Projet d'Informatique S5
les algorithmes de tri

E. Albers

Cet atelier sur les tris est à effectuer en autonomie par groupes de 4 ou 5 étudiant(e)s. Il a pour objectif de découvrir les principaux algorithmes de tris de valeurs. Cet atelier permet de sensibiliser à la prise en compte des performances des différents algorithmes (on parle de *la complexité des algorithmes*, voir en section 4 la notation de Landau). La difficulté des parties est précisée par un nombre d'étoiles.

Table des matières

1 La recherche de valeurs

1.1 Rechercher une valeur maximum (*)

On voudrait trouver un algorithme pour **déterminer la maximum d'un ensemble de valeurs**. On part de l'hypothèse que ces valeurs sont stockées dans un tableau et que toutes les valeurs sont des entiers positifs ou nuls.

1. Dans un premier temps, mettez-vous en groupe et écrivez sur papier les étapes élémentaires que vous avez utilisées spontanément pour déterminer le maximum d'un ensemble de valeurs. Pour cela, employez des phrases simples sans formalisme particulier.
2. A partir de ces étapes, écrire sur papier un algorithme en pseudo langage le plus complet possible. Cet algorithme cherchera une valeur donnée dans un tableau d'entiers.
   Si votre algorithme contient des boucles, discutez entre vous le choix de chaque boucle que vous utilisez (*pour*, *tant que* ou *répéter tant que*).
3. Ouvrez le fichier projet. Traduisez votre algorithme en Phyton en complétant la méthode $rechercheValeurMaximum()$ qui se trouve dans la classe *Recherche*. Cette méthode prend en paramètre un tableau quelconque d'entiers et devra renvoyer la valeur maximale trouvée.
4. Testez votre code avec $TestRechercheValeurMaximum()$.
   Ce test utilise la méthode $remplirTableauValeurAleatoire()$ qui renvoie un tableau d'entiers de valeurs aléatoires dont la valeur maximale possible est passée en paramètre.
5. Discutez entre vous de ce qu'il faut modifier dans le code pour trouver la valeur minimale du tableau.

1.2 Rechercher une valeur de manière itérative (*)

On voudrait maintenant déterminer si une valeur donnée est présente ou non dans un tableau. Dans un premier temps, nous allons utliser la manière itérative, c'est à dire que toutes les valeurs du tableau vont être comparées une à une avec la valeur recherchée dans un ordre donné.

1. Comme dans l'exercice précédent, écrire en groupe et sur papier les étapes élémentaires que vous utilisez spontanément pour rechercher une valeur donnée dans un ensemble de valeurs.
2. Ecrire en groupe et sur papier l'algorithme de recherche itérative. Jusitifiez une nouvelle fois l'emploi de la boucle utilisée.
3. Traduisez votre algorithme en Python en complétant la méthode rechercheValeurIterative() qui se trouve dans la classe *Recherche*. Cette méthode prend en paramètre un tableau quelconque d'entiers et la valeur recherchée. Elle renverra un booléen pour indiquer la présence ou non de cette valeur.
4. Testez votre code avec $TestRechercheValeurIterative()$.

1.3 Rechercher une valeur de manière dichotomique (***)

Lorsque les données d'un ensemble sont triés, il est possible de chercher une valeur de manière dichotomique. Cette recherche est semblable à celle utilisée par exemple à la recherche que l'on cherche un mot dans un dictionnaire. Le fait que les mots soient rangés par ordre alphabétique peut faire gagner du temps. L'idée est de couper l'espace de recherche en temps en deux.

1. Ecrire en groupe et sur papier les étapes élémentaires que vous utilisez spontanément pour rechercher un mot dans un dictionnaire.
2. Ecrire ensuite l'algorithme de recherche dichotomique d'une valeur dans un tableau trié par ordre croissant.

> *Indication :* on cherchera à avoir deux indices $indiceMax$ et $indiceMin$ qui permettront de chercher dans une partie du tableau. On choisira au hasard un indice (compris entre $indiceMin$ et $indiceMax$) pour chercher l'élément. Ces deux derniers indices changeront en fonction de la recherche.

Jusitifez l'utilisation du la boucle choisie pour l'algorithme. En particulier, on cherchera à déterminer quand s'arrête la recherche.

3. Traduisez votre algorithme en Python en complétant la méthode rechercheValeurDichotomique() qui se trouve dans la classe Recherche. Cette méthode prend en paramètre un tableau trié d'entiers et la valeur recherchée. Elle renverra un booléen pour indiquer la présence ou non de cette valeur.
4. Testez votre code avec $TestRechercheValeurDichotomique()$.

1.4 Comparaison de performances entre les versions itérative et dichotomique (*)

On cherche dans cette partie à déterminer les performances des deux algorithmes de recherche itérative et de recherche dichotomique.

Pour cela, on cherche à déterminer le nombre de boucles effectuées pour les deux types de recherches.

1. Modifiez le code des méthodes des deux questions précédentes pour qu'elles comptabilisent le nombre boucles effecutées. Avant de quitter ces méthodes, ajoutez ce nombre de boucles aux variables de classe *compteurIteration* ou *compteurDichotomique* qui se trouvent dans la classe $TestComparaisonIteratifDichotomique$, comme ceci :

   ```
   TestComparisonIteratifDichotomique.compteurDichotomique += compteurDeBoucles;
   ```

2. Lancez le test $TestComparaisonIteratifDichotomique()$ qui effectuent 50 recherches consécutives.
3. Qu'en déduisez-vous ?

2 Implémentation du tri à bulle (**)

Le tri à bulle repose sur le principe de remonter les plus grandes valeurs (ou les plus petites) à la fin du tableau, un peu comme une bulle d'air remontrait à la surface. L'idée est de comparer deux à deux les valeurs et de les intervertir pour garder la plus grande valeur pour la prochaine comparaison.

La figure 1 présente le fonctionnement du tri à bulle sur un tableau de 6 valeurs.

1. À partir de cet exemple, écrivez sur papier le principe de l'algorithme du tri à bulle.
   On peut identifer deux boucles dans cet algorithme. À quelle phase correspondent-elles ?
2. Traduisez l'algorithme en Java en complétant la méthode $triBulle()$ qui se trouve dans la classe Tri. Cette méthode prend en paramètre un tableau d'entiers. Les valeurs du tableau doivent être triées par ordre croissant par la méthode.
3. Testez votre code avec le test $TestTriBulle()$.

