Chapitre 14

Informatique : Tris

14.1 Introduction

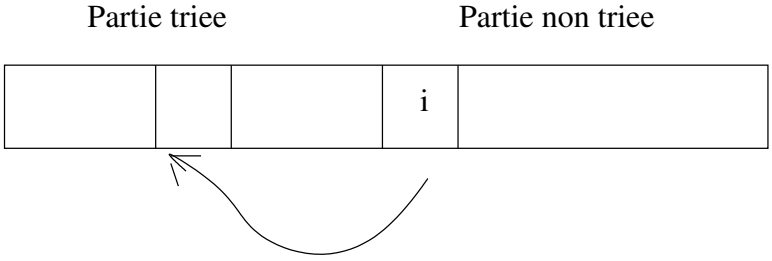
Algorithmes de tris Le tri d'un ensemble d'objets consiste à les ordonner en fonction de clés et d'une relation d'ordre définie sur cette clé. Le tri est une opération classique et très fréquente. De nombreux algorithmes et méthodes utilisent des tris. Par exemple pour l'algorithme de Kruskal qui calcule un arbre couvrant de poids minimum dans un graphe, une approche classique consiste, dans un premier temps, à trier les arêtes du graphe en fonction de leurs poids. Autre exemple, pour le problème des éléphants, trouver la plus longue séquence d'éléphants pris dans un ensemble donné, telle que les poids des éléphants dans la séquence soient croissants et que leurs Q.I. soient décroissants, une approche classique consiste à considérer une première suite contenant tous les éléphants ordonnés par poids croissants, une deuxième suite avec les éléphants ordonnés par Q.I. décroissants, puis à calculer la plus longue sous séquence commune à ces deux suites. Trier un ensemble d'objets est aussi un problème simple, facile à décrire, et qui se prête à l'utilisation de méthodes diverses et variées. Ceci explique l'intérêt qui lui est porté et le fait qu'il est souvent présenté comme exemple pour les calculs de complexité. Dans le cas général on s'intéresse à des tris en place, c'est-à-dire des tris qui n'utilisent pas d'espace mémoire supplémentaire pour stocker les objets, et par comparaison, c'est-à-dire que le tri s'effectue en comparant les objets entres eux. Un tri qui n'est pas par comparaison nécessite que les clés soient peu nombreuses et 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 $O(n \times \ln n)$ (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 et globalement le plus rapide est le tri rapide (un bon nom : quicksort) ; nous l'étudierons en TD. 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 chainées, on peut soit les recopier dans un tableau temporaire, soit utiliser un tri adapté comme le tri par fusion.

14.2 Tri par sélection, tri par insertion.

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 insertion, c'est le tri du joueur de cartes, 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 chainée on insère les maillons les uns après les autres dans une nouvelle liste initialement vide. partie trié et partie non trié



Algorithme TriParInsertion: Tri d'un tableau par insertion
entrée : $T[1,n]$ est un tableau d'entiers, $n \geq 1$.
résultat : les éléments de T sont ordonnés par ordre croissant.
débutpour $i = 2$ à n
faire $j := i, v := T[i]$,
tant que $j > 1$ et $v < T[j-1]$
faire $T[j] := T[j-1]$ et $j := j-1$
fintq
$T[j] := v$
finpour
fin

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 $\sum_{i=1}^{n} i - 1 = \mathrm{O}(n^2)$. Remarquons que 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 déja triée (lorsque la suite est stockée dans une liste chainée c'est la cas lorsque la liste est triée à l'envers puisqu'on insère en tête de liste).

En Python : def IndiceMin(T, i) :
res $= i$
for k in range(len(T)) $[i+1:]$:
if $T[k] < T$[res] : res $= k$
return res
def TriSelection(T) :
for i in range(len(T) -1) :
ind $=$ IndiceMin(T, i)
$T[i], T[ind] = T[ind], T[i]$
return T

14.3 Tri par fusion et tri rapide

14.3.1 Tri par fusion

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. L'efficacité du tri par fusion 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

