , puis à appliquer récursivement l'algorithme sur les tableaux T_1 et T_2 .

Il existe plusieurs méthodes pour partitionner un tableau. Le principe général consiste à utiliser un élément particulier, le *pivot*, comme valeur de partage.

Dans l'algorithme qui suit, le pivot est le premier élément du tableau. On peut également choisir aléatoirement un élément quelconque du tableau (voir TD).

L'algorithme **partition** prend en entrée un tableau $T[m, n]$ (avec $m \leq n$) et réordonne T de façon à

- mettre en début de tableau les éléments $T[i]$ de $T[m+1, n]$ tels que $T[i] \leq T[m]$,
- mettre en fin de tableau les éléments $T[i]$ de $T[m+1, n]$ tels que $T[i] > T[m]$,
- placer $T[m]$ entre les deux.

Il retourne l'indice de $T[m]$ dans le nouveau tableau : $T[m]$ s'appelle le *pivot*.

Algorithme 17: partition

entrée : $T[m, n]$ est un tableau d'entiers.

sortie : l'indice du pivot dans le tableau réarrangé

début

```

    pivot := T[m]
    i := m + 1 # indice où insérer le premier élément ≤ T[m]
    pour j := m + 1 à n faire
        # j est l'indice de l'élément courant
        si T[j] ≤ pivot alors
            permuter T[i] et T[j]
            i := i + 1
    permuter T[m] et T[i - 1]
    retourner i - 1

```

Le **QuickSort** consiste à partitionner un tableau $T[m, n]$ autour du pivot $T[m]$ puis à trier récursivement chacune des deux parties $T[m, \text{IndPivot} - 1]$ et $T[\text{IndPivot} + 1, n]$ à l'aide du **QuickSort**.

Algorithme 18: QuickSort

entrée : $T[m, n]$ est un tableau d'entiers.

sortie : T est trié par ordre croissant

début

```

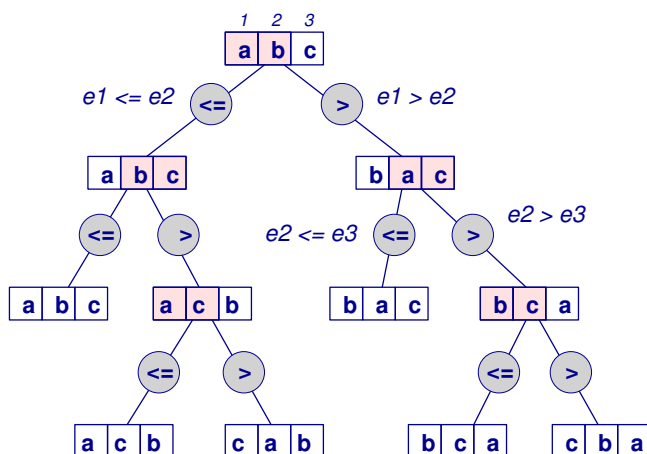
    si m < n alors
        IndPivot = partition(T[m, n])
        QuickSort(T[m, IndPivot - 1])
        QuickSort(T[IndPivot + 1, n])

```

On peut montrer que la complexité du tri rapide est $O(n \log n)$ en moyenne, mais aussi $O(n^2)$ dans le pire des cas (voir une étude de la complexité en TD). En pratique, c'est l'algorithme le plus utilisé et très souvent, le plus rapide.

4.3 Complexité optimale d'un algorithme de tri par comparaison

L'*arbre de décision* d'un tri par comparaison représente le comportement du tri dans toutes les configurations possibles. Les configurations correspondent à toutes les permutations des objets à trier, en supposant qu'ils soient tous comparables et de clés différentes. S'il y a n objets à trier, il y a donc $n!$ configurations possibles. On retrouve toutes ces configurations sur les feuilles de l'arbre, puisque deux permutations initiales distinctes ne peuvent pas produire le même comportement du tri : en effet, dans ce cas le tri n'aurait pas fait son travail sur un des deux ordonnancements. Chaque noeud de l'arbre correspond à une comparaison entre deux éléments et a deux fils, correspondants aux deux ordres possibles entre ces deux éléments.



