Notes de cours d'algorithmique
École de combinatoire et algorithmique de Madagascar

Roberto Mantaci
transcription de Jean-Baptiste Yunès

ESPA, Vontovorona, Septembre 2011

Table des matières

1 Le problème de la recherche d'un élément

Problème : étant donné un tableau A (par exemple d'entiers) trié en ordre croissant et un entier x, déterminer si $x \in A$ (c.a.d. $\exists i \,/\, A[i] = x$).

La recherche dichothomique

L'idée est de découper le tableau en deux parties (à peu près égales) et comparerr x avec l'élément du milieu, de la sorte
- ou bien x est l'élément « milieu » et l'on a donc trouvé la réponse (positive),
- ou bien x est plus petit que l'élément milieu et l'on peut « recommencer » à appliquer la méthode à la seule partie gauche du tableau,
- ou bien x est plus grand que l'élément milieu et c'est avec la partie droite que l'on recommence.

L'algorithme doit s'arrêter (c'est une condition pour être un algorithme), il faut donc décider quand il est nécessaire de terminer. La condition évidente est que si l'on a à faire à un tableau de longueur égale à 1, c'est qu'il n'y a pas de partie gauche, ni droite (seulement un élément milieu) et donc l'algorithme peut terminer en décidant si l'élément x est ou n'est pas dans le tableau A en le comparant avec cet unique élément.

Comme pour beaucoup d'autres algorithmes présentés par la suite, on utilisera deux indices d (pour « début ») et f (pour « fin ») pour désigner la partie du tableau sur laquelle on est en train d'effectuer la recherche (initialement $d = 1$ et $f = n$).

```
Algo Recherche_dichotomique(A:tableau[1..n] d'entiers; x,d,f:entiers) : booléen
DÉBUT
    m:entier;
    si (f<d) alors
        retourner (faux);
    si (d=f) alors
        si (A[d]=x) alors
            retourner (vrai);
        sinon
            retourner (faux);
    sinon
        m := (d+f)/2; (*division entière*)
        si (x<A[m]) alors
            retourner (Recherche_dichotomique(A,x,d,m-1)); (*est-il en partie gauche ?*)
        sinon
            si (x>A[m]) alors
                retourner (Recheche_dichotomique(A,x,m+1,f)); (*est-il à droite ?*)
            sinon
                retourner (vrai); (*il est exactement ici!*)
FIN
```

Quelle est la complexité en temps de cet algorithme ? La fonction de complexité $f(n)$ satisfait la récurrence :

$$f(n) = \begin{cases} c & \text{si } n = 1 \\ p + f(\frac{n}{2}) & \text{sinon} \end{cases}$$

Supposons que $n = 2^k$ (une puissance de 2) et définissons $g(k) = f(2^k) = f(n)$. L'équation devient :

$$g(k) = \begin{cases} c & \text{si } k = 0 \\ p + g(k-1) & \text{sinon} \end{cases}$$

puisque

$$f(n/2) = f(2^k/2) = f(2^{k-1}) = g(k-1).$$

Alors on calcule de proche en proche :

$g(k) = p + g(k-1) = p + p + g(k-2) = p + p + p + g(k-3) = \cdots = (k-1)\cdot p + g(1) = k \cdot p + g(0) = k \cdot p + c$

donc $g(k) = k \cdot p + c$. Ainsi, en revenant à f et n, on obtient $f(n) = p \cdot \log_2(n) + c \in \Theta(logn)$.

Exercice 1 *Montrer que* $\log_2 n \in \Theta(\log_3 n)$ *(et, en fait, que pour tout* $b > 1$*, on a* $\log_2 n \in \Theta(\log_b n)$*).*

Exercice 2 *Montrer que* $2^n \in \mathcal{O}(3^n)$ *mais que* $3^n \notin \mathcal{O}(2^n)$.

Revenons aux algorithmes dont la complexité n'est pas seulement fonction de la taille. On a déjà vu des exemples d'analyse du meilleur cas et du pire des cas.

Comment effectuer l'analyse du cas moyen ?

Dans un tel cas, l'algorithme a des coûts différents pour certaines données. Ainsi, l'ensemble de toutes les données de taille n du problème peut être partitionné en classes. Chaque classe contient toutes les données de taille n pour lesquelles le coût en temps de l'algorithme est le même (*i.e.* la classe des données à coût le meilleur, la classe des données à coût le pire, et toutes les classes correspondantes aux coûts intermédiaires). Soient $c_1 < c_2 < c_3 < \cdots < c_k$ tous les coûts possibles pour l'algorithme sur une donnée de taille n et $p_1, p_2, p_3, \ldots, p_k$ les probabilités associées. Autrement dit, p_i est la probabilité que le coût de l'algorithme sur une donnée de taille n choisie aléatoirement soit c_i.

Definition 1 *Le coût moyen de l'algorithme pour une donnée de taille* n *est* $C(n) = \sum_{i=1}^{k} p_i c_i$.

Un exemple (facile) : la recherche « simple » ou « naïve » d'un élément dans un tableau (non trié).

```
Algo Recherche_naïve(A:tableau[1..n] d'entiers; x:entier) : booléen
DÉBUT
    pour i de 1 à n faire
        si (A[i]==x) retourner (vrai); (*on a trouvé*)
    finpour
    retourner (faux); (*on ne l'a pas trouvé*)
FIN
```

Le tableau suivant nous donne les coûts pour tous les cas possibles :

	Comparaisons	Retour	Coût total
Meilleur cas ($A[1] = x$)	1	1	2
cas ($A[2] = x$)	2	1	3
cas ($A[3] = x$)	3	1	4
$\vdots$	$\vdots$	$\vdots$	$\vdots$
cas ($A[n-1] = x$)	$n-1$	1	n
Pire des cas ($A[n] = x$ ou x$\notin A$)	n	1	$n+1$

Supposons de plus que nous connaissions la probabilité p que $x \in A$. Puisque si $x \in A$, toutes les positions sont équiprobables, la probabilité de trouver x à un certaine position i de A est p/n. On peut donc calculer la probabilité de chaque cas :

	Comparaisons	Retour	Coût total	Probabilité
Meilleur cas ($A[1] = x$)	1	1	2	$\frac{p}{n}$
cas ($A[2] = x$)	2	1	3	$\frac{p}{n}$
cas ($A[3] = x$)	3	1	4	$\frac{p}{n}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\frac{p}{n}$
cas ($A[n-1] = x$)	$n-1$	1	n	$\frac{p}{n}$
Pire des cas ($A[n] = x$ ou x$\notin A$)	n	1	$n+1$	$\frac{p}{n} + (1-p)$

$$\begin{aligned} C(n) &= \frac{p}{n}.\sum_{i=1}^{n}(i+1) + (1-p)(n+1) \\ &= \frac{p}{n}.\sum_{i=1}^{n} i + p + (1-p)(n+1) \\ &= \frac{p}{n}\frac{n.(n+1)}{2} + p + (1-p)(n+1) \\ &= \frac{pn + p + 2p + 2n + 2 - 2pn - 2p}{2} \\ &= \frac{1}{2}(2-p)n + \frac{1}{2}(p+2) \end{aligned}$$

On observe donc que $C(n) \in \mathcal{O}(n)$.

On note aussi que le coefficient du terme en n est plus grand quand p est petit ce qui peut facilement être expliqué.

2 Le problème du tri (sorting problem)

Étant donné un tableau A dont les éléments appartiennent à un ensemble totalement ordonné, permuter les éléments de manière telle que : $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Quelques algorithmes élémentaires.

Le tri par insertion

C'est un tri inspiré par la méthode que l'on emploie parfois pour ranger les cartes à jouer que l'on possède en main. L'idée est d'effectuer n passes de calcul. Lors de la i-ème passe, les $i-1$ premiers éléments du tableau sont déjà triés et l'on range le i-ème élément en effectuant des échanges avec les éléments à sa gauche jusqu'à ce qu'il se trouve à sa bonne place. Cette idée se traduit en l'algorithme suivant :

```
Algo Tri_insertion(A:tableau[1..n] d'entiers)
DÉBUT
    pour i de 2 à n faire (*dans la passe pour i = 1 il y a rien à faire*)
        j=i;
        tantque ((j>0) et (A[j]<A[j-1])) faire
            temporaire := A[j];
            A[j] := A[j-1];
            A[j-1] := temporaire;
            j := j-1;
        fintantque
```

```
    finpour
FIN
```

Quelle est la complexité en temps de cet algorithme ?