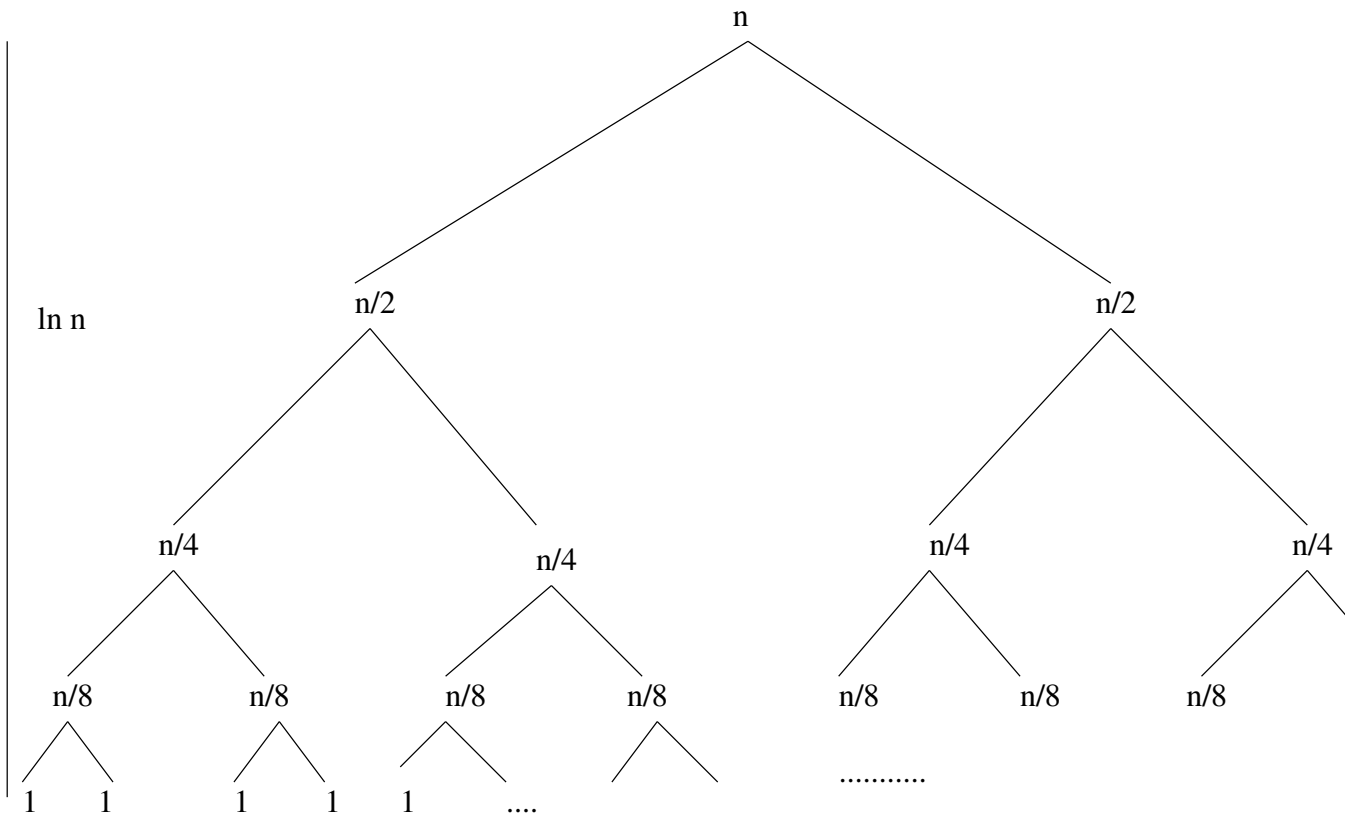
le plus petit. Le tri par fusion est bien adapté aux listes chainées : pour scinder la liste il suffit de la parcourir en liant les éléments de rangs pairs d'un coté et les éléments de rangs impairs de l'autre. La fusion de deux listes chainées se fait facilement. Inversement, si la suite à trier est stockée dans un tableau il est nécessaire de faire appel à un tableau annexe lors de la fusion, sous peine d'avoir une complexité en $O(n^2)$.

```
Fonction TriParFusion(S)
Data : S: la suite à trier
begin
    if |S| ≥ 1then          scinder la suite S en deux suites S1 et S2 de longueurs égales;
        S1 := TriParFusion (S1);
        S2 := TriParFusion (S2);
        S := fusion (S1, S2);
    return S;
```

Dans le cas général, on peut évaluer à $O(n)$ le coût de la scission de la suite S et à $O(n)$ le coût de la fusion des suites S_1 et S_2. L'équation récursive du tri par fusion est donc

$$T(n) = 1 + O(n) + 2T(n/2) + O(n)$$

On en déduit que le tri par fusion est en $O(n \ln n)$. 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 $O(n) \times log_2 n = O(nlogn)$.



La fusion peut-être effectuée en conservant l'ordre des éléments de même valeur (le tri par fusion est stable), mais elle nécessite l'utilisation de tableaux auxiliaires (au moins dans ses implémentations les plus courantes).

Exercice 1 Exercice 1 Écrivez l'algorithme de fusion.

```
def TriFusion(T) :
        if len(T) > 1 :
        if len(T) == 2 :
                if T[0] > T[1] : T[0], T[1] = T[1], T[0]
                        else :
                        m = len(T)/2
        T =Fusion(TriFusion(T[: m]),TriFusion(T[m :]))
        return T
def Fusion (L1, L2) :
        res = []
        i1 = i2 = 0
        while i1 < len(L1) and i2 < len(L2) :
                if L1[i1] < L2[i2] :
                        res.append(L1[i1])
                        i1 = i1 + 1
                else :
                        res.append(L2[i2])
                i2 = i2 + 1
                if i1 == len(L1) :
                        res + = L2[i2 :]
                else :
                        res + = L1[i1 :]
        return res
```

14.3.2 Tri rapide

Le tri rapide est une méthode de type diviser pour régner. L'idée de base est la suivante : pour trier la suite $S = (s_g, \cdots, s_d)$ on la partitionne en deux sous suites non vides $S' = (s_g, \cdots, s_q)$ et $S'' = (s_q, \cdots, s_d)$ telles que les éléments les plus petits sont dans la première et les éléments les plus grands dans la seconde. En appliquant récursivement le tri sur les suites S' et S'' la suite S est triée. Il existe plusieurs méthodes pour partitionner une suite. Le principe général consiste à utiliser un élément particulier de la suite, le pivot, comme valeur de partage. Il faut faire très attention à ne pas produire une suite vide et l'autre contenant tous les éléments de la suite initiale, auquel cas le tri risque de boucler indéfiniment. Une façon simple de contourner ce problème est d'isoler les éléments de même valeur que le pivot. Ces éléments sont simplement placés entre les deux sous suites après la partition, c'est leur place définitive. Nous allons écrire plusieurs versions de la partition, en utilisant des tableaux pour stocker les éléments de la suite à trier.