Sur la figure ci-dessus nous avons représenté l'arbre de décision du tri par insertion sur une suite de trois éléments. A la racine de l'arbre le tri compare tout d'abord les deux premiers éléments de la suite a et b , afin d'insérer l'élément b dans la suite triée constituée uniquement de l'élément a . Suivant leur ordre les deux éléments sont permutés (fils droit) ou laissés en place (fils gauche). Au niveau suivant l'algorithme compare les éléments de rang 2 et de rang 3 afin d'insérer l'élément de rang 3 dans la suite triée constituée des 2 premiers éléments, et ainsi de suite. Les branches de l'arbre de décision du tri par insertion n'ont pas toutes la même longueur du fait que

dans certains cas l'insertion d'un élément est moins coûteuse, en particulier quand l'élément est déjà à sa place.

Les feuilles de l'arbre représentent toutes les permutations du tableau en entrée. Puisque le nombre de permutations de n éléments est $n!$, l'arbre de décision d'un tri a donc $n!$ feuilles. On peut montrer facilement, par récurrence sur h , que le nombre de feuilles n_f d'un arbre binaire de hauteur h , vérifie $n_f \leq 2^h$. On en déduit que la hauteur h de l'arbre de décision vérifie

$$h \geq \log(n!)$$

or, d'après la formule de Stirling¹

$$\log(n!) \sim \log \sqrt{2\pi n} + \log \left(\frac{n}{e}\right)^n = \Theta(n \log n)$$

et donc la hauteur minimale de l'arbre est en $\Omega(n \log n)$.

Comme la hauteur de l'arbre est aussi la longueur de sa plus longue branche, on en conclut qu'aucun tri **par comparaison** ne peut avoir une complexité dans le pire des cas inférieure à $O(n \times \log n)$.

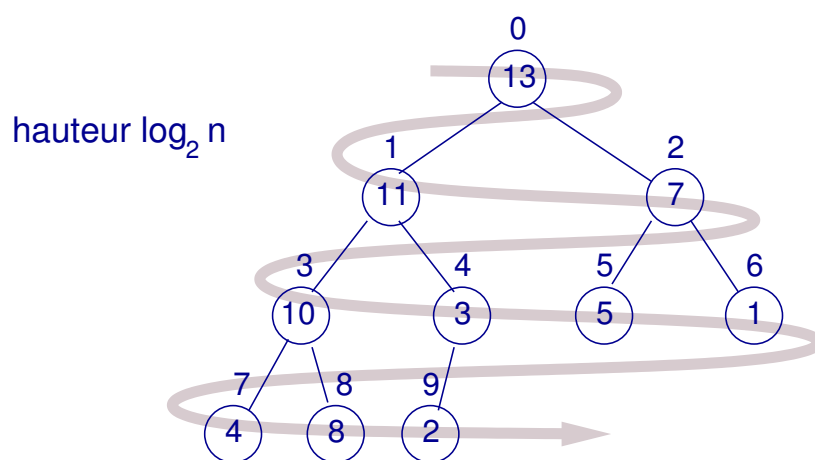
4.4 Tri par tas.

Un *tas* (*heap* en anglais) ou *file de priorité* (*priority queue*) est un arbre binaire étiqueté presque complètement équilibré : tous ses niveaux sont remplis sauf peut-être le dernier, qui est rempli à partir de la gauche jusqu'à un certain noeud. On définit ensuite une propriété sur les tas : chaque noeud est dans une relation d'ordre fixée avec ses fils. En général on considère des tas dans lesquels chaque noeud a une valeur plus petite que celles de ses fils. Pour le tri par tas on utilise des *arbres maximiers* (*max-heap property*), c'est-à-dire des tas dans lesquels chaque noeud porte une valeur plus grande que celles de ses fils. La valeur la plus grande du tas se trouve donc à la racine. Les opérations et les techniques présentées dans ce chapitre pour des arbres maximiers s'appliquent de la même façon à des tas basés sur l'ordre inverse. Les opérations essentielles sur les tas sont :

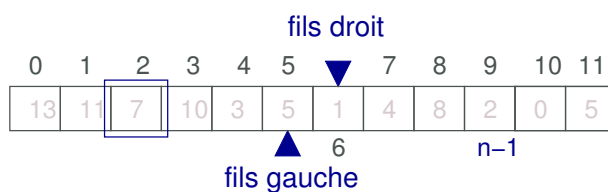
- la construction du tas,
- l'extraction du maximum,
- l'ajout d'une nouvelle valeur,
- la modification de la valeur d'un noeud.

1. sans utiliser la formule de Stirling, on peut remarquer que $(\frac{n}{2})^{\frac{n}{2}} \leq n! \leq n^n$, ce qui conduit directement à $\log(n!) = \Theta(n \log n)$.