Dans le pire des cas, il y a pour chaque itération (première boucle) $n-1$ comparaisons et $n-1$ échanges. La complexité est donc en $O(n^2)$ (voir la partie 4).

Si en moyenne la complexité reste en $O(n^2)$, la complexité descend en $O(n)$ dans le meilleur des cas (en optimissant l'algorithme). Le meilleur des cas survient lorsque le tableau est déjà trié.

On peut donc dire que l'utilisation de ce tri à bulle est intéressant uniquement lorsque les tableaux sont presque triés. La complexité est proche alors de la complexité linéaire.

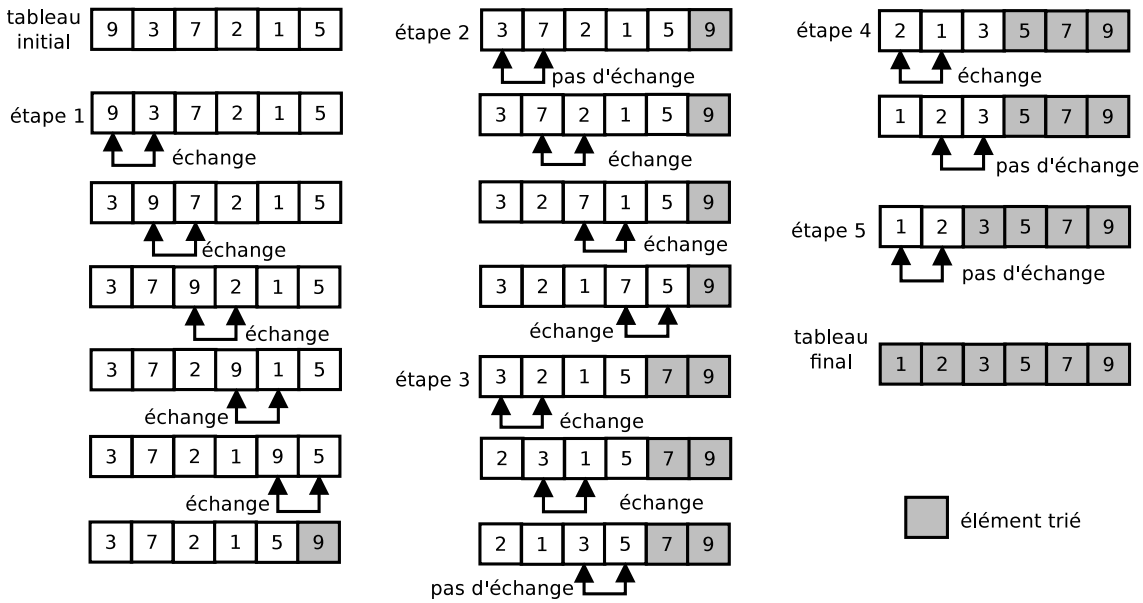

FIGURE 1 – Exemple de fonctionnement du tri à bulle

3 Implémentations de deux autres tris au choix

Le but de cette partie est d'implémenter deux autres algorithmes de tri. Vous choisirez **deux tris** à implémenter parmi les tris suivants :

— le tri par insertion (*),
— le tri par sélection (*),
— le tri par *shell* (**),
— le tri rapide (***),
— le tri par comptage (**),
— et le tri par base (**).

1. Implémentez deux tris que vous avez choisis.
2. Testez les en lançant $TestTri2$ et $TestTri3$.
3. Lancez le $TestUltime$ afin de récupérer les temps d'exécution des trois algorithmes. Copiez les différents temps dans un tableur afin d'obtenir un graphique.
4. Faites un document afin de commenter les résultats et d'insérer le graphique. Vous pourrez le déposer ensuite sur le campus.

3.1 Tri par insertion (*)

Le tri par insertion repose sur le principe simple suivant : l'idée est de prendre les éléments un par un et de les classer au fur et à mesure.

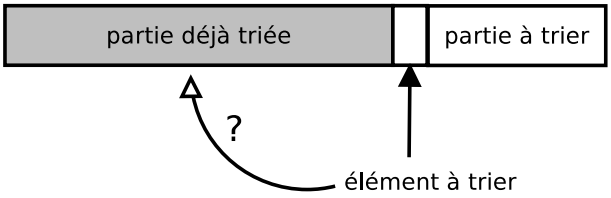

FIGURE 2 – Principe du tri par insertion

Au début de l'algorithme, la partie déjà triée est composée uniquement du premier élément, la partie à trier étant constituée du reste des éléments. Voici le principe de cet algorithme :

pour tous les éléments un à un (sauf le premier) **faire**
on cherche l'élément qui est plus petit que l'élément à trier et on décale les valeurs au fur et à mesure.

Voici le code de l'algorithme en pseudo langage :

```
triParInsertion(tableau d'entiers)
  elementAtrier : entier
  place: place;
début
  pour i allant de 1 à nbElements-1 faire
    // on sauvegarde l'élément à trier:
    elementAtrier <-  tableau[i];
    place <- i-1;
    // on cherche sa place dans la partie triée tout en décalant:
    tant que place>0 et tab[place]>elementAtrier faire
      tab[place+1] <- tab[place]
      place <- place -1
    finTq
    // on place l'élément à trier à sa place:
    tab[place+1] <- elementAtrier
  finPour
fin
```

Le tri par insertion est de complexité en $O(n^2)$ en moyenne. Dans le meilleur cas où le tableau est déjà trié, la complexité devient en $O(n)$. Le pire survient lorque le tableau est trié dans l'ordre inverse.