Analysons ce qui se passe dans le meilleur et le pire des cas.

Dans le meilleur des cas la condition ($A[j] < A[j-1]$) n'est jamais vraie, aucun échange sera effectué :

	Comparaisons	Échanges
Meilleur cas	$n-1$	0

Dans le meilleur cas, les éléments sont donc déjà triés et l'algorithme se contente de vérifier ceci en temps $\Theta(n)$.

Dans le pire des cas, il faut à chaque fois ramener une carte à insérer au tout début de sa « main » (ceci arrive quand le tableau est trié en ordre décroissant). Pour la i-ème passe, on effectuera $i-1$ comparaisons, $i-1$ échanges et $i-1$ décrémentations de la variable j (soit $5(i-1)$ opérations élémentaires, puisque un échange nécessite 3 affectations) soit au total :

$$5\sum_{i=2}^{n}(i-1) = \frac{5n(n-1)}{2}$$

opérations élémentaires.
On constate donc que dans le cas le pire, la complexité en temps est $\Theta(n^2)$.

Par ailleurs, il n'est pas difficile de monter que dans le cas moyen on a aussi : $C(n) \in \Theta(n^2)$.

Le tri par sélection (du minimum).

Idée de l'algorithme : on effectue n passes ; lors de la i-ème passe, les $i-1$ premiers éléments sont déjà triés. On cherche alors le plus petit élément parmi $A[i]$, $A[i+1], \ldots, A[n]$ et on le place à sa bonne position en l'échangeant avec l'élément $A[i]$.

```
Algo Tri_selection(A:tableau[1..n] d'entiers)
DÉBUT
    pour i de 2 à n faire  (*dans la passe pour i = 1 il y a rien à faire*)
        position_du_minimum := i;
        pour j de i+1 à n faire
            si (A[j]<A[position_du_minimum]) alors
                position_du_minimum := j;
        finpour
        temporaire := A[position_du_minimum];
        A[position_du_minimum] := A[i];
        A[i] := temporaire;
    finpour
FIN
```

Exercice 3 *Calculer le nombre précis de comparaisons et d'affectations effectués par l'algorithme dans le meilleur et dans le pire des cas.*

Algorithmes non élémentaires.

Le Tri par Fusion (Merge Sort), une autre application de la stratégie du « Diviser pour régner »

On partage le tableau en deux moitiés (si n est impair, l'une des deux moitiés contiendra un élément en plus). On trie chaque moitié indépendamment l'une de l'autre, puis on fusionne les deux moitiés qui viennent d'être triées.

```
Algo Tri_fusion(A:tableau[1..n] d'entiers; d,f:entiers)
DÉBUT
    si (d<f) alors (*si d=f, le tableau n'a qu'un seul élément et il n'y a rien à faire*)
     moitié = (d+f)/2; (*On calcule l'indice de la moitié du sous-tableau*)
        Tri_fusion(A,d,moitie);
        Tri_fusion(A,moitie+1,f);
        Fusion(A,d,moitie,f); (*Une macro-instruction à décrire!!!! Voir en dessous*)
FIN


Algo Fusion(A:tableau[1..n] d'entiers; d,m,f:entiers)
DÉBUT
    G,D:tableaux[1..n] d'entiers; i,j,k:entiers;
    pour i de d à m faire
        G[i-d+1] := A[i];  (*On copie la partie gauche dans G*)
    finpour
    pour i de m+1 à f faire
        D[i-m] := A[i]; (*On copie la partie droite dans D*)
    finpour
    G[m-d+2] := INFINI; (*INFINI est une valeur plus grande que toutes les autres*)
    D[f-m+1] := INFINI; (*c'est une sentinelle qui marque la fin des données*)
    i := 1; j := 1;
    pour k de d à f faire
        if (G[i]<D[j]) alors
            A[k] := G[i];
            i := i+1;
        sinon
            A[k] := D[j];
            j := j+1;
FIN
```

Le coût de `Fusion` est clairement $O(n)$. On notera que dans l'algorithme `Fusion` nous avons utilisé deux tableaux auxilliaires pour réaliser la fusion des deux portions du tableau. En effet, la fusion « sur place » des deux portions du tableau (sans utiliser de tableaux auxilliaires), si elle est tout à fait possible est un peu compliquée à écrire. D'autre part, nous avons aussi supposé l'existence d'une valeur sentinelle (le symbole `INFINI`) qui permet aussi de simplifier l'algorithme de fusion ; cette valeur, pour être utile, doit avoir comme propriété d'être plus grande que toutes les valeurs possibles que nous pourrions trouver dans les tableaux que nous souhaitons trier.

Quelle est la complexité en temps de cet algorithme ?

On a :

$$f(n) = \begin{cases} c & \text{si } n = 1 \\ 2f(\frac{n}{2}) + dn & \text{sinon} \end{cases}$$

où dn représente le coût de `Fusion`.
Supposons $n = 2^k$ (nous règlerons ensuite le problème des autres longueurs possibles). On pose $g(k) = f(2^k) = f(n)$ et on réécrit

$$g(k) = \begin{cases} c & \text{si } k = 0 \\ 2g(k-1) + d2^k & \text{sinon} \end{cases}$$

Nous avons

$$\begin{array}{lcl} g(k) & = & 2g(k-1) + d2^k \\ g(k-1) & = & 2g(k-2) + d2^{k-1} \\ g(k-2) & = & 2g(k-3) + d2^{k-2} \\ \vdots & & \vdots \\ g(2) & = & 2g(1) + d2^2 \\ g(1) & = & 2g(0) + d2^1 \\ g(0) & = & c \end{array}$$

en multipliant de façon adéquate les différentes équations par des puissances de 2 (de sorte qu'une élimination possible apparaisse) on obtient

$$\begin{array}{lcl} g(k) & = & 2g(k-1) + d2^k \\ 2^1 g(k-1) & = & 2^2 g(k-2) + d2^{k-1}2 \\ 2^2 g(k-2) & = & 2^3 g(k-3) + d2^{k-2}2^2 \\ \vdots & & \vdots \\ 2^{k-2} g(2) & = & 2^{k-1} g(1) + d2^2 2^{k-2} \\ 2^{k-1} g(1) & = & 2^k g(0) + d2^1 2^{k-1} \\ 2^k g(0) & = & c2^k \end{array}$$

Ainsi

$$g(k) = c2^k + dk2^k$$

Si on revient à n et à f on a

$$f(n) = c.n + dn \log n \in \mathcal{O}(n. \log n)$$

Pour le cas où n n'est pas une puissance de 2, il suffit de constater qu'il est possible de compléter le tableau en ajoutant des éléments à fin pour que sa longueur devienne la puissance de 2 la plus proche (par exemple si $n = 25$ on ajoutera 7 éléments pour atteindre la longueur 32). Les valeurs ajoutées sont simplement égales à la sentinelle et n'influeront donc pas au final.

Cette opération modifie t-elle la complexité ? Non ! L'opération de complétion du tableau est une opération linéaire (de l'ordre de la taille du tableau, car au pire il faut doubler la longueur du tableau) ; mais comme la complexité est $\mathcal{O}(n \log n)$, le prix linéaire supplémentaire à payer est simplement négligeable devant l'opération de tri.
Note : Alors que les algorithmes élémentaires (`Insertion` et `Sélection`) nécessitent $\mathcal{O}(1)$ mémoire (leur complexité en espace est $\mathcal{O}(1)$), le `Tri_fusion` présenté ici utilise $\mathcal{O}(n)$ mémoire (les deux tableaux G et D), ceci car il a été décidé de ne pas fusionner sur place pour des raisons de simplicité. On peut indiquer ici, sans l'exhiber, qu'il existe bien un tri par fusion dont la complexité en temps est la même et dont la complexité en mémoire est en $\mathcal{O}(1)$; cette version du tri par fusion utilise une fusion « sur place » des données dont le coût en temps ne modifie pas globalement la complexité de l'algorithme.