Question 1. Ecrivez une fonction partition en $O(n)$ où n est le nombre d'éléments de la suite en vous inspirant de la méthode dite du drapeau : la partie du tableau où sont stockés les éléments de la suite est segmentée en quatre zones, contenant respectivement des éléments de valeur inférieure au pivot, des élément de même valeur que le pivot, des éléments de valeur supérieure au pivot et enfin les éléments restant (qui n'ont pas encore été traités). Le partitionnement consiste simplement à prendre un élément de la dernière zone et à l'ajouter dans l'une des trois premières zones suivant sa valeur, puis à répéter cette opération tant qu'il reste des éléments non traités.

Question 2. Ecrivez une fonction récursive de tri rapide qui utilise la fonction de partition précédente.

Question 3 . En terme de complexité, quel est pire des cas pour le tri rapide, et comment éviter ce cas avec une bonne probabilité.

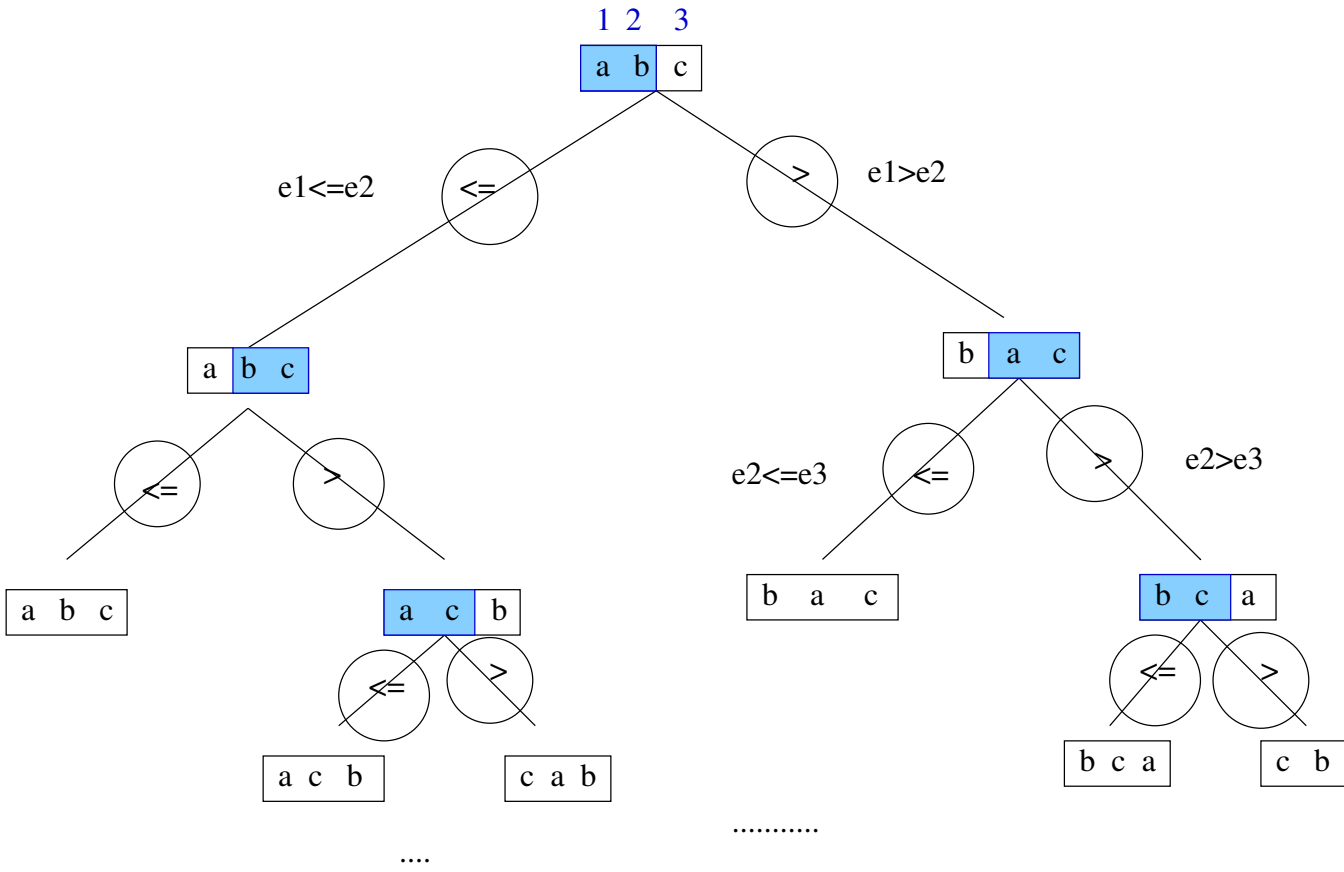
Question 4. Il existe une façon plus efficace pour partitionner la suite : le principe général consiste à parcourir la suite simultanément en partant de la gauche (et en allant vers la droite) et en partant de la droite (et en allant vers la gauche), afin de trouver à gauche une valeur supérieure à la valeur du pivot, et à droite une valeur inférieure à la valeur du pivot. Il suffit ensuite d'échanger ces deux valeurs et de continuer le processus tant que les deux indices ne se sont pas croisés. Cette partition produit deux sous suites, l'une contenant des valeurs plus

petites ou égales à la valeur du pivot et l'autre contenant des valeurs plus grandes ou égales à la valeur du pivot. Ecrivez une fonction de partition qui suit ce schéma et justifiez la, c'est à dire démontrez qu'elle se termine et que le résultat est le bon. En particulier il vous faut démontrer qu'aucune des deux suites produites ne peut être vide. Ecrivez une nouvelle version du tri rapide qui intégre cette partition.

Question 5. Ecrivez une fonction qui, étant donné un entier k et une suite S, renvoie la valeur du $k$ème plus petit élément du tableau, c'est à dire de l'élement de rang k dans la suite triée. Cette fonction fera appel à la fonction partition. Dans quels cas cette méthode est-elle meilleure que si on avait tout d'abord trié la suite.