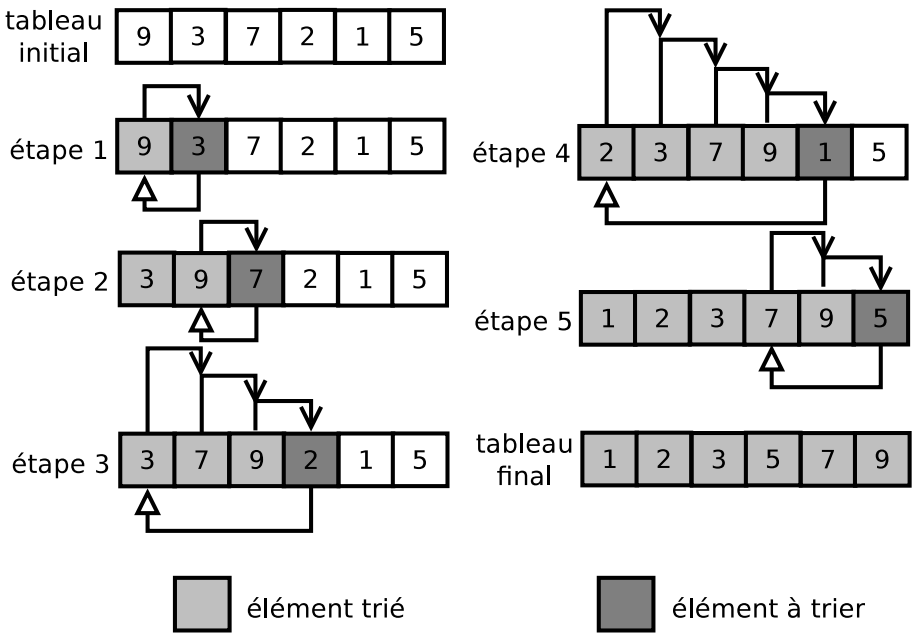

FIGURE 3 – Exemple de fonctionnement du tri par insertion

La figure 3 présente un exemple de simulation de l'algorithme. Détaillons maintenant l'algorithme :

étape 1 : l'élément à trier est la valeur 3, et le tableau trié correspond au singleton | 9 | . 3 étant inférieur à 9, on entre dans le *tant que* et on intervertit ces deux valeurs. La variable *place* valant 0, le *tant que* s'arrête avec comme tableau trié | 3 | 9 | . Les autres valeurs du tableau (partie non triée) n'ont pas bougé : | 3 | 9 | 7 | 2 | 1 | 5 | .

étape 2 : l'élément à trier est la valeur 7. 7 étant inférieur à 9, la condition de la boucle *tant que* est vérifiée ; on intervertit alors ces deux valeurs, ce qui donne le tableau | 3 | 7 | 9 | .
7 n'étant pas inférieur 9, on sort du *tant que*. Le tableau trié est donc : | 3 | 7 | 9 | .

étape 3 : l'élément à trier est la valeur 2. 2 étant inférieur à 9, on intervertit 2 et 9, puis 2 et 7 et enfin 2 et 3 pour arriver finalement au tableau trié suivant : | 2 | 3 | 7 | 9 | .

étape 4 : l'élément à trier est la valeur 1. De même qu'à l'étape 3, 1 est inférieur à toutes les valeurs présentes dans le tableau trié. On intervertit donc 2 avec toutes les valeurs pour arriver au tableau trié suivant : | 1 | 2 | 3 | 7 | 9 | .

étape 4 : dans cette dernière étape, 5 est la valeur à trier. 5 étant respectivement inférieur à 9 et à 7, on arrive au tableau : | 1 | 2 | 3 | 5 | 7 | 9 | . 5 n'étant pas inférieur à 3, la boucle *tant que* s'arrête.
Le tableau est donc effectivement bien trié : | 1 | 2 | 3 | 5 | 7 | 9 | .

3.2 Tri par sélection (*)

Le principe du tri à bulle est de chercher l'élément le plus petit dans le tableau et de le placer à l'indice 0 ; on cherche ensuite l'élément le plus petit dans les éléments restant et on le place à l'indice 1, *etc.* L'algorithme continue jusqu'à ce que l'avant dernier élément restant soit placé.

Voici le principe de l'algorithme :

pour i allant de 0 à $taille - 1$ **faire**
on cherche l'indice j du plus petit élément du tableau à trier avec $i + 1 \leq j \leq taille$,
on échange les valeurs $tab[i]$ et $tab[j]$.
fin pour

La recherche du plus petit élément dans le tableau à trier se traduit par une boucle *tant que*. Dans tous les cas, il y a donc $n * ((n-1)/2)$ comparaisons ; la complexité de cet algorithme est par conséquent en $O(n^2)$.

Mais contrairement au tri par insertion, il y a peu d'échanges de valeur, $n - 1$ fois dans le pire cas. Ce tri est donc adapté lorsque les éléments sont difficiles à déplacer (accès disque coûteux en temps par exemple).

Voici une implémentation possible de cet algorithme en pseudo langage :

```
triParSelection(tableau d'entiers)
  min : entier
début
  pour i allant de 0 à taille-1 faire
    min <- i
    pour j allant de i+1 à taille-1 faire
      si tableau[j]<tableau[min] alors
         min <- j
      finSi
    finPour
    si min <> i alors
      échanger les valeurs tableau[i] et tableau[min]
    finSi
  finPour
fin
```

On peut remarquer que le code ci-dessus traite le cas supplémentaire où l'élément le plus petit se trouve à l'indice i (cas traité par le deuxième *si*). En effet, il n'y a pas à échanger de valeurs dans ce cas.

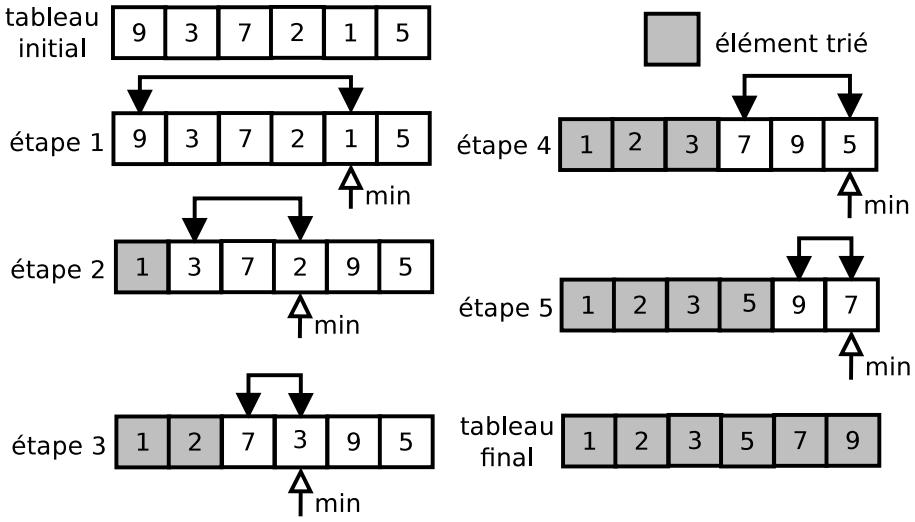

FIGURE 4 – Exemple de fonctionnement du tri par sélection

La figure 4 présente un exemple de simulation de l'algorithme du tri par sélection. Détaillons maintenant l'algorithme étape par étape :

étape 1 : on parcourt le tableau pour trouver l'élément le plus petit. Il s'agit de la valeur 1 qu'il faut échanger avec la première case du tableau. Les valeurs 1 et 9 sont donc échangées.