En plus du temps, d'autres critères peuvent déterminer le choix de l'algorithme de tri le plus approprié.

Definition 2 (Tri sur place) *Un algorithme de tri est « sur place » s'il n'utilise que $\mathcal{O}(1)$ mémoire.*

Par exemple, `Tri_insertion` est *on line* alors que `Tri_fusion` ne l'est pas.

Definition 3 (Tri *on line*) *Un algorithme de tri est* on line *s'il peut s'exécuter au fur et à mesure de l'arrivée des données.*

Par exemple, `Tri_insertion` est *on line* alors que `Tri_fusion` ne l'est pas.

Note : il s'avère que les constantes multiplicatives qui apparaissent dans les fonctions de complexité de ces algorithmes font que pour des petites valeurs de n certains algorithmes rapides pour de grandes valeurs de n sont lents et vice versa. Expérimentalement, on constate que pour $n \leq 10$, le tri par insertion est le plus rapide. En conséquence, tout algorithme de tri peut être amélioré en effectuant un simple test sur la taille de la donnée et choisir le tri le plus adéquat pour cette taille. Par exemple, si $n \leq 10$ on appelle `Tri_insertion` sinon un autre tri.

Le tri rapide (Quick sort)

Il s'agit d'un algorithme récursif aussi basé sur le « diviser pour régner » dont la complexité est $\mathcal{O}(n \log n)$ dans le cas moyen mais $\mathcal{O}(n^2)$ dans le pire des cas.
L'idée est de partitionner le tableau non pas sur la critère de sa taille mais en choisissant dans le tableau un pivot $A[p]$ (un élément du tableau) et en réarrangeant les éléments de sorte que tous les éléments plus petits $A[p]$ se trouvent à sa gauche et que tous les éléments plus grands que ou égaux à $A[p]$ soient à sa droite. On remarque qu'après cette opération, $A[p]$ est à sa bonne place. On recommence ensuite sur les deux parties gauche et droite.

```
Algo Tri_rapide(A:tableau[1..n] d'entiers; d,f:entiers)
DÉBUT
    indice_pivot:entier;
    si (d<f) alors
        indice_pivot := d; (*ici le pivot est toujours le 1er élément de la partition*)
        indice_pivot := Partition(A,d,f,indice_pivot); (*le pivot est à sa place*)
        Tri_rapide(A,d,indice_pivot-1); (*Un coup à gauche*)
        Tri_rapide(A,indice_pivot+1,f); (*Un coup à droite*)
FIN
```

Exercice 4 *Trouver un algorithme pour* `Partition` *qui fonctionne en temps* $\mathcal{O}(n)$.

Et la complexité de ce `Tri_rapide` (en supposant que `Partition` est de complexité $\mathcal{O}(n)$) ?

$$f(n) = \begin{cases} c & \text{si } n = 1 \\ dn + f(k) + f(n-k-1) & \text{sinon (avec } k \text{ la position du pivot)} \end{cases}$$

On a le meilleur cas quand les deux parties ont (à une unité près) la même taille. On tombe alors sur le même type de récurrence que pour `Tri_fusion` et donc une complexité en $O(n \log n)$.

Si par malheur, le pivot était toujours la plus petite valeur de la partition (une des deux parties est vide), on aurait

$$\begin{aligned} f(n) &= dn + f(n-1) \\ &= dn + d(n-1) + f(n-2) \\ &\vdots \\ &= d(n + n - 1 + n - 2 + n - 3 + \ldots + 1) \\ &= d\tfrac{n(n-1)}{2} \end{aligned}$$

Donc dans ce cas (qui est le pire) on a

$$f(n) \in \mathcal{O}(n^2)$$

Dans le pire des cas, `Tri_rapide` est en $\mathcal{O}(n^2)$ (ceci correspond au cas dans lequel l'une des deux partitions est toujours vide). Si la stratégie du pivot consiste simplement à choisir le premier élément, ce cas se vérifie quand le tableau est déjà trié ! ! !
Pour éviter d'obtenir des parties vides, on peut adopter la stratégie consistant à choisir comme pivot l'élément médian parmi les 3 premiers éléments de la partition.

On peut montrer que la complexité de `Tri_rapide` est $\mathcal{O}(n \log n)$ et qu'en fait on a cette complexité même si `Partition` effectue un partage très déséquilibré (une partie beaucoup plus grande que l'autre).

La question générale de la complexité du tri

Les tris basés sur des comparaisons

La question est : « Peut-on faire mieux que $\mathcal{O}(n \log n)$ dans le pire des cas ? »
La réponse est non pour les algorithmes de tri basés sur l'opération de comparaison (s'ouvre donc une autre question « Comment trier sans comparer les éléments ? » ; une réponse sera fournie plus tard).
Cette réponse est obtenue par observation des arbres binaires (les nœuds ont au plus 2 descendants).

On rappelle qu'un arbre binaire complet est un arbre dans lequel chaque nœud interne a exactement deux fils et les feuilles (nœuds sans descendant) sont toutes à la même profondeur (à la même distance de la racine). Si h est la hauteur de l'arbre (la plus longue descendance est de longueur h), un tel arbre contient $f = 2^h$ feuilles. Il est évident que pour tout autre arbre binaire de hauteur h, son nombre de feuilles est $f \leq 2^h$.

Imaginons l'arbre de décision binaire dans lequel à chaque nœud qui n'est pas une feuille, correspond un test entre deux éléments ($x_i < x_j$?) (voir la figure 1 pour un exemple). Si la réponse est vraie on a une réponse sur la position relative des éléments x_i et x_j dans le tableau trié, il faut sinon considérer de modifier les positions relatives. Une analyse simple permet de constater que les feuilles sont nécessairement en nombre $n!$ (une pour chaque permutation possible, un algorithme de tri quelconque doit être capable d'effectuer toutes ces permutations). La hauteur de l'arbre de décision vérifie donc nécessairement $h > \log(n!)$.

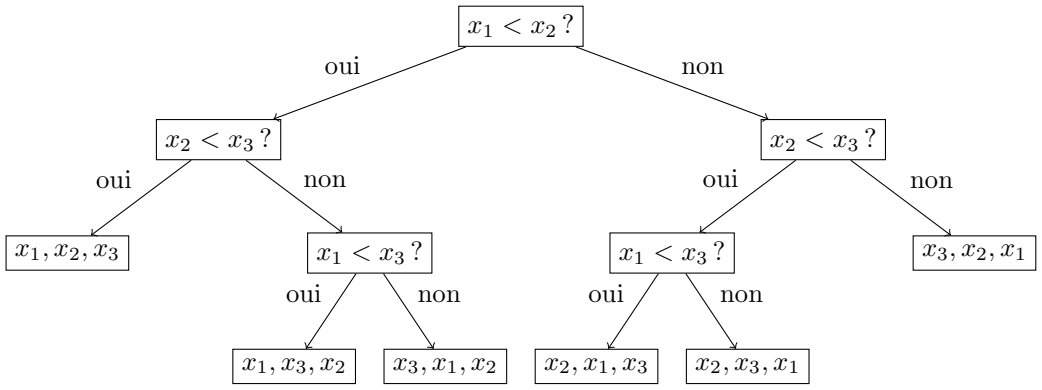


FIGURE 1 – Arbre de décision pour trier un tableau de trois éléments

Montrons que $\log_2(n!) \in \Theta(n \log n)$.
D'abord, $\log(1.2.3 \ldots n) < \log(n.n.n \ldots n) = \log(n^n) = n \log n$.

Exercice 5 *Montrer que* $n \log n \in \mathcal{O}(\log(n!))$

En conséquence, un algorithme de tri basé sur des comparaisons effectue au moins $\Theta(n \log n)$ comparaisons pour trier les objets dans le pire des cas.