Habituellement les tas représentés dans des tableaux. Les valeurs du tas sont stockées dans les premières cases du tableau. Si le tas est composé de n éléments, ces derniers apparaissent donc aux indices $0, 1, \dots, n-1$. La racine du tas figure dans la case d'indice 0. La disposition des éléments du tas dans le tableau correspond à un parcours de l'arbre par niveau, en partant de la racine et de gauche à droite.



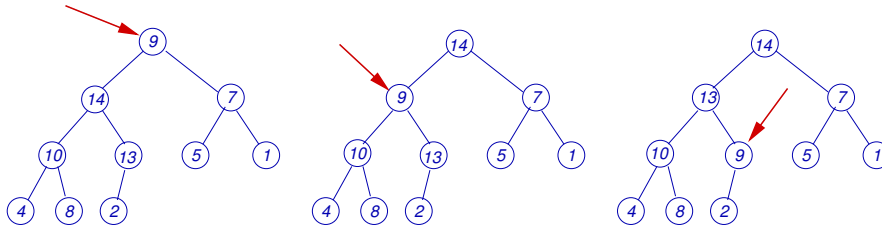
Le fils gauche du noeud qui figure à l'indice i , s'il existe, se trouve à l'indice $\text{FilsG}(i) = 2 \times i + 1$, et son fils droit, s'il existe, se trouve à l'indice $\text{FilsD}(i) = 2 \times i + 2$. Inversement, le père du noeud d'indice i non nul se trouve à l'indice $\lfloor \frac{i-1}{2} \rfloor$.



Cette disposition entraîne que l'arbre est forcément presque complètement équilibré (i.e. toutes ses branches ont la même longueur à un élément près), et les plus longues branches sont à gauche. La hauteur d'un tas, i.e. son

nombre de niveaux, contenant n éléments est donc $\lfloor \log_2 n \rfloor + 1$ (le nombre de fois que l'on peut diviser n par 2 avant d'obtenir 1). Si le dernier noeud du tas se trouve à la position $n-1$, son père se trouve à la position $\lfloor \frac{n}{2} - 1 \rfloor$. C'est le dernier noeud qui a au moins un fils. Les feuilles de l'arbre se trouvent donc entre la position $\lfloor \frac{n}{2} \rfloor$ et la position $n-1$ dans le tableau puisqu'il n'y a pas de feuilles avant le dernier père.

L'opération *Entasser* est une opération de base sur les tas. Elle est utilisée notamment pour la construction des tas ou encore pour l'extraction de la valeur maximale. Elle consiste à reconstruire le tas lorsque seule la racine viole (éventuellement) la propriété de supériorité entre un noeud et ses fils, en faisant descendre dans l'arbre l'élément fautif par des échanges successifs.



Fonction Entasser(i, T, n)

entrée : T est un tas ; le fils gauche et le fils droit du noeud d'indice i vérifient la propriété *Max-heap* ; ce n'est pas forcément le cas du noeud d'indice i .

sortie : La propriété *max-heap* est vérifiée par le noeud d'indice i .

début

```

     $iMax \leftarrow i$ 
    si (FilsG( $i$ ) <  $n$ ) ET ( $T[\text{FilsG}(i)] > T[iMax]$ ) alors
         $iMax := \text{FilsG}(i)$ 
    si (FilsD( $i$ ) <  $n$ ) ET ( $T[\text{FilsD}(i)] > T[iMax]$ ) alors
         $iMax := \text{FilsD}(i)$ 
    si ( $iMax \neq i$ ) alors
        Echanger  $T[i]$  et  $T[iMax]$ 
        Entasser( $iMax, T, n$ )

```

— la condition ($\text{FilsG}(i) < n$) signifie que le fils gauche du noeud i existe ;

- si le nœud i est une feuille ou si $T[i]$ est supérieure aux valeurs portées par ses fils, l'algorithme ne fait rien ;
- si le nœud i a des fils qui portent des valeurs supérieures à $T[i]$, la fonction échange cette valeur avec la plus grande des valeurs de ses fils et effectue un appel récursif sur le fils qui porte la valeur $T[i]$.

La fonction **Entasser** a un coût en $O(\log_2 n)$ puisque, dans le pire des cas, il faudra parcourir une branche entièrement.