étape 2 : on parcourt le tableau pour trouver l'élément le plus petit, excepté la première case du tableau. La valeur 2 est trouvée et échangée avec la valeur 3.

étape 3 : La valeur 3 est la valeur la plus petite dans les éléments du tableau restant. Cette valeur est échangée avec la valeur 7.

étape 4 : La valeur 5 est la valeur la plus petite dans les éléments du tableau restant, qui est échangée avec la valeur 7.

étape 5 : La valeur 7 est la plus petite des valeurs restantes. La valeur 7 est échangée avec la valeur 9. Le tableau résultant est bien trié par ordre croissant.

3.3 Tri *shell* (**)

Le tri *shell* est une amélioration du tri par insertion. On part du constat que le tri par insertion est efficace lorsque les éléments sont quasi triés, et que son inefficacité en moyenne provient qu'il déplace les éléments un à un.

L'idée est d'appliquer l'algorithme du tri par insertion en plusieurs fois en ne regardant que les éléments dont les indices sont multiples d'une valeur n que l'on nomme : **pas**. On part d'un pas assez grand et on le diminue au fur et à mesure. On peut intuitivement voir le tri *shell* comme plusieurs applications successives et de plus en plus fines du tri par insertion. Le pas 1 revient à appliquer directement le tri par insertion. Ceci dit, comme plusieurs passes ont été déjà réalisées, on peut penser que les valeurs ont déjà été triées grossièrement, car le tri par insertion reste efficace lorsque le tableau est quasi trié.

Pour des raisons d'efficacité, les valeurs successives des pas ne doivent pas être multiples l'une de l'autre. De manière empirique, les valeurs qui sont les plus appropriées sont les suivantes : $1, 4, 10, 23, 57, 132, 301, 701, 1750$.

Dans l'algorithme suivant, on prend comme valeurs successives de n, les termes de la suite suivante :

$$\begin{cases} u_0 = 0; \\ u_{n+1} = 3u_n + 1 \end{cases}$$

```
triShell(tableau d'entiers)
   n : entier
   valeur : entier
   j : entier
début
   n <- 0
   tant que n<longueur faire
       n== <-3*n+1;
   finTq
   tant que n <>1 faire
       n <- n/3;
       pour i allant de n à la taille du tableau -1 faire
             valeur <-tableau[i]
             j <- i
             tant que  j>n-1 et tableau[j-n]>valeur faire
                 tableau[j] <- tableau[j-n]
                 j <- j-n;
             finTq
             tableau[j] <-valeur;
       finPour
   finTq
fin
```

La figure 5 présente un exemple d'application du tri *shell*. Le premier pas pris est 4 car 13 dépasse la longueur du tableau.

étape 1 : On regarde tout d'abord tous les éléments dont les indices sont multiples de 4, soient les valeurs 9 et 1.

L'élément à insérer est 1. Comme il inférieur à 9, les deux valeurs sont échangées.

À noter que les valeurs 3, 7, 2 et 5 ne sont pas considérées dans cette première étape. Elles ne changent pas donc de places.

étape 2 : Le pas est de 1, le tri par insertion s'applique donc sur la totalité des valeurs.

Le tri *shell* devient intéressant lorsque le nombre de valeurs des tableaux devient important. Il est évident que pour un petit nombre de valeurs, le tri par insertion simple est plus efficace.

3.4 Tri rapide (***)

Le tri rapide (*quick sort*) est l'un des tris les plus efficaces. Il repose sur le principe de la dichotomie. L'idée est de séparer le tableau en deux en choisant une valeur que l'on nomme **pivot**. Le choix du pivot est crucial pour l'efficacité de l'algorithme ; mais comme en pratique sa recherche prend trop de temps, on peut le choisir de manière aléatoire (le premier, le dernier, celui qui se trouve au milieu, *etc.*

Une fois que le pivot est choisi, on parcourt les valeurs du tableau. Pour un tri dans l'ordre croissant, on déplace les valeurs qui sont inférieures au pivot. Une fois que l'on a parcouru toutes les valeurs, toutes les valeurs inférieures sont placées à gauche du tableau et celles qui sont supérieures à sa droite. Il suffit de compter le nombre de décalages effectués pour ensuite déplacer la valeur du pivot.

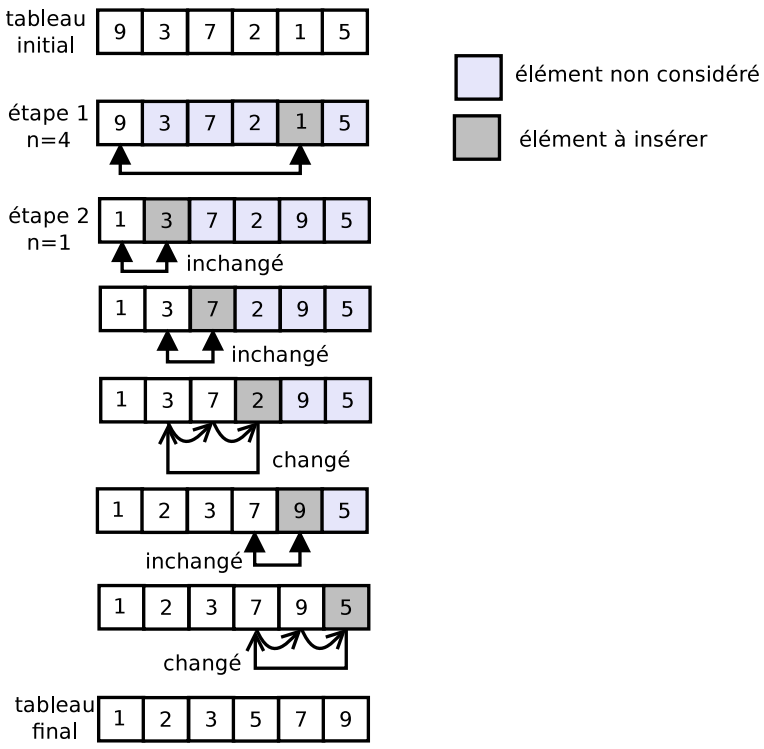

FIGURE 5 – Exemple de fonctionnement du tri *shell*

De cette manière, l'algorithme a scindé le tableau en deux. À gauche du pivot, toutes les valeurs lui sont inféreures et à droite, toutes les valeurs lui sont supérieures.