Un tri n'utilisant pas de comparaisons

Supposons que nous sachions que les valeurs présentes dans le tableau à trier A appartiennent toutes à l'intervalle fini discret $[a, b]$. Sans perte de généralité et pour faciliter les choses on supposera que $a = 1$ et $b = m$. Le tri sans comparaison est alors obtenu en introduisant un tableau de compteurs $C[1..m]$ d'entiers tel que $C[k] = |\{A[i] \,/\, A[i] = k\}|$.

Cette idée se traduit en un simple algorithme dit de `Tri_par_comptage` (ou *Counting sort*) :

```
Algo Tri_par_comptage(A:tableau[1..n] d'entiers; m:entier)
DÉBUT
    C:tableau[1..m] d'entiers;
    pour i de 1 à m faire (*initialisation du tableau de compteurs*)
        C[i] := 0;
    finpour
    pour i de 1 à n faire (*comptage*)
        C[A[i]] := C[A[i]]+1;
    finpour
    k := 1;
    pour i de 1 à m faire (*remise en place des objets comptés*)
        pour j de 1 à C[i] faire
            A[k] := i;
            k := k+1;
        finpour
    finpour
FIN
```

Quelle est la complexité de cet algorithme ? La première boucle est clairement en $\mathcal{O}(m)$, alors que la deuxième est en $\mathcal{O}(n)$. Pour la double boucle imbriquée on a que la boucle intérieure est de complexité $\mathcal{O}(C[i])$, il faut donc sommer $\mathcal{O}(\sum_{i=1}^{m} C[i]) = \mathcal{O}(n)$. Au total, cela donne $\mathcal{O}(\max(m, n))$.

Exercice 6 *Transformer l'algorithme de sorte qu'il permette de trier un tableau de n éléments compris entre des bornes $[a, b]$ quelconques ($a, b \in \mathbb{Z}$).*

Si $m \in \mathcal{O}(n)$ alors on obtient un tri en $\mathcal{O}(n)$ dans le pire des cas. Par contre, si m est d'un ordre de grandeur plus grand que celui de n, on obtient un algorithme en $\mathcal{O}(m)$ ce qui rend, en pratique, l'algorithme peu intéressant dès que m est au moins $\mathcal{O}(n \log n)$.

Definition 4 *Un algorithme de tri est dit **stable** s'il préserve l'ordre relatif des éléments égaux (ou ayant clé égale).*

Exercice 7 *Parmi les algorithmes déjà étudiés, déterminer lesquels sont stables et lesquels ne le sont pas.*

3 Les arbres

Nous avons pour l'instant exhibé des algorithmes qui travaillent sur de (très) simples structures de données : les tableaux. Nous verrons ensuite que certains tris utilisent d'autres types de structures de données. Nous allons nous arrêter un peu sur celle d'arbre.

Definition 5 *Un objet tout seul forme un arbre d'un seul nœud dont il est la racine.*

Si $A_1, A_2, \cdots, A_p$ sont p arbres de racines respectives $r_1, r_2, \cdots, r_p$, on peut créer un nouvel arbre de racine r en définissant que r est le père de $r_1, r_2, \cdots, r_p$ (l'ordre des fils est important).

Definition 6 *L'arité d'un arbre est le nombre maximal de fils qu'un nœud de cet arbre peut possèder.*

Definition 7 *La hauteur d'un arbre est la longueur du plus long chemin menant de la racine à une feuille.*

Implémentation à l'aide d'un tableau

La réalisation d'un arbre (on parle aussi d'**implémentation**) peut être effectuée par utilisation d'un tableau `PÈRE` représentant la relation de parenté, où `PÈRE`$[i]$ est le numéro du nœud qui est le père du nœud i.

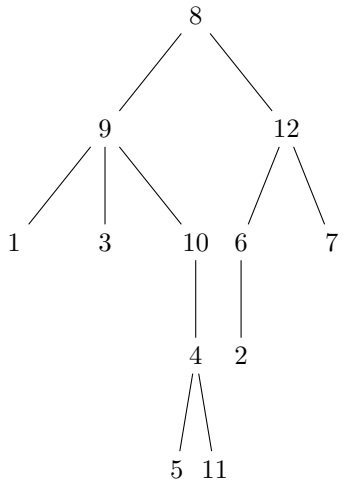


FIGURE 2 – Un arbre.

En général ceci est représenté en machine par deux tableaux : le tableau `PÈRE` (qui définit la relation de parenté - par exemple celle de l'arbre de la figure 2) et, un autre tableau où on range les données contenues dans les noeuds :

1	2	3	4	5	6	7	8	9	10	11	12	nœud
9	6	9	10	4	12	12	8	8	9	4	8	numéro de son père
3,8	2,4	.	.	.	.	.	.	.	.	.	.	*informations* attachées au nœud

On note que, par convention, le père du nœud racine est lui-même (dans l'exemple, `PÈRE[8]=8`).

Implémentation à l'aide de pointeurs

Un nœud sera une structure de données composée d'une partie informative (contenant les informations que l'on souhaite stocker) et d'un ensemble de p pointeurs (si l'arité de l'arbre est p) chacun d'eux pouvant désigner (pointer) un autre nœud de même nature.
En particulier pour les arbres d'arité 2 (dits arbres binaires), il n'y a que deux fils possibles donc deux pointeurs (voir figure 3), l'un vers le fils gauche (`fg`) et l'autre vers le fils droit (`fd`) .

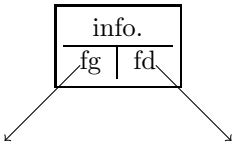


FIGURE 3 – Structure d'un nœud d'arbre binaire

Si l'arité n'est pas déterminée à l'avance, il est possible d'utiliser un arbre binaire avec les deux relations `PREMIER_FILS` et `FRERE_DROIT`, ainsi seuls deux pointeurs sont nécessaires.

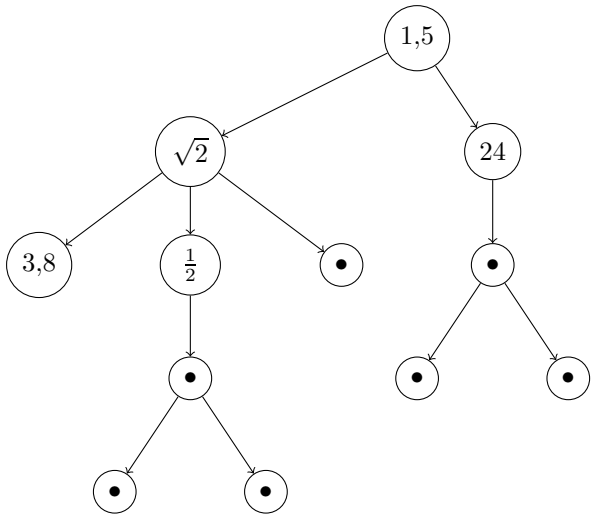


FIGURE 4 – Un arbre n-aire

Après avoir définit le type nœud de manière appropriée, on définit le type arbre comme un pointeur vers un nœud (sa racine).
Notation : Si `p` est un pointeur, on note par `*p` l'objet pointé par `p` (l'objet dont l'adresse se trouve dans `p`). Si l'objet pointé par `p` est un type structuré, on note par `p->field` le champ `field` de l'objet pointé par `p`. Ainsi, si `A` est un objet de type arbre (un pointeur sur un nœud), on utilisera par exemple `*A` pour indiquer le nœud racine de `A`, ou `A->info` pour indiquer juste sa partie informative, ou `A->fg` pour avoir l'adresse de son fils gauche, etc.
Rappel : On a vu la possibilité de définir des structures *arborescentes* pour stocker des données. La relation de parenté peut être représentée soit à l'aide d'un tableau contenant la relation `PÈRE`, soit à l'aide de structures contenant des pointeurs qui, à partir d'un nœud de l'arbre permettent d'accéder à d'autres nœuds (généralement ses fils).
Note : Avec le tableau `PÈRE`, on peut déterminer en $\mathcal{O}(1)$ le père d'un nœud donné, alors que pour déterminer qui sont les fils d'un nœud donné il faut $\mathcal{O}(n)$ (dans le pire des cas). En revanche, avec la représentation par pointeurs on détermine en $\mathcal{O}(1)$ l'adresse des fils d'un nœud donné, mais il est plus difficile de remonter à l'adresse du père d'un nœud donné.
Une solution possible pour résoudre ce petit problème de complexité avec les pointeurs consiste à ajouter dans la structure d'un nœud un pointeur supplémentaire, permettant pour tout nœud de retrouver directement son père (voir figure 6).