14.4 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'élement de rang 3 dans la suite triée constituée des 2 premiers éléments, et ainsi de suite. Remarquons que 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 couteuse, en particulier quand l'élement est déja à sa position.

Puisque le nombre de permutations de n él éments est n!, l'arbre de décision d'un tri a donc n! feuilles. La hauteur d'un arbre binaire de n! feuilles est, dans le meilleur des cas, c'est-à-dire si l'arbre est parfaitement équilibré (le nombre de noeuds est multiplié par deux chaque fois qu'on descend d'un niveau)

$$h = \ln(n!)$$

or, d'après la formule de Stirling,

$$\ln(n!) \geq \ln(\frac{n}{e})^n \simeq n \times \ln n - n \times \ln e$$

et donc la hauteur minimale de l'arbre est de l'ordre de $n \times \ln n$. On en conclut qu'aucun tri par comparaison ne peut avoir une complexité meilleure que $O(n \times \ln n)$.

14.5 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), c'est-àdire 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 faÂ¸con à des tas basés sur l'ordre inverse.

Les opérations essentielles sur les tas sont :

-la construction du tas,

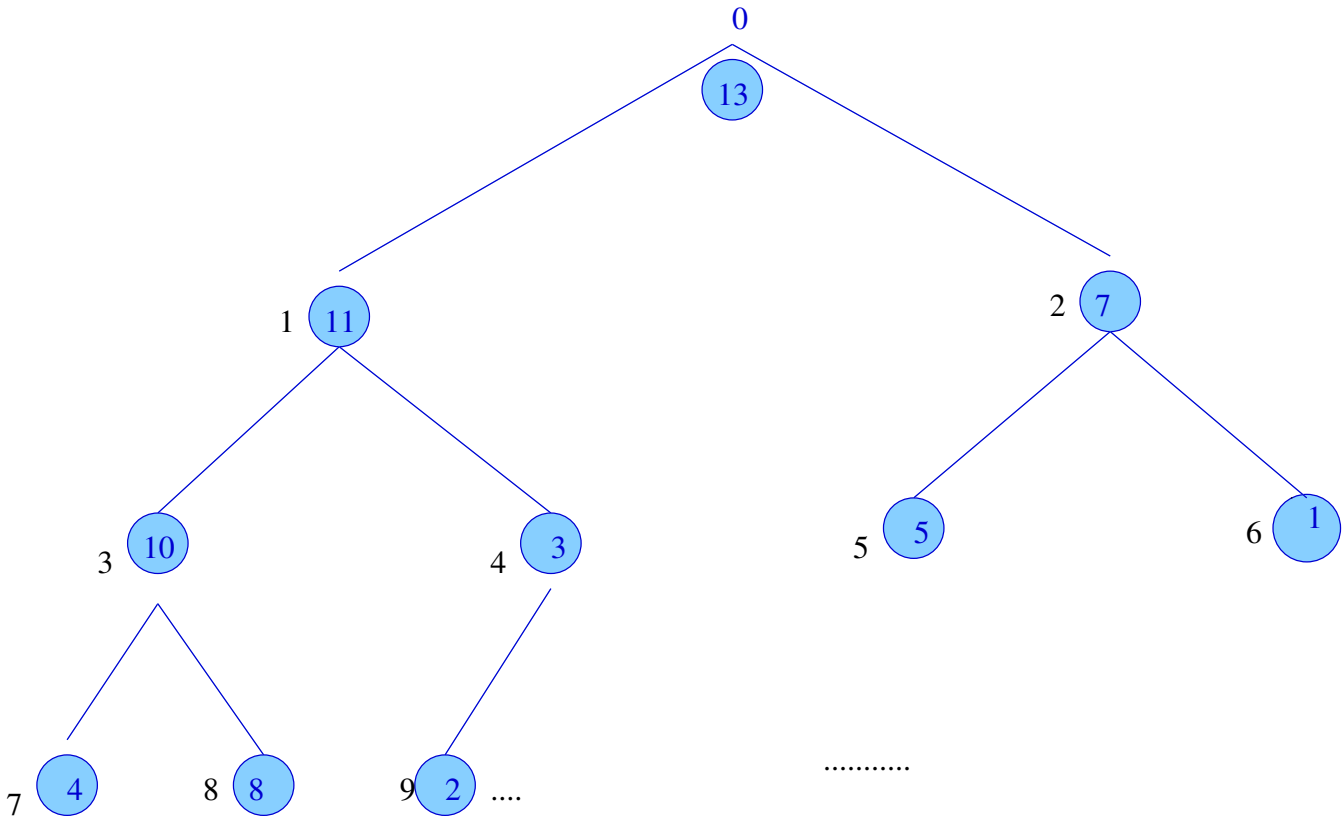
-l'extraction du maximum,

-l'ajout d'une nouvelle valeur,

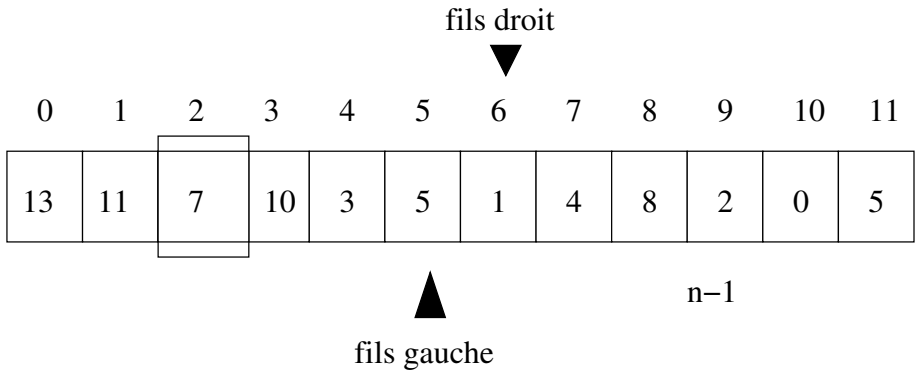
-la modification de la valeur d'un noeud.