Il suffit d'appeler récursivement l'algorithme pour trier les deux parties des valeurs qui sont respectivement supérieures et inférieures au pivot. La récursivité s'arrête une fois qu'il n'y a plus qu'un seul élément à trier ou qu'il soit vide. La figure 6 présente un exemple de tri. Dans cet exemple, la première valeur du tableau est systématiquement prise comme pivot.

étape 1 : On traite le tableau | 9 | 3 | 7 | 2 | 1 | 5 | .

9 est choisi comme pivot. On va comparer toutes les autres valeurs à celle-ci. L'idée est de compter le nombre de valeurs qui seront inférieures à 9.
3 est inférieur à 9, il faut décaler 3. Comme le nombre de valeurs inférieur vaut 0, 3 ne change pas de place.
7 est inférieur à 9. Le nombre de valeurs vaut 1, 7 ne change pas non plus de place.
Ainsi de suite, jusqu'à la valeur 5.

Il faut maintenant échanger la valeur du pivot avec la dernière valeur inférieure. Ainsi, les valeurs 9 et 5 s'intervertissent.

Deux nouveaux appels sont effectués. Le premier avec le tableau | 5 | 3 | 7 | 2 | 1 |, et le second avec un tableau vide. L'appel au tableau vide n'a aucun effet sur le tableau.

étape 2 : On traite le tableau | 5 | 3 | 7 | 2 | 1 | .

5 est choisi comme pivot. La valeur 3 doit être décalée mais reste à la même place. La valeur 7 ne bouge pas, et le compteur de valeurs inférieures n'est pas modifié.
La valeur 2 doit être décalé à la place de 7. De même, la valeur 1 doit être décalé à la place de 7.
Enfin, la valeur du pivot doit être échangée avec la dernière valeur inférieure qui est 1.

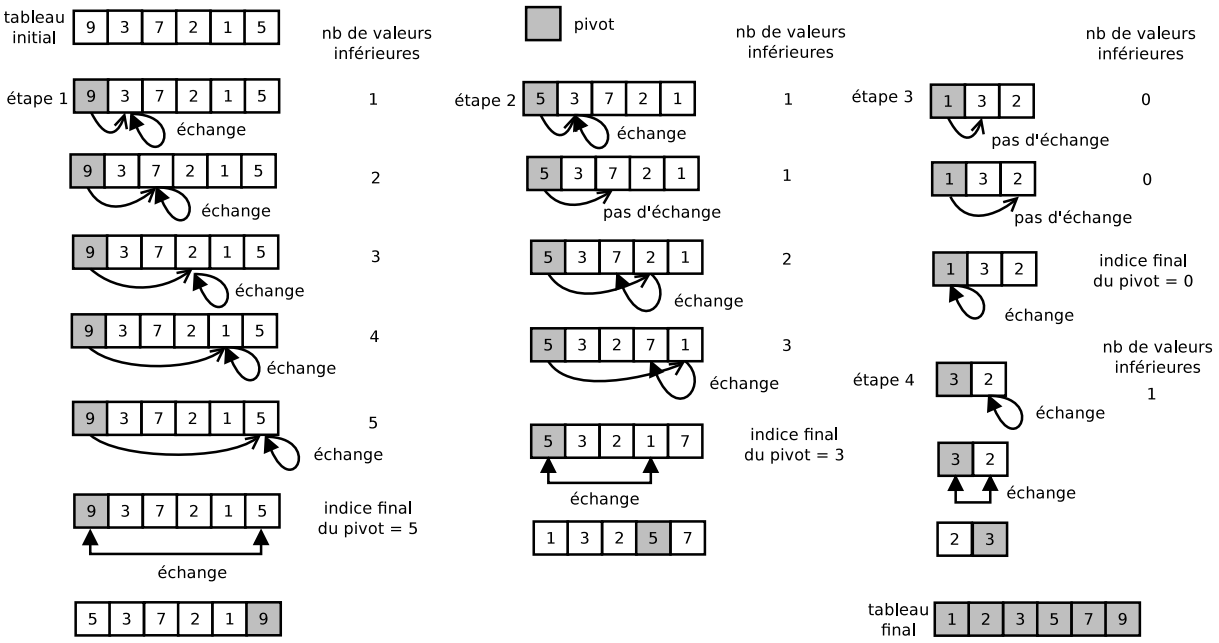

FIGURE 6 – Exemple de fonctionnement du tri rapide

Deux nouveaux appels sont effectués. Le premier avec le tableau | 1 | 3 | 2 |, et le second avec le tableau | 1 | .

Le second appel ne modifie pas le tableau.

étape 3 : On traite le tableau | 1 | 3 | 2 | . La valeur 1 est choisie comme pivot.

Les valeurs 3 et 2 reste sur place puisqu'elles sont supérieures à 1. La valeur du pivot n'est donc pas non plus décalée.

Deux nouveaux appels sont effectués. Le premier avec un tableau vide et le second avec le tableau | 3 | 2 | .

étape 4 : On traite le tableau | 3 | 2 | . 2 est choisi comme pivot.

La valeur 3 doit être décalée mais reste à sa place. La valeur du pivot doit être échangée avec la valeur 3.

Deux nouveaux appels sont effectués. Le premier avec le tableau | 2 | qui ne modifie pas le tableau et le second avec un tableau vide.

La figure 7 résume les différents appels récursifs qui sont effectués dans cet exemple.

Voici ci-dessous une implémentation en pseudo langage de ce tri rapide. La procédure *echangerValeurTableau* n'a pour but que d'échanger deux valeurs du tableau. La procédure *partition* renvoie quant à elle l'indice de la place du pivot dans le tableau, et modifie le tableau de telle sorte que toutes les valeurs entre les indices $[deb, pivot[$ seront inférieures à la valeur du pivot, et toutes les valeurs entre les indices $]pivot, fin]$ seront supérieures à la valeur du pivot.