Extraction de la valeur maximale. La valeur maximale d'un tas qui vérifie la propriété *Max-heap* est à la racine de l'arbre. Pour un tas de taille n stocké dans le tableau T c'est la valeur $T[0]$ si n est non nul. L'extraction de la valeur maximale consiste à recopier en $T[0]$ la dernière valeur du tas $T[n-1]$, à décrémenter la taille du tas, puis à appliquer l'opération *Entasser* à la racine du tas, afin que la nouvelle valeur de la racine prenne sa place.

Fonction ExtraireLeMax(T, n)

entrée : T est un tableau, n est la taille du tas stocké dans T .

sortie : Retourne la valeur maximale et met à jour le tas.

début

$max := T[0]$
 $T[0] := T[n-1]$
 Entasser($0, T, n-1$)
retourner $\langle max, T, n-1 \rangle$

fin

Insérer une nouvelle valeur dans un tas consiste à ajouter la valeur à la fin du tas, en dernière position dans le tableau, puis à la faire remonter dans le tas en l'échangeant avec son père tant qu'elle ne se trouve pas à la racine et que la valeur de son père lui est inférieure. L'opération d'insertion n'est pas utile pour le tri par tas.

Principe général du tri par tas. Supposons que l'on ait à trier une suite de n valeurs stockée dans un tableau T . On commence par construire un tas dans T avec les valeurs de la suite. Ensuite, tant que le tas n'est pas vide on répète l'extraction de la valeur maximale du tas. Chaque fois, la valeur extraite est stockée dans le tableau immédiatement après les valeurs du tas. Lorsque le processus se termine on a donc la suite des valeurs triée dans le tableau T .

Construction du tas.

Les valeurs de la suite à trier sont stockées dans le tableau T . La procédure consiste à parcourir les noeuds qui ont des fils et à leur appliquer l'opération *Entasser*, en commençant par les noeuds qui sont à la plus grande profondeur dans l'arbre. Il suffit donc de parcourir les noeuds dans l'ordre décroissant des indices et en partant du dernier noeud qui a des fils, le noeud d'indice $\lfloor (n/2) \rfloor - 1$. L'ordre dans lequel les noeuds sont traités garantit que les sous-arbres sont des tas.

Fonction ConstruireUnTas(T, n)

entrée : T est un tableau, n est un entier.

sortie : Structure les n premiers éléments de T sous forme de tas.

début

pour $i := \lfloor n/2 - 1 \rfloor$ **à 0 faire**

 Entasser(i, T, n)

finpour

retourner T

fin

Invariant : tous les nœuds de T d'indice $> i$ vérifient la propriété *max-heap*.

Complexité de la construction. De façon évident la complexité est au plus $O(n \log n)$. En effet la hauteur du tas est $\log n$ et on effectue $n/2$ fois l'opération *Entasser*. En fait le coût de la construction est $O(n)$. En effet, la fonction effectue un grand nombre d'entassements sur des arbres de petites hauteur (pour les noeuds les plus profonds), et très peu d'entassements sur la hauteur du tas.

Le niveau d'un nœud d'indice i est : $\lfloor \log_2(i+1) \rfloor + 1$.

Comme le nombre total de niveau est $\lfloor \log_2 n \rfloor + 1$, la hauteur du sous-arbre de racine le nœud d'indice i est donc : $\lfloor \log_2 n \rfloor - \lfloor \log_2(i+1) \rfloor + 1$.

On en déduit que la complexité de l'algorithme *Entasser*(i, T, n) est : $O(\log_2 \frac{n}{i+1})$.

La complexité de la construction est donc égale à :

$$\sum_{i=0}^{\lfloor n/2-1 \rfloor} O(\log_2 \frac{n}{i+1}) = O(\sum_{i=1}^n \log_2 \frac{n}{i}) = O(\log_2 \frac{n^n}{n!}).$$

D'après la formule de Stirling, $n! = \Omega((n/e)^n)$. On en déduit que

$$\log_2 \frac{n^n}{n!} = O(\log_2 e^n) = O(n).$$

La complexité de la construction d'un tas est donc linéaire dans le nombre d'élément à entasser.

Exercice : construisez un tas à partir de $T = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$.

Tri par tas.

On construit un tas. On utilise le fait que le premier élément du tas est le plus grand : autant de fois qu'il y a d'éléments, on extrait le premier du tas et on reconstruit le tas avec les éléments restants. Enlever le premier élément consiste simplement à l'échanger avec le dernier du tas et à décrémenter la taille du tas. On rétablit la propriété de tas en appliquant l'opération *Entasser* sur ce premier élément.