Habituellement les tas sont des arbres binaires 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 $0, 1, \cdots, n-1$. 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.

hauteur ln_2n



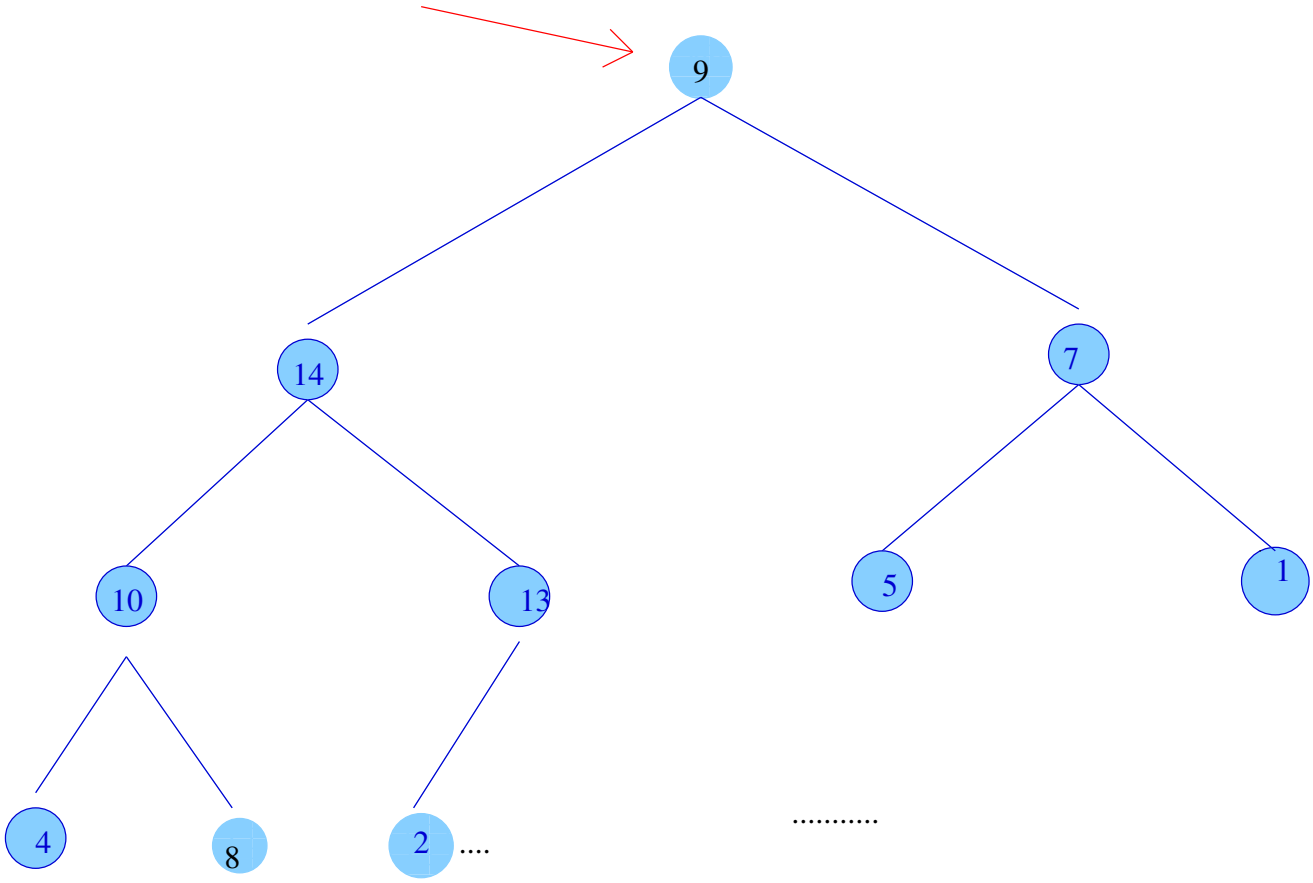
Le fils gauche du noeud qui figure à l'indice i, s'il existe, se trouve à l'indice Fils $G(i) = 2 \times i + 1$, et son fils droit, s'il existe, se trouve à l'indice Fils$D(i) = 2 \times i + 2$. Inversement, le père du noeud d'indice i non nul se trouve à l'indice $[\frac{i-1}{2}]$