La procédure *triRapide*2 cherche tout d'abord le pivot et s'appelle récursivement avec comme paramètre les deux sous tableaux. La valeur du pivot étant correctement positionnée, elle n'est plus considérée.

```
triRapide(tablean d'entiers)
début
  triRapide2(tableau,0,taille-1)
fin
```

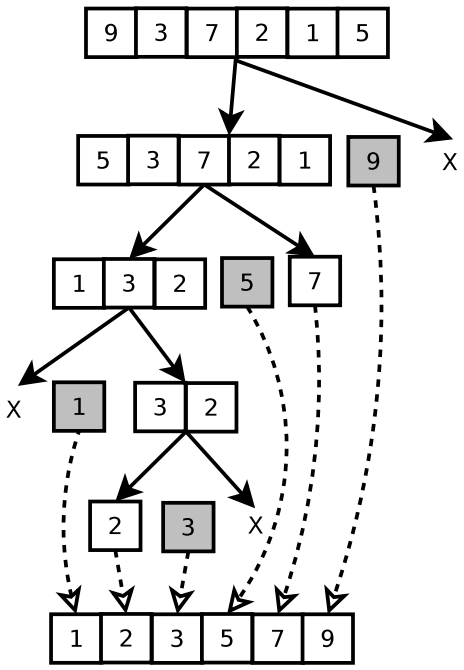

FIGURE 7 – Différents appels récursifs de l'exemple de la figure 6

```
triRapide2(tableau d'entiers, debut:entier, fin: entier)
    pivot : entier
début
  si debut<fin alors
    pivot <- partition(tableau,debut,fin)
    triRapide2(tableau,debut,pivot-1)
    triRapide2(tableau,pivot+1,fin)
  finSi
fin

partition(tableau d'entiers, deb: entier, fin; entier){
  compt : entier
  pivot : entier
début
  compt <- deb
  pivot <- tableau[deb]

  pour i allant de deb+1 à fin faire
    si tableau[i]<pivot faire
      compt <- compt+1
      echangerValeurTableau(tableau,compt,i);
    finSi
  finPour
  echangerValeurTableau(tableau,compt,deb)
  retourner compt
fin
```

3.5 Tri par tas (****)

Le tri par tas (*heapsort*) repose sur le fait que le tableau est vu comme un arbre binaire[1]. Le premier élément du tableau étant la racine, le deuxième et le troisième étant les deux fils, *etc.* De manière générale, le $n^{\text{ième}}$ élément du tableau d'indice n aura pour fils les éléments d'indice $2n$ et $2n+1$.

D'autre part, l'arbre binaire a les caractéristiques suivantes :

— toutes les feuilles de l'arbre se trouvent sur deux niveaux maximum (arbre complet),

— les feuilles de longueur maximale sont toutes sur la gauche de l'arbre,

— chaque nœud de l'arbre est supérieur (respectivement inférieur) à ses deux fils pour un ordre croissant (respectivement décroissant).
La racine de l'arbre sera par conséquent la valeur maximale de l'arbre (respectivement minimale).

Tout élément mal placé dans l'arbre devra être **tamisé**. L'opération de tamisage consiste à échanger la racine avec le plus grand de ses fils, et ceci de manière récursive jusqu'à ce que les propriétés ci-dessus de l'arbre soient vérifiées.

Une fois que l'arbre a été tamisé, on retire la valeur maximale et on recommence l'opération sur les éléments restant de l'arbre.

Voici une implémentation en pseudo langage de l'algorithme du tri par tas :

```
tamiser(tableau d'entiers, noeud:entier, n:entier
  k : entier
  j :entier
  continu : booleen
début
  k <- noeud
  j <- 2*noeud
  continu <- true
  tant que continue et j<=n faire
    si j<n et tableau[j]<tableau[j+1] faire
      j <- j+1
    finSi
    si tableau[k]<tableau[j] faire
      echangerValeurTableau(tableau, k, j)
      k <- j
      j <- 2 * k
    sinon
      continu <- false
    finSi
  finTq
fin

triParTas(tableau d'entiers, longueur: entier)
début
  pour i allant de longueur-1 à 0 faire
    tamiser (tableau, i, longueur-1)
  finPour
  pour i allant de longueur-1 à 1  faire
    echangerValeurTableau(tableau, i, 0)
```

1. Le cours sur les arbres sera vu en *S6*.

```
    tamiser(tableau, 0, i-1)
  finPour
fin
```

La figure 8 montre le résultat du premier tamisage. La figure 9 montre les différents échanges de valeur ainsi que les tamisages successifs.

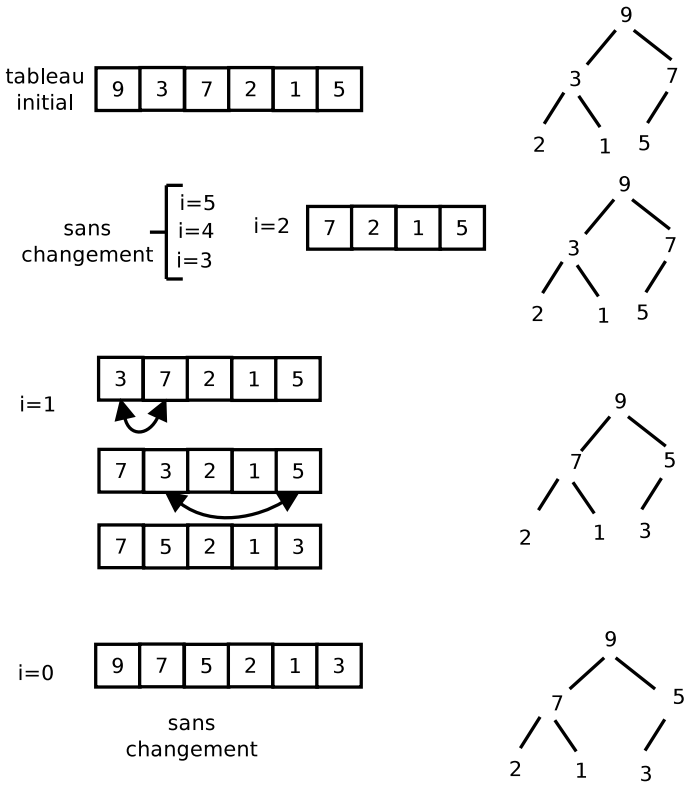

FIGURE 8 – Premier tamisage dans l'algorithme du tri par tas

La complexité du tri pas tas est en $O(n * log_2 n)$, et donc assymptotiquement optimal. D'autre part, ce tri ne nécessite pas d'allocation dynamique supplémentaire, ce qui est un autre avantage. Il reste cependant dans les faits deux fois moins efficace que le tri rapide.