Les Parcours d'arbre

Puisque les arbres contiennent des données, on a fréquemment besoin de parcourir la structure afin de visiter les nœuds (pour chercher l'un d'entre eux par exemple, ou pour les afficher, etc.)

Le parcours préfixe d'un arbre

Ce parcours permet d'obtenir les feuilles et les nœuds en ordre préfixe (de *gauche à droite*), L'idée est donc de :
– rendre visite à la racine ;
– parcourir récursivement et en ordre préfixe les sous-arbres A_1, A_2, …, A_p.
On obtient alors l'algorithme `Parcours_préfixe` suivant :

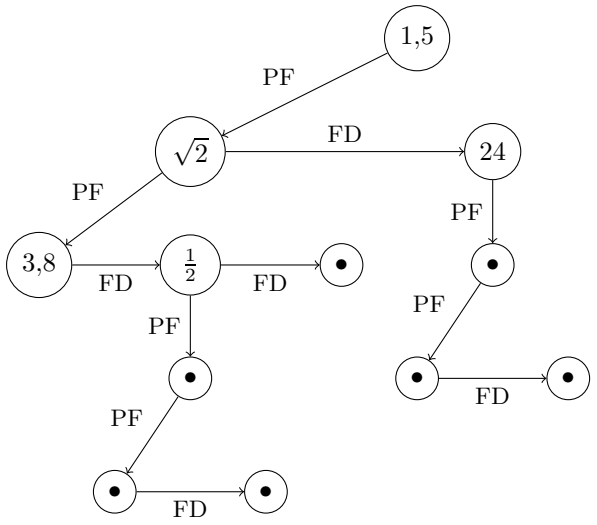


FIGURE 5 – Un arbre n-aire codé à l'aide d'un arbre binaire.

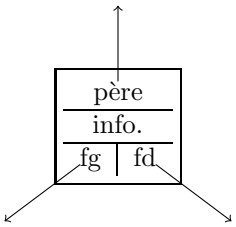


FIGURE 6 – Structure d'un nœud d'arbre binaire avec ajout de la relation `PÈRE`

```
Algo Parcours_préfixe(A:arbre)
DÉBUT
    si (A <> NULL) alors
        Rendre_visite(*A); (*faire ce que l'on a à faire avec le n\oeu{}d courant*)
        Parcours_prefixe(A.fils[1]);
        Parcours_prefixe(A.fils[2]);
        (* ... *)
        Parcours_prefixe(A.fils[p]); (*si p est l'arité de l'arbre*)
FIN
```

Le parcours postfixe d'un arbre

Le parcours postfixe est obtenu par simple symétrie de l'algorithme de parcours préfixe de la façon suivante :
– parcourir récursivement et en ordre postfixe les sous-arbres A_1, A_2, …, A_p ;
– rendre visite à la racine.
On obtient alors l'algorithme `Parcours_postfixe` suivant :

```
Algo Parcours_postfixe(A:arbre)
DÉBUT
    si (A <> NULL) alors
        Parcours_postfixe(A.fils[1]);
```

```
        Parcours_postfixe(A.fils[2]);
        (* ... *)
        Parcours_postfixe(A.fils[p]); (*si p est l'arité de l'arbre*)
        Rendre_visite(*A); (*faire ce que l'on a à faire avec le n\oeu{}d courant*)
FIN
```

Le parcours infixe d'un arbre

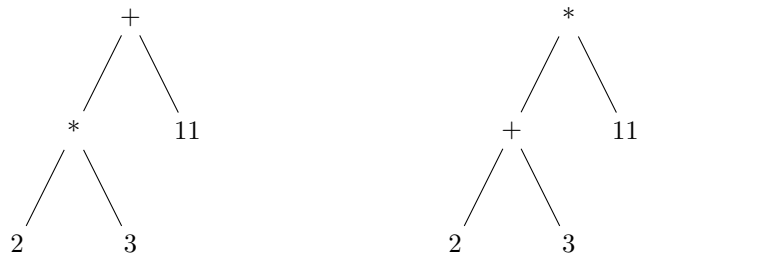
Bien qu'il soit possible de le définir avec les arbres d'arité quelconque, cela prend en général tout son sens avec les arbres binaires (par exemple ceux représentant une expression arithmétique). Pour les arbres p-aires, l'idée est donc :
– parcourir en infixe le sous-arbre A_1 ;
– visiter la racine ;
– parcourir récursivement et en ordre infixe les sous-arbres $A_2, \ldots, A_p$.
Donc pour les arbres binaires ceci se réduit à :
– parcourir en infixe le sous-arbre A_g ;
– visiter la racine ;
– parcourir en infixe le sous-arbre A_d.

```
Algo Parcours_infixe(A:arbre binaire)
DÉBUT
    si (A <> NULL) alors
        Parcours_infixe(A.filsGauche);
        Rendre_visite(*A); (*faire ce que l'on a à faire avec le noeud courant*)
        Parcours_infixe(A.filsDroit);
FIN
```

La complexité en temps des algorithmes de parcours d'arbres est en $\mathcal{O}(N)$, où N est le nombre de nœuds. En effet, on effectue un appel à la fonction de parcours une et une seule fois pour chacun des N nœuds et chaque appel prend un temps en $\mathcal{O}(1)$.

Exercice 8 *Donner le résultat pour les parcours préfixe et postfixe de l'arbre de la figure 2.*

La figure 7(a) illustre la représentation sous la forme d'un arbre binaire d'une expression arithmétique. Attention car cette représentation suppose, dans la lecture infixe, l'utilisation de parenthèse... En effet,



(a) Exemple

(b) La lecture infixe peut être ambigüe si l'on ne prend pas de précautions.

FIGURE 7 – Des expressions arithmétiques codées dans un arbre binaire.

l'arbre de la figure 7(b) représente l'expression $(2+3)*11$ mais sa lecture infixe sans utilisation de parenthèse donnerait $2+3*11$, ce qui est évidemment très différent. Donc, en lecture infixe, il est fréquent d'utiliser un système de parenthèses pour éviter les problèmes.

Des arbres particuliers : les tas

Supposons dorénavant que les données de l'arbre fassent partie d'un ensemble totalement ordonné (*i.e.* on peut les comparer) ; par exemple, des nombres entiers, des chaînes de caractères, etc.

Definition 8 *Un arbre binaire A de hauteur h est un tas si :*
- *pour tout i avec $0 \leq i < h$, l'arbre A possède exactement 2^i nœuds de profondeur i ;*
- *les nœuds de profondeur h sont stockés le plus à gauche possible ;*
- *pour tout nœud n de A, le contenu de n est plus petit (respectivement plus grand) que le contenu de ses deux fils, on parle alors de* `Tas_min` *(resp.* `Tas_max`*)*

La figure 8 est un exemple de `Tas_min`.

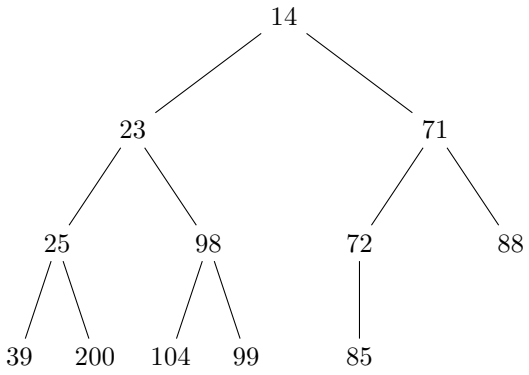


FIGURE 8 – Un exemple de `Tas_min`.