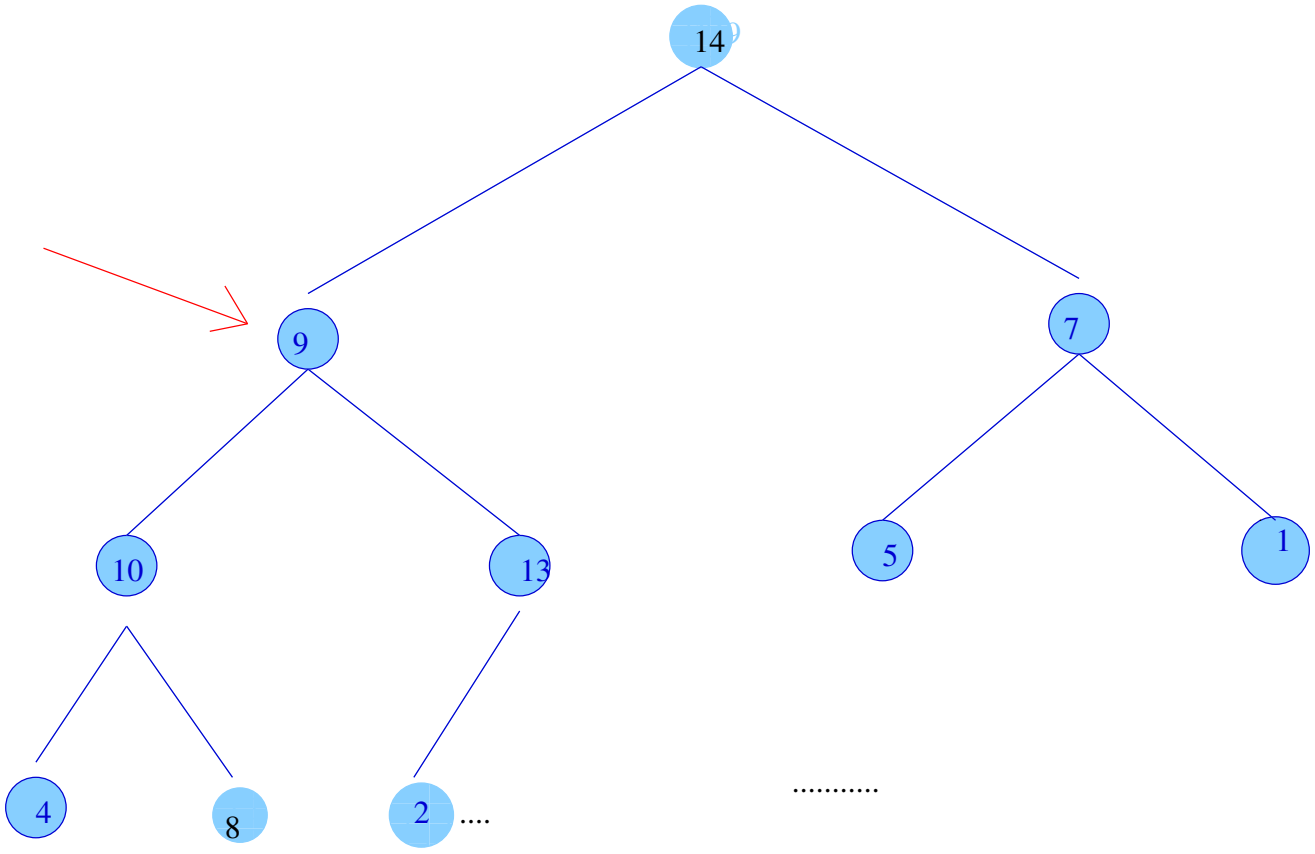


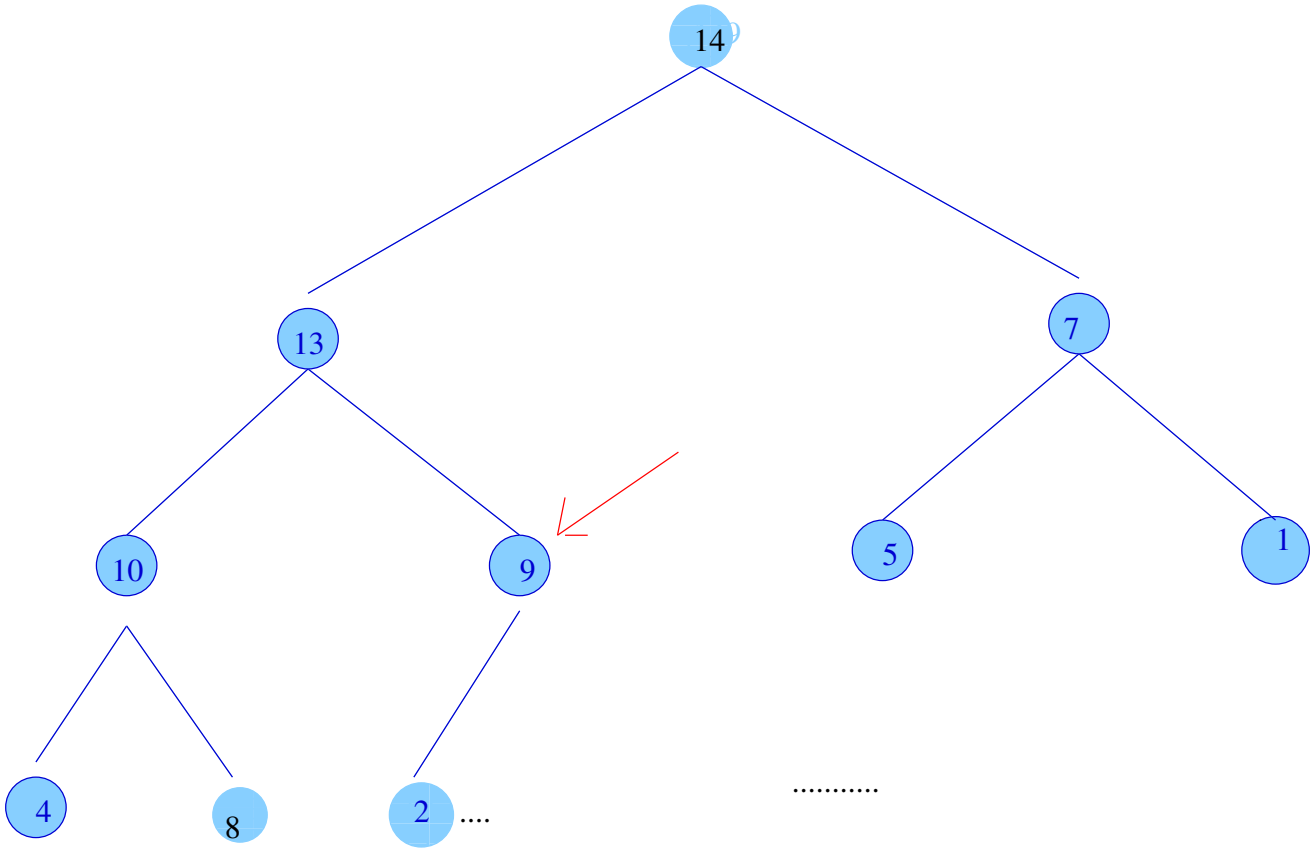
Cette disposition entraîne que l'arbre est forcément équilibré (i.e. toutes ses branches ont la même longueur à un élément près), et les plus longues branches sont à gauche. La hauteur d'un tas, i.e. son nombre de niveaux, contenant n éléments est donc $[log_2 n] + 1$ (le nombre de fois que l'on peut 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 $[\frac{n}{2} - 1]$ C'est le dernier noeud qui a au moins un fils. Les feuilles de l'arbre se trouvent donc entre la position $\frac{n}{2}$ 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.