3.6 Le tri par comptage (**)

Cet algorithme ne fonctionne qu'avec des tableaux d'entiers. Il est linéaire mais nécessite l'utilisation d'une seconde liste qui permet de compter le nombre d'éléments du tableau.

Par exemple, on suppose qu'on dispose d'un tableau composé d'entiers entre 0 et 10 (bornes comprises). Le procédé du tri par comptage est le suivant :

— on compte le nombre de 0, puis le nombre de 1, ...et le nombre des 10 présents dans le tableau,
— ce nombre de valeurs est stocké dans un tableau de comptage des éléments,
— on écrase les valeurs du tableau initial à l'aide du tableau de comptage.

Soit par exemple le tableau d'entiers $[1, 7, 3, 1, 3]$ qui contient 2 fois 1, 2 fois 3 et 1 fois 7. Le tableau trié par la méthode du tri comptage est donc : $[1, 1, 3, 3, 7]$.

x	1	2	3	4	5
tableau[x]	1	7	3	1	3
tableau[x] trié	1	1	3	3	7

tableau avant et après le tri

x	0	1	2	3	4	5	6	7	8	9	10
tableauComptage[x]	0	2	0	2	0	0	0	1	0	0	0

tableau de comptage des éléments

Voici l'algorithme de ce tri en pseudo langage :

```
triParComptage(tableau d'entiers, borneSuperieure: entier)
    tabComptage: tableau d'entiers
    l : entier
début
   pour i allant 0 à la taille du tableau faire
      tabComptage[i] <- 0
   finPour

   /* Création du tableau de comptage */
   pour i allant de 0 à la taille de tableau faire
      tabComptage[ tab[ i ] ] <- tabComptage[ tab[ i ] ] + 1
   finPour

   /* Création du tableau trié */
   l <- 0
   pour i allant de 0 à borneJusqu'à borneSuperieure faire
      pour j allant de 1 à tabComptage[i] faire
         tab[l] <- i
         l <- l + 1
      finPour
   finPour
fin
```

3.7 Le tri par base (**)

Le tri par base (ou tri radix) est utilisé pour ordonner des éléments identifiés par une clef unique. Chaque clef est une chaîne de caractères ou un nombre que le tri par base trie selon un ordre lexical.

Le temps d'exécution est $O(n\ k)$ où n est le nombre d'objets et k la taille moyenne des clefs. Cet algorithme était utilisé pour trier des cartes perforées en plusieurs passages.

Le tri par base a été réutilisé comme une alternative à des algorithmes de tri plus puissants (comme les tris rapides, par tas et par fusion) qui demandent $O(n\ log\ n)$ comparaisons où n est le nombre d'objets à trier. Ces algorithmes sont plus efficaces pour trier des données selon un ordre qui n'est pas lexical, mais c'est de peu d'importance pour les applications pratiques.

Si la taille de l'espace possible de clefs est proportionnel au nombre d'éléments, alors chaque clef aura une taille de $log\ n$ caractères et le tri par base s'effectuera en un temps $O(n\ log\ n)$ dans ce cas.

En pratique, si les clefs utilisées sont de petits entiers, le tri peut être réalisé en deux temps, les comparaisons peuvent alors être faites avec quelques opérations qui opèrent en un temps constant. Dans ce cas, le tri par base s'exécutera en $O(n)$ et en pratique il s'avérera plus rapide que d'autres algorithmes de tri.

Le plus grand désavantage du tri par base est qu'il nécessite $O(n)$ espace mémoire supplémentaire et il requiert une analyse de chaque caractère des clefs de la liste d'entrée, il peut donc être très lent pour des clefs longues.

L'ordre de tri est typiquement le suivant : les clefs courtes viennent avant les clefs longues, les clefs de même taille sont triées selon un ordre lexical. Cette méthode correspond à l'ordre naturel des nombres s'ils sont représentés par des chaînes de chiffres.

Son mode opératoire est :

1. prendre le chiffre (ou groupe de bits) le moins significatif de chaque clef,
2. trier la liste des éléments selon ce chiffre, mais conserve l'ordre des éléments ayant le même chiffre.
3. répèter le tri avec chaque chiffre plus significatif.

Voici un exemple d'utilisation. Soit le tableau $[170, 45, 75, 90, 2, 24, 802, 66]$:
— trier par le chiffre le moins significatif (unités) :
$[170, 90, 2, 802, 24, 45, 75, 66]$
— trier par le chiffre suivant (dizaines) :
$[2, 802, 24, 45, 66, 170, 75, 90]$
— trier par le chiffre le plus significatif (centaines) :
$[2, 24, 45, 66, 75, 90, 170, 802]$

4 Notation de Landau

La notation de Landau est souvent utilisée pour classifier les algorithmes par leur complexité. Cette notation se base sur une comparaison asymptotique. On dit que $f(x) \in O(g(x))$, lorsque f est bornée par le dessus par g de manière assymptotique.

Dans le tableau suivant, n correspond au nombre de données sur lequel l'algorithme travail, et c est une constante. Cela peut être par exemple, le nombre d'éléments d'un tableau à trier.

notation	croissance
$O(1)$	contante
$O(log(n))$	logarithmique
$O(log(n)^c)$	polylogarithmique
$O(n)$	linéaire
$O(n * log(n))$	quasi-linéaire
$O(n^2)$	quadratique
$O(n^c)$	polynomiale
$O(c^n)$	exponentielle
$O(n!)$	factorielle

Le tableau suivant présente les différentes valeurs calculées en fonction des complexités. Le nombre d'instructions et en conséquence le temps d'exécution d'un algorithme seront proportionnels à ce nombre.

complexité	valeurs de n							
	5	10	20	50	100	1000	100000	1000000
$O(1)$	1	1	1	1	1	1	1	1
$O(log(n))$	0,7	1	1,3	1,7	2	3	5	6
$O(n)$	5	10	20	50	100	1000	100000	1000000
$O(n * log(n))$	3,5	10	26	85	200	3000	500000	6000000
$O(n^2)$	25	100	400	2500	10000	1.10^6	1.10^{10}	1.10^{12}
$O(n^3)$	125	1000	8000	125000	1.10^6	1.10^9	1.10^{15}	1.10^{18}
$O(n!)$	120	3.10^6	2.10^{18}	3.10^{64}	9.10^{157}	...	...	...