Remarque : Si on numérote les nœuds du haut vers le bas et de la gauche vers le droite (la racine porte le numéro 1), les fils du nœud de numéro k, portent les numéros $2k$ et $2k+1$. Cette remarque permet d'implémenter un tas de taille N à l'aide d'un tableau de taille N en stockant dans la composante k du tableau le contenu du nœud qui porte le numéro k, comme ceci :

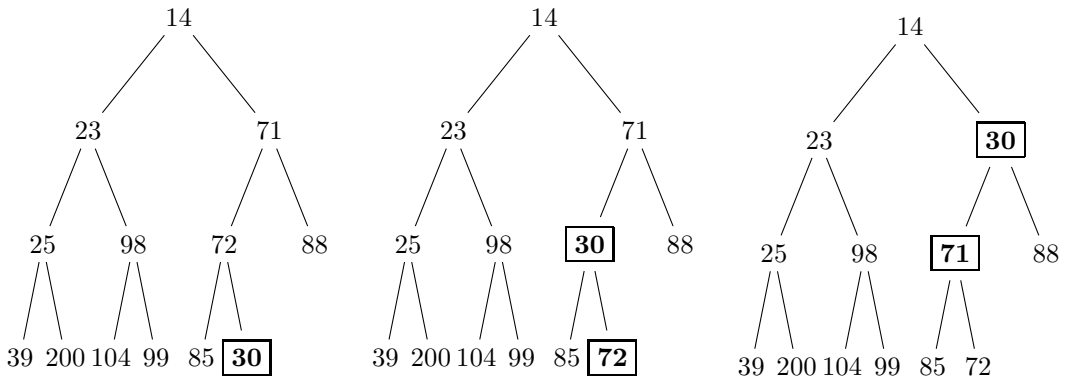
1	2	3	4	5	6	7	8	9	10	11	12
14	23	71	25	98	72	80	39	200	104	99	85

Note : Si un tas contient N nœuds, sa hauteur h est en $\mathcal{O}(\log N)$.
Une application des tas sont les files de priorité. Schématiquement, il s'agit d'une structure de données sur laquelle on veut effectuer les opérations suivantes :
- l'ajout d'un nouvel élément ;
- la suppression de l'élément minimal.

L'insertion d'un élément dans un tas

L'insertion de l'élément 30 produit dans l'arbre de la figure 8 la séquence de la figure 9.

Note : Pour des raisons pratiques, en machine, un tas est une structure à deux *champs*, un tableau nommé `contenu` stockant les données, et un entier nommé `taille` contenant la longueur effective du tableau. Il s'agit d'un tableau dynamique (dont la taille est variable).



(a) étape 1 (b) étape 2 (c) étape 3

FIGURE 9 – L'insertion de l'élément 30 dans le tas de la figure 8.

```
Algo Insertion_tas(T:Tas; x:élément)
DÉBUT
    T.taille := T.taille+1; (*le tas contient un élément de plus*)
    T.contenu[T.taille] := x; (*on range le nouvel élément en bout*)
    i := T.taille;
    tant que ( (i>1) et (T.contenu[i/2]>T.contenu[i]) ) faire
        Échanger(T.contenu[i/2],T.contenu[i]);
        i := i/2;
    fintantque
FIN
```

La complexité est $\mathcal{O}(h)$ car on effectue au plus h échanges, donc l'algorithme est en $\mathcal{O}(\log N)$.

La suppression d'un élément dans un tas

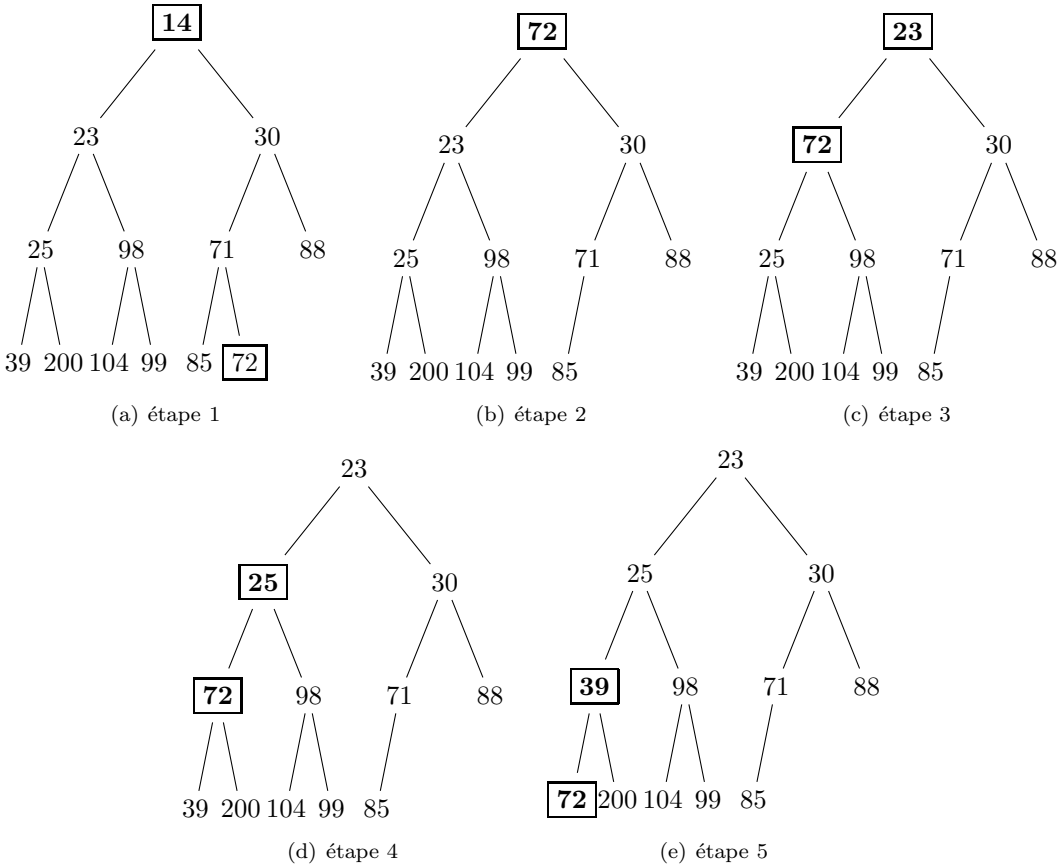
La suppression de l'élément 14 produit dans l'arbre de la figure 8 la séquence de la figure 10.

Exercice 9 *Écrire l'algorithme de suppression de l'élément minimal d'un tas en s'inspirant de la figure 10.*

Le tri par tas

On peut donc écrire un tri utilisant un tas de la façon suivante :

```
Algo Tri_tas(A:Tableau[1..n] d 'entiers)
DÉBUT
    T := TAS_VIDE;
    pour i de 1 à n faire
         Insertion_tas(T,A[i]);
    finpour
    pour i de 1 à n faire
         A[i] := Suppression_tas(T);
    finpour
FIN
```



(a) étape 1 (b) étape 2 (c) étape 3

(d) étape 4 (e) étape 5

FIGURE 10 – La suppression de l'élément 14.

On calcule facilement la complexité de cet algorithme qui est $\mathcal{O}(n \log n)$.

Note : On remarquera que cet algorithme `Tri_tas` utilise une mémoire auxiliaire de taille $\mathcal{O}(n)$ (le tas). Mais on peut construire une version dans laquelle le tri s'effectue sans cette mémoire auxiliaire, donc qui n'utilise pas de mémoire autre que les variables auxiliaires habituelles en $\mathcal{O}(1)$. On dira d'un tel tri qu'il s'effectue **sur place** ou **en place**.
L'idée est de d'abord considérer que le tableau est coupé en deux : une première partie (à gauche) contenant un `tas_max` et une seconde (à droite) contenant les éléments non triés.
La première phase consiste donc à grignoter petit à petit la partie droite en insérant ses éléments dans le tas à gauche (suppression d'un élément en partie droite, insertion dans le tas à gauche, la frontière se décale vers la droite).
La seconde phase consiste à considérer le tableau comme coupé en deux : une première partie (à gauche) avec un `tas_max` et la seconde (à droite) avec une liste d'éléments triés. On grignote petit à petit le tas (donc on retire les éléments dans l'ordre inverse, du plus grand au plus petit) de façon à remplir la partie droite triée.