9
14
7
10
13
5
1
4
8
2
....
.....


14
9
7
10
13
5
1
4
8
2
....
.....


14
13
7
10
9
5
1
4
8
2
....
.....

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 (Fils$G(i) < n$) ET (T[Fils$G(i)] > T[iMax]$) alors
$iMax :=$ Fils$G(i)$
si (Fils$D(i) < n$) ET (T[Fils$D(i)] > T[iMax]$) alors
$iMax :=$ Fils$D(i)$
si ($iMax = i$) alors
Echanger $T[i]$ et $T[iMax]$
Entasser$(iMax, T, n)$

La fonction procède de la façon suivante : si l'on n'a pas atteint une feuille ou que la valeur du noeud courant d'indice i viole la propriété de supériorité des pères sur leurs fils, on échange cette valeur avec la plus grande des valeurs des fils. De cette faÂ¸con le noeud d'indice i aura une valeur plus grande que les valeurs de ses fils. On réitère l'opération tant que la propriété de tas n'est pas rétablie.

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 max$, T, n-1$
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 commenÂ¸cant 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 $(n/2) - 1$. 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 $i := n/2 - 1$à 0 faire
Entasser(i, T, n)

finpour
retourner T
fin

Complexité de la construction. De façon évident la complexité est au plus $O(n \ln n)$. En effet la hauteur du tas est $\ln n$ et on effectue $n/2$ fois l'opération Entasser. En fait le coût de la construction est $O(n)$. 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. Considérons pour simplifier un tas complet, c'est-à-dire dans lequel toutes les branches ont la même longueur. On dénombre $\frac{n}{2}$ noeuds à la hauteur 0 (les feuilles), $\frac{n}{2_2}$noeuds à la hauteur1, $\frac{n}{2^3}$noeuds à la hauteur2, $\cdots$,
On a donc $\frac{n}{2^{h+1}}$ noeuds à la hauteur h, pour chacun desquels on effectuera au plus h échanges dans l'opération Entasser. Le coût de la construction du tas est donc borné par

$$\sum_{h=1}^{\ln n} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor \times h \leq n \times \sum_{h=1}^{\ln n} \frac{h}{2^h}$$