Fonction $\text{TriParTas}(T, n)$

entrée : T est un tableau de n éléments.

sortie : Trie le tableau T par ordre croissant.

début

```

  ConstruireUnTas( $T, n$ )
  pour  $i := n - 1$  à 1 faire
    Echanger( $T, 0, i$ )
    Entasser( $0, T, i$ )
  retourner  $T$ 
```

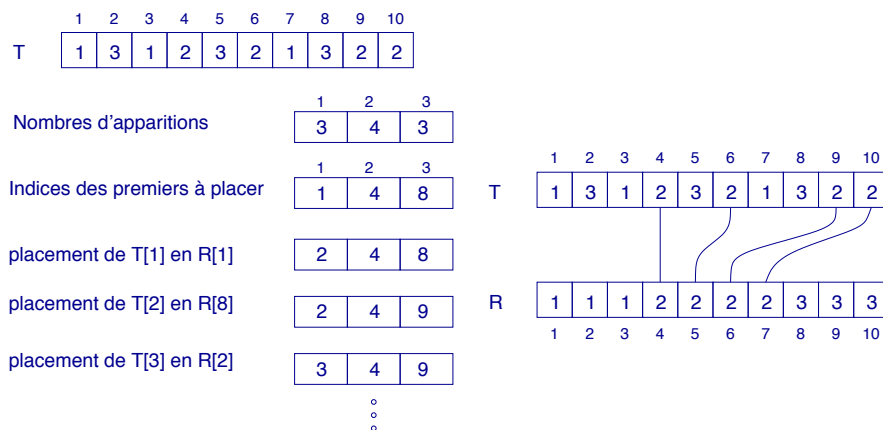
La construction du tas coûte $O(n)$. On effectue ensuite n fois un échange et l'opération *Entasser* sur un tas de hauteur au plus $\log n$. La complexité du tri par tas est donc $O(n \log n)$.

4.5 Tri par dénombrement, tri par base.

4.5.1 Tri par dénombrement

Le tri par dénombrement (*counting sort*) est un tri sans comparaisons qui est stable, c'est-à-dire qu'il respecte l'ordre d'apparition des éléments dont les clés sont égales. Un tri sans comparaison suppose que l'on sait indexer les éléments en fonction de leur clé. Par exemple, si les clés des éléments à trier sont des valeurs entières comprises entre 0 et 2, on pourra parcourir

les éléments et les répartir en fonction de leur clé sans les comparer, juste en utilisant les clés comme index. Le tri par dénombrement utilise cette propriété pour tout d'abord recenser les éléments pour chaque valeur possible des clés. Ce comptage préliminaire permet de connaître, pour chaque clé c , la position finale du premier élément de clé c qui apparaît dans la suite à trier. Sur l'exemple ci-dessous on a recensé dans le tableau T, 3 éléments avec la clé 0, 4 éléments avec la clé 1 et 3 éléments avec la clé 2. On en déduit que le premier élément avec la clé 0 devra être placé à la position 0, le premier élément avec la clé 1 devra être placé à la position 3, et le premier élément avec la clé 2 devra être placé à la position 7. Il suffit ensuite de parcourir une deuxième fois les éléments à trier et de les placer au fur et à mesure dans un tableau annexe (le tableau R de la figure), en n'oubliant pas, chaque fois qu'un élément de clé c est placé, d'incrémenter la position de l'objet suivant de clé c . De cette façon les éléments qui ont la même clés apparaissent nécessairement dans l'ordre de leur apparition dans le tableau initial.



Fonction TriParDenombrements(T, n)

entrée : T est un tableau de n éléments.

sortie : R contient les éléments de T triés par ordre croissant.

début

/* Initialisations */

pour $i := 1$ à k **faire**

 | $Nb[i] := 0$

/* Calcul des nombres d'apparitions */

pour $i := 1$ à n **faire**

 | $Nb[T[i]] := Nb[T[i]] + 1$

/* Calcul des indices du premier */

 $Nb[k] := n - Nb[k] + 1$

/* Élément de chaque catégorie */

pour $i := k - 1$ à 1 **faire**

 | $Nb[i] := Nb[i + 1] - Nb[i]$

 /* Recopie des éléments originaux du tableau T dans R */

pour $i := 1$ à n **faire**

 | $R[Nb[T[i]]] := T[i]$

 | $Nb[T[i]] := Nb[T[i]] + 1$
retourner R

La suite des objets à trier est parcourue deux fois, et la table Nb contenant le nombre d'occurrences de chaque clé est parcourue une fois pour l'initialiser. La complexité finale est donc $O(n + k)$ si les clés des éléments à trier sont comprises entre 0 et k . Le tri par dénombrement est dit *linéaire* (modulo le fait que k doit être comparable à n).