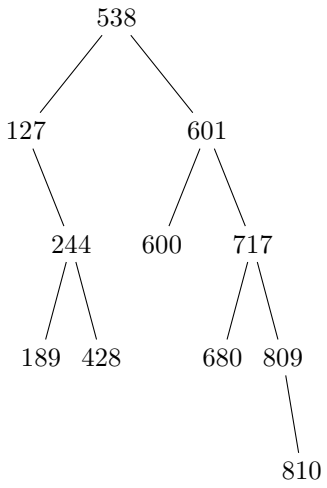


FIGURE 11 – Un arbre binaire de recherche.

Les dictionnaires et les arbres binaires de recherche

Definition 9 *On appelle* **Dictionnaire** *un ensemble de données sur lequel on peut effectuer les opérations suivantes :*
- *ajout d'un nouvel élément ;*
- *suppression d'un élément existant ;*
- *recherche d'un élément.*

Il est donc nécessaire de trouver une implémentation des dictionnaires telle que ces trois opérations s'effectuent en $\mathcal{O}(\log n)$ en moyenne ou idéalement en $\mathcal{O}(\log n)$ dans le pire des cas.
La structure de données que nous allons utiliser pour réaliser les dictionnaires s'appelle de façon générique **arbre binaire de recherche**.

Les arbres binaires de recherche

Definition 10 *Un arbre binaire A est un* **arbre binaire de recherche** *(on notera parfois ABR) s'il satisfait la condition suivante :*

> *Pour tout nœud n de A, le contenu de n est plus grand (respectivement plus petit) que le contenu de tous les nœuds qui se trouvent dans le sous-arbre gauche (resp. droit) de n.*

Voir la figure 11 pour un exemple.

Remarque : Le parcours infixe d'un arbre binaire de recherche permet de reconstruire une liste triée des éléments. D'autre part, on rappelle qu'un parcours d'arbre se fait en temps $\mathcal{O}(N)$ (où N est la taille de l'arbre).

La recherche d'un élément dans un ABR

L'algorithme suivant retourne `NULL` si l'élément à rechercher ne se trouve pas dans l'ABR. Il retourne par contre l'adresse du nœud contenant l'élément, si celui-ci est bien dans l'ABR.

```
Algo Recherche_ABR(A:ABR; x:élément):ABR
DÉBUT
    si (A==NULL) alors
```

```
        retourner (A);
    si (x==A->info) alors
        retourner (A);
    si (x<A->info) alors
        retourner (Recherche_ABR(A->gauche,x));
    sinon
        retourner (Recherche_ABR(A->droit,x));
FIN
```

L'insertion d'un élément dans un ABR

```
Algo Insertion_ABR(A:ABR; x:élément)
DÉBUT
    si (A==NULL) alors
        nouveau(A); (*on créé un nouveau noeud et on met son adresse dans A*)
        A->info := x;
        A->gauche := NULL;
        A->droit := NULL;
    sinon
        si ( x < A->info ) alors
            Insertion_ABR(A->gauche,x);
        sinon
            Insertion_ABR(A->droit,x);
FIN
```

La suppression d'un élément dans un ABR

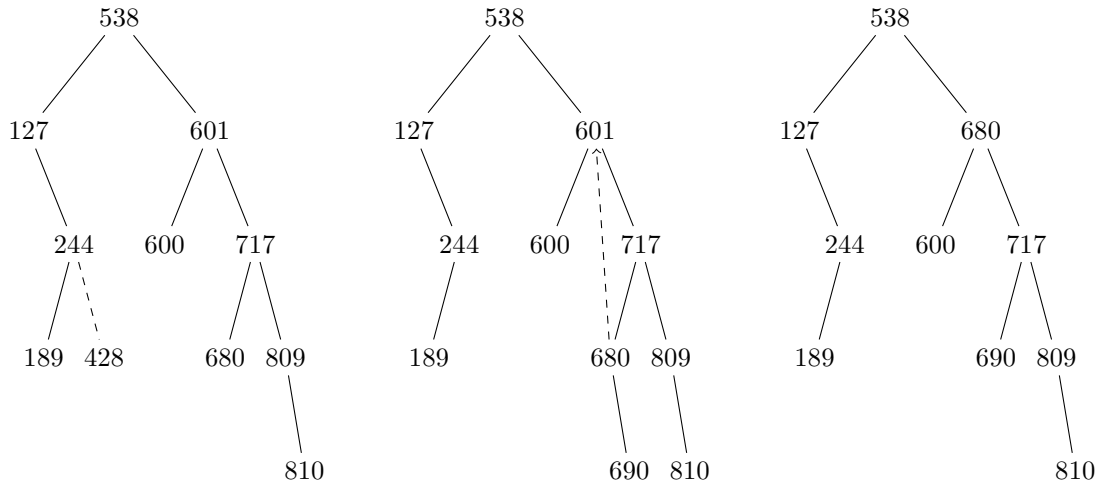
La suppression nécessite quelques précautions car des cas un peu particuliers peuvent se produire. On remarquera aussi que si l'idée générale est simple, de nombreux détails techniques (la gestion des pointeurs et de la mémoire allouée dynamiquement) encombrent lourdement l'écriture de l'algorithme.

Le premier exemple est la suppression de l'élément 428 dans l'exemple de la figure 11, qui conduit à l'ABR de la figure 12(a). Le cas est facile car le nœud ne possède pas de descendant. Le nœud doit être simplement libéré (sa mémoire doit être rendue) et le lien de son père vers lui détruit.

Le deuxième exemple est la suppression de 127. Ce cas est aussi relativement facile car le nœud possède un seul enfant. Dans ce cas, pour supprimer le nœud il suffit de le « courtcircuiter » en ajoutant un lien direct de son père (qui contient 538) vers son fils (qui contient 244). Pour ceci il suffit de modifier la valeur du pointeur du nœud père pour qu'il pointe directement sur son (unique) petit-fils. Il faudra faire attention à libérer la mémoire où était stocké le nœud contenant l'élément à supprimer et pour ceci quelques astuces seront nécessaires.

Le troisième exemple est la suppression de l'élément 601 qui nécessite un peu plus de travail (voir les figures 12(b) et 12(c). Ici il ne faut pas supprimer le nœud contenant 601 mais supprimer le nœud contenant la plus petite valeur de son sous-arbre droit et prendre sa valeur (ici 680) pour la mettre à la place de 601 ; ou de façon symétrique, supprimer le nœud contenant la plus grande valeur de son sous-arbre gauche (ici 600), prendre sa valeur pour la mettre à de 601. Il est facile de voir que l'un comme l'autre de ces valeurs se trouvent toujours dans un nœud ayant zéro ou un fils (on sait donc les supprimer comme vu dans les deux exemples précédents). Dans l'algorithme nous avons choisi la première possibilité, d'où l'écriture de l'algorithme `Suppression_min` (qui retourne aussi la valeur minimale contenue dans l'ABR).

```
Algo Suppression_min(A:ABR;):élément
DÉBUT
```



(a) Suppression de 428 (b) Suppression de 601 (c) Suppression de 601

FIGURE 12 – La suppression dans un ABR.

```
    tant que ( (A->gauche) <> NULL ) faire
        A := A->gauche;
    fintantque  (*à la fin de la boucle, A pointe sur le noeud contenant le minimum *)
    valeur := A->info;
    q := A;        (*on sauvegarde l'adresse du noeud pour libérer la mémoire ensuite*)
    A := A->droit; (*le noeud à supprimer possède au plus un fils droit *)
    libère(q);
    retourner (valeur);
FIN


Algo Suppression_ABR(A:ABR; x:élément):ABR
DÉBUT
    p : ABR;
    p := Recherche_ABR(A,x);
    si (p <> NULL) alors
        (*le noeud n'a pas de fils*)
        si ( (p->gauche=NULL) et (p->droit=NULL) ) alors
            libère(p);
            p := NULL;
        sinon
            (*le noeud a au moins un fils*)
            si ( (p->gauche=NULL) ou (p->droit=NULL) ) alors
                q := p;
                (*le noeud n'a pas de fils gauche*)
                si (p->gauche=NULL) alors
                    p := p->fd;
                sinon
                    (*le noeud n'a pas de fils droit*)
                    p := p->gd;
                libère (q);
```

```
            sinon
                (*le noeud a deux fils*)
                p->info := Suppression_min(p->fd);
FIN
```

Quelle est la complexité de la suppression ?