Puisque

$$\sum_{i=0}^{\infty} i \times x^i = \frac{x}{(1-x)^2}$$

on a

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

et donc le nombre d'échanges est borné par

$$n \times \sum_{h=0}^{\ln n} \frac{h}{2^h} \leq 2 \times n = \mathrm{O}(n)$$

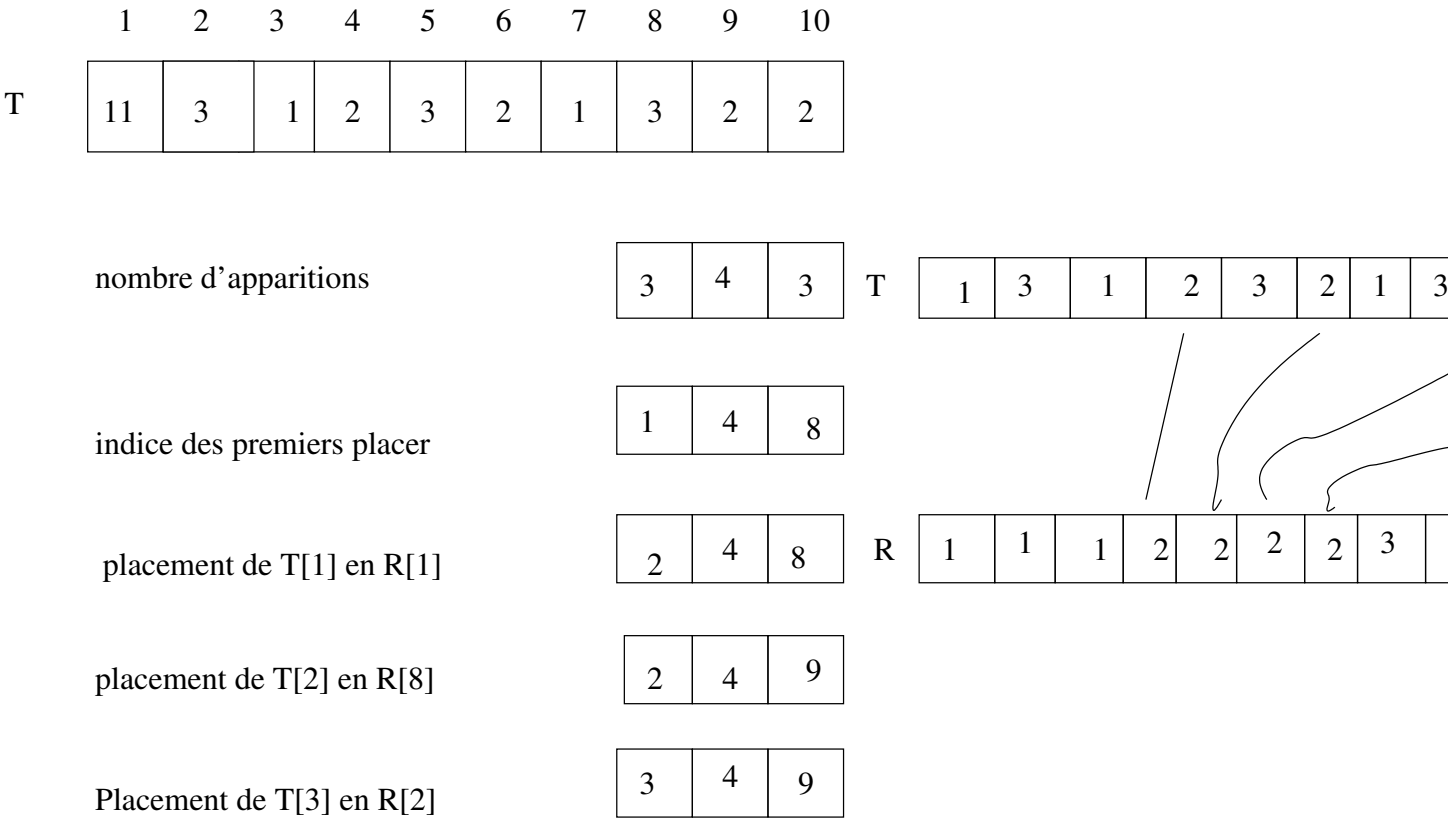
Tri par tas. On construit un tas. On utilise le fait que le premier élément du tas est le plus grand : autant de fois qu'il y a d'éléments, on extrait le premier du tas et on reconstruit le tas avec les éléments restants. Enlever le premier élément consiste simplement à l'échanger avec le dernier du tas et à décrémenter la taille du tas. On rétablit la propriété de tas en appliquant l'opération Entasser sur ce premier élément.
Fonction TriParTas(T, n)
entrée : T est un tableau de n él éments.
sortie : Trie le tableau T par ordre croissant.
début
    ConstruireUnTas(T, n)
    pour $i := n - 1$ à 1 faire
        Echanger($T, 0, i$)
        Entasser($0, T, i$)
    finpour
    retourner T
fin
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 logn. La complexité du tri par tas est donc $O(n \ln n)$.

14.5.1 Tri par dénombrement, tri par base.

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 élément 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 propri été 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 faÂ¸con 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 él éments.
sortie : Rcontient les éléments de T triés par ordre croissant.
début
    /* Initialisations */
    pour $i := 1$ à $k$faire
        $Nb[i] := 0$
    finpour
    /* Calcul des nombres d'apparitions */
    pour $i := 1$ à n faire
        $Nb[T[i]] := Nb[T[i]] + 1$
    finpour
    /* Calcul des indices du premier */

$Nb[k] := n - Nb[k] + 1$
/* Élément de chaque catégorie */
pour $i := k - 1$ à 1 faire
    $Nb[i] := Nb[i + 1] - Nb[i]$
finpour
/* Recopie des éléments originaux du tableau T dans R */
pour $i := 1$ à n faire
    $R[Nb[T[i]]] := T[i]$
    $Nb[T[i]] := Nb[T[i]] + 1$
finpour
retourner R
fin

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).

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 apparaitront dans l'ordre croissant de leurs chiffres des dizaines, et si certaines ont le même chiffre des dizaines alors elles apparaitront 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.

14.5.2 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 1 *La suite $E = (e_1, ..., e_n)$ 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 entres eux le plus souvent et le plus tôt possible. Dans la procédure ci-dessous la suite des valeurs de h est $:\cdots, 1093, 364, 121, 40, 13, 4, 1$.
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 (conjecturesO($n\times (\ln n)^2$) ou $n^{1,25}$).
Algorithme TriShell: 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 ≤ n/9 faire
        h := 3h + 1
    fintq
    tant que h > 0faire
        /* Tri par insertion pour h-ordonner T */
        pour i := h + 1 à n faire
            j := i, v := ei
            tant que ((j > h) et (v < ej − h)) faire
                ej := ej − h, j := j − h
            fintq
            ej := v
        finpour
        h := h/3
    fintq
```

fin
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 faÂ¸con à ce que cette ultime exécution du tri par insertion soit très peu couteuse.

```
def quicksort(T, bg = 0, bd = None) :
    if bd == None :bd = len(T) − 1
    if bg < bd :
    indp = indicepivot(T, bg, bd)
    quicksort(T, bg, indp − 1)
    quicksort(T, indp + 1, bd)
    Pivotage
def indicepivot(T, bg, bd) :
    p = T[bg]
    l = bg + 1
    r = bd
    while (l <= r) :
    while (l <= bd) and (T[l] <= p) :
        l+ = 1
    while (T[r] > p) :
        r− = 1
    if (l < r) :
        T[r], T[l] = T[l], T[r]
    l, r = l + 1, r − 1
    T[r], T[bg] = p, T[r]
return r
```