4.5.2 Tri par base.

Le tri par dénombrement est difficilement applicable lorsque les valeurs que peuvent prendre les clés sont très nombreuses. Le principe du tri par base (*radix sort*) consiste, dans ce type de cas, à fractionner les clés, et à effectuer un tri par dénombrement successivement sur chacun des fragments des clés. Si on considère les fragments dans le bon ordre (i.e. en commençant par les fragments de poids le plus faible), après la dernière passe, l'ordre des éléments respecte l'ordre lexicographique des fragments, et donc la suite est triée.

Considérons l'exemple suivant dans lequel les clés sont des nombres entiers à au plus trois chiffres. Le fractionnement consiste simplement à

prendre chacun des chiffres de l'écriture décimale des clés. La colonne de gauche contient la suite des valeurs à trier, la colonne suivante contient ces mêmes valeurs après les avoir trié par rapport au chiffre des unités, . . . Dans la dernière colonne les valeurs sont effectivement triées. Du fait que le tri par dénombrement est stable, si des valeurs ont le même chiffre des centaines, alors elles apparaîtront dans l'ordre croissant de leurs chiffres des dizaines, et si certaines ont le même chiffre des dizaines alors elles apparaîtront dans l'ordre croissant des chiffres des unités.

536	592	427	167
893	462	536	197
427	893	853	427
167	853	462	462
853	536	167	536
592	427	592	592
197	167	893	853
462	197	197	893

Supposons que l'on ait n valeurs dont les clés sont fractionnées en c fragments avec k valeurs possibles pour chaque fragment. Le coût du tri par base est alors $O(c \times n + c \times k)$ puisque l'on va effectuer c tris par dénombrement sur n éléments avec des clés qui auront k valeurs possibles.

Si $k = O(n)$ on peut dire que le tri par base est linéaire. Dans la pratique, sur des entiers codés sur 4 octets que l'on fragmente en 4, le tri par base est aussi rapide que le *Quick Sort*.

4.6 Tri shell.

C'est une variante du tri par insertion. Le principe du tri shell est de trier séparément des sous-suites de la table formées par des éléments pris de h en h dans la table (on nommera cette opération *h-ordonner*).

Définition. La suite $E = (e_1, \dots, e_n)$ est *h-ordonnée* si pour tout indice $i \leq n - h$, $e_i \leq e_{i+h}$.

Si h vaut 1 alors une suite *h-ordonnée* est triée.

Pour trier, Donald Shell le créateur de ce tri, propose de *h-ordonner* la suite pour une série décroissante de valeurs de h . L'objectif est d'avoir une série de valeurs qui permette de confronter tous les éléments entre eux le plus souvent et le plus tôt possible. Dans la procédure ci-dessous la suite des valeurs de h est : . . . , 1093, 364, 121, 40, 13, 4, 1.

La complexité de ce tri est $O(n^2)$. Avec la suite de valeurs de h précédente on atteint $O(n^{3/2})$ (admis). Cependant dans la pratique ce tri est très performant et facile à implémenter (conjectures $O(n \times (\log n)^2)$ ou $n^{1,25}$).

Algorithme 19: TriShell

entrée : $T[1, n]$ est un tableau d'entiers, $n \geq 1$.

résultat : les éléments de T sont ordonnés par ordre croissant.

début

```

   $h := 1$ 
  tant que  $h \leq n/9$  faire
     $h := 3h + 1$ 
  tant que  $h > 0$  faire
    /* Tri par insertion pour  $h$ -ordonner  $T$  */
    pour  $i := h + 1$  à  $n$  faire
       $j := i, v := e_i$ 
      tant que  $((j > h) \text{ et } (v < e_{j-h}))$  faire
         $e_j := e_{j-h}, j := j - h$ 
       $e_j := v$ 
     $h := h/3$ 

```

On notera que le tri utilisé pour h -ordonner la suite est un tri par insertion. La dernière fois que ce tri est appliqué (avec h qui vaut 1) on exécute donc simplement un tri par insertion. Les passes précédentes ont permis de mettre en place les éléments de façon à ce que cette ultime exécution du tri par insertion soit très peu coûteuse.