Exemple de calcul de complexité Pour calculer la complexité d'un algorithme, il suffit de compter le nombre d'instructions qui est effectuée. Le comportement lorsque le nombre de données tend vers l'infini détermine la classe de complexité de l'algorithme.

Soit la définition de fonction suivante dont le but est de renvoyer le nombre d'occurrences d'une valeur donnée :

```
int nbOccurrenceValeur(int tab[], int taille, int valeur){
  int nombre, i;
  nombre = 0;
  for(i=0; i< taille; i++)
    if(tab[i] == valeur)
      nombre++;
  return nombre;
}
```

Pour mieux analyser la complexité, transformons la boucle *for* en *while*, ce qui donne le code suivant :

ligne	code	nombre d'instructions
1	$int\ nbOccurrenceValeur(int\ tab[],\ int\ taille,\ int\ valeur)\{$	0
2	$int\ nombre, i;$	0
3	$nombre = 0;$	1
4	$i = 0;$	1
5	$while(i < taille)\{$	$1 \quad * taille$
6	$if(tab[i] == valeur)$	$1 \quad * taille$
7	$nombre++;$	$1 \quad * [0, taille]$
8	$i = i + 1;$	$1 \quad * taille$
9	$\}$	0
10	$return\ nombre;$	0
11	$\}$	0

Les lignes 3 et 4 sont faites une fois. À chaque boucle, les tests des lignes 5, 6 et 8 sont réalisées systématiquement ; comme ces trois lignes sont dans la boucle, elles seront effectivement réalisées *taille* fois.
Seule la ligne 7 n'est faite que si le test de la ligne 6 s'avère vrai. Si la valeur n'est pas présente dans le tableau, la ligne 7 sera faire 0 fois (meilleur des cas). Si le tableau ne contient que la valeur recherchée, alors la ligne 7 sera effectuée *taille* fois (pire des cas).

En conclusion, le nombre d'instructions est compris entre $2 + (4 * taille)$ (pire des cas), et $2 + (3 * taille)$ (meilleur des cas). Ce qui implique que le nombre d'instructions tend vers *taille* lorsque *taille* tend vers l'infini.

La complexité de cet algorithme est par conséquent en $O(n)$, où n correspond à la taille du tableau, *i.e.* son nombre d'éléments.

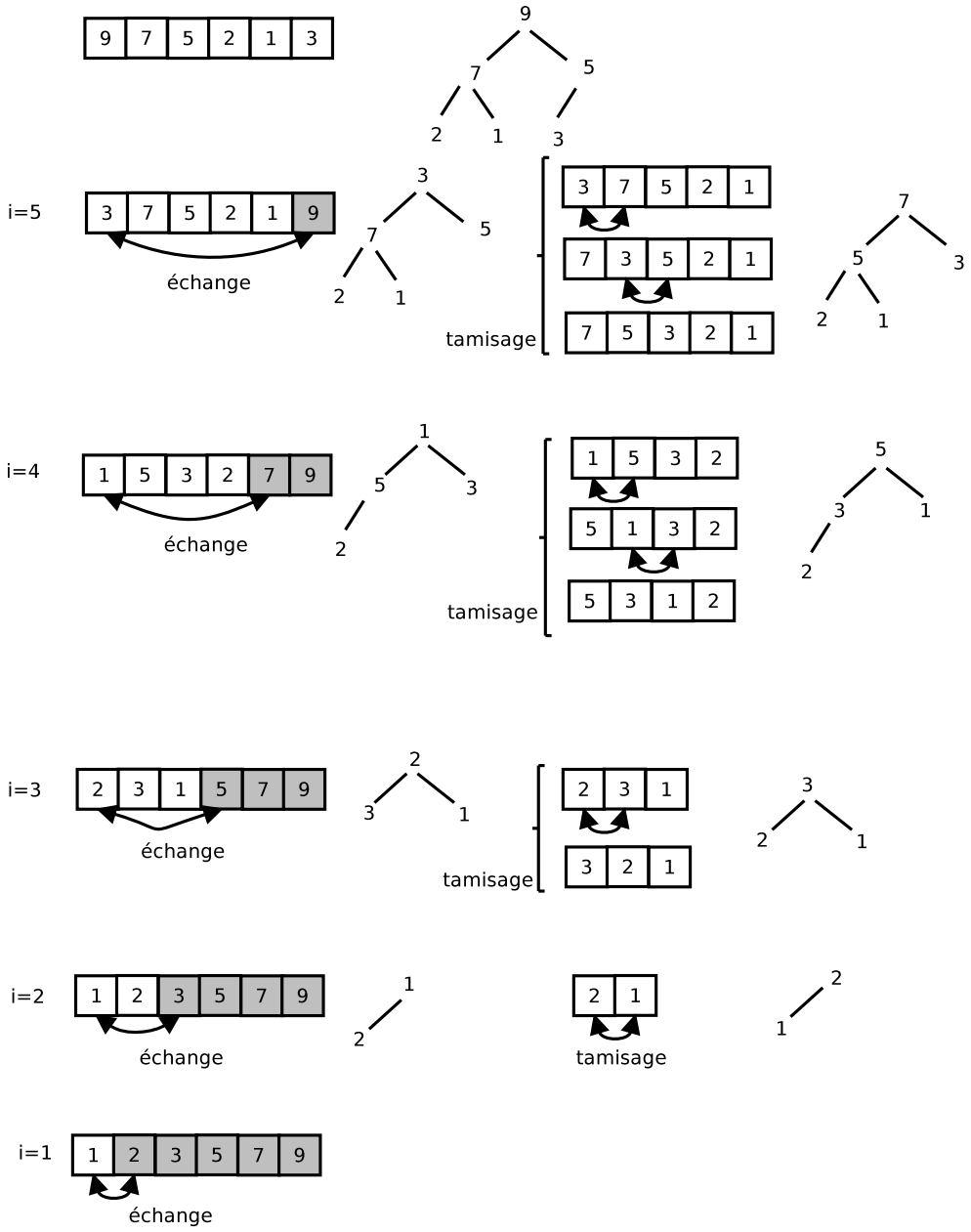

FIGURE 9 – Échange des valeurs et tamisage successifs