La plupart des opérations sont en temps constant, seules les utilisations de `Recherche_ABR` et `Suppression_min` ont un coût non constant en $\mathcal{O}(h)$ (h est la hauteur de l'arbre). On observera que dans le pire des cas on a $h = \mathcal{O}(n)$ (n est le nombre de nœuds) et l'algorithme est donc linéaire (l'arbre est tellement déséquilibré qu'il est équivalent à une quasi liste chaînée). Toutefois, on peut démontrer que dans le cas moyen, l'arbre est suffisamment équilibré de sorte que la complexité moyenne est $\mathcal{O}(\log n)$.

Les arbres binaires de recherche équilibrés

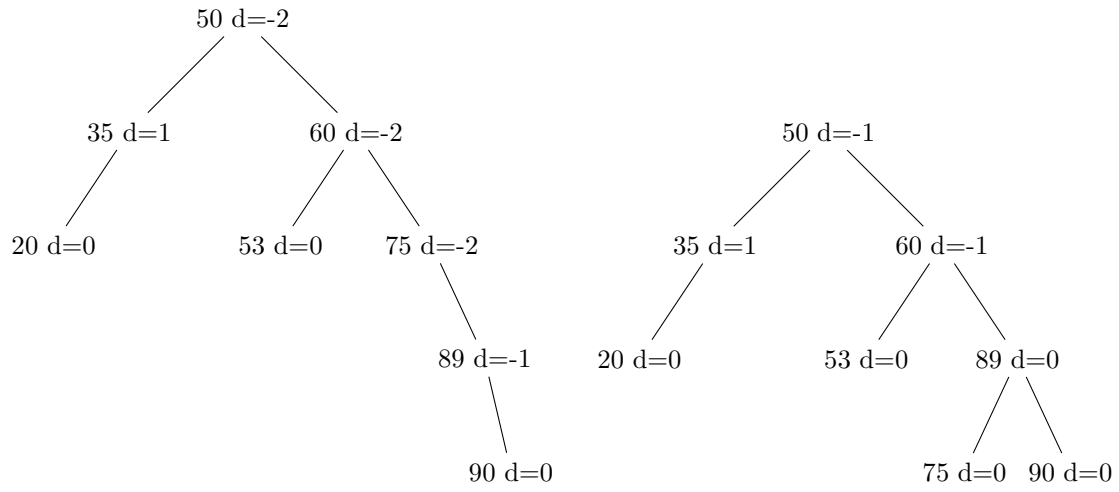
Nous venons de voir que le problème est que, tels que définis, les arbres binaires de recherche peuvent être parfois suffisamment déséquilibrés de sorte que les algorithmes ne se comportent pas de façon satisfaisante ou attendue du point de vue de leur complexité. Pour garantir la complexité logarithmique, il est possible d'enrichir la structure de données de sorte qu'une modification appropriée des algorithmes permettent de garantir un certain équilibre dans l'arbre.
L'idée générale est de maintenir en tout nœud une donnée permettant de mesurer le déséquilibre, par exemple la hauteur du sous-arbre ayant ce nœud comme racine, ou la différence des hauteurs de ses deux propres sous-arbres. Nous ne détaillerons pas ici ces algorithmes mais l'un d'entre eux est illustré dans la figure 13 (le déséquilibre y est représenté par la valeur d qui donne, pour chaque noeud la différence entre la hauteur du sous-arbre gauche et celle du sous-arbre droit). Pour maintenir l'ABR suffisamment équilibré (de sorte que sa hauteur totale reste en $\mathcal{O}(\log n)$), il suffit de s'assurer que le déséquilibre de chaque nœud reste compris entre -1 et 1.
L'algorithme de rééquilibrage n'est pas détaillé ici mais on peut mentionner que sa complexité est en $\mathcal{O}(\log n)$. Donc maintenir l'équilibre d'un arbre de façon à garantir la complexité de ses algorithmes `Recherche_ABR`, `Insertion_ABR` et `Suppression_ABR` n'est pas une opération qui modifie leur complexité moyenne mais abaisse leur complexité dans les pires de cas à $\mathcal{O}(\log n)$.

4 Les tableaux de Young et l'algorithme RSK

RSK désigne les concepteurs de l'algorithme soit : Robinson, Schensted, et Knuth.
Dans sa forme particulière présentée ici, l'algorithme associe à une suite de lettres $a_1, a_2, \ldots, a_n$ (ici il s'agit d'entier, mais on peut utiliser n'importe quel autre alphabet totalement ordonné), une paire (P, Q) de tableaux bidimensionnels. Ces tableaux, dits Tableaux de Young satisfont les propriétés suivantes :
- P et Q ont la même *forme*, c'est-à-dire qu'ils possèdent le même nombre de lignes et chaque ligne de P contient le même nombre de cases que la ligne correspondante de Q. De plus, les longueurs des lignes sont (faiblement) décroissantes ;
- P contient les lettres $a_1, a_2, \ldots, a_n$ alors que Q contient les entiers $1, 2, \ldots, n$;
- les éléments sont disposés en ordre faiblement croissant sur les lignes et strictement croissant sur les colonnes.

Dans l'algorithme d'insertion on suppose l'existence, dans les tableaux manipulés, d'une valeur « sentinelle » permettant de représenter le fait qu'une case est « vide », on a choisit la valeur 0.

Remarque : Attention, car en machine, une case n'est **jamais** vide, si une case existe elle contient obligatoirement quelque chose, il faut donc utiliser des valeurs particulières pour représenter des propriétés « logiques », ici nous avons choisi que si la case contient 0 on la dira vide, et que toute autre valeur différente de 0 représentera une valeur exploitable.



(a) L'ajout de 90 déséquilibre l'arbre

(b) Après rotation au niveau de 75

FIGURE 13 – Le rééquilibrage d'un ABR.

```
Algo Insertion_RSK(i,x:entiers,P,Q:tableaux[1..n+1,1..n+1] d'entiers,l:entier)
DÉBUT
    c, v:entiers;
    c := 1;
    (*on cherche l'emplacement possible de x sur la ligne l*)
    tantque ( (P[l,c]<>0) et (P[l,c]<=x) ) faire
        c := c+1;
    fintantque
    (*si la place est vide*)
    si ( P[l,c]=0 ) alors
        P[l,c] := x;
        Q[l,c] = i;
    sinon
        v := P[l,c];
        P[l,c] := x;
        Insertion_RSK(i,v,P,Q,l+1);
FIN


Algo RSK(A:tableau[1..n] d'entiers)
DÉBUT
    P,Q:tableaux[1..n+1,1..n+1] d'entiers;
    (*on initialise les tableaux P et Q*)
    pour i de 1 à n+1 faire
        pour j de 1 à n+1 faire
            P[i,j] := 0;
            Q[i,j] := 0;
        finpour
    finpour
    (*insertion de tous les éléments du tableau A*)
    pour i de 1 à n faire
```

```
        Insertion_RSK(i,A[i],P,Q,1);
    finpour
    (*affichage des résultats*)
    Affichage(P);
    Affichage(Q);
FIN


Algo Suppression_RSK(P,Q:tableaux[1..n+1,1..n+1],n:entier):entier
DÉBUT
    l,c,v:entiers
    l := 1;
    c := 1;
    (*on recherche n dans le tableau, on SAIT qu'il est y est *)
    tantque ( Q[l,c]<>n ) faire
        c := c+1;
        si ( Q[l,c]=0 ) alors
            l := l+1;
            c := 1;
    fintantque
    (*on enlève n du tableau Q*)
    Q[l,c] := 0;
    (*on enlève son équivalent dans P en conservant la valeur*)
    v := P[l,c];
    P[l,c] := 0;
    l := l-1;
    (*on redescend vers le bas jusqu'au bout*)
    tantque (l >0) faire
        (*pour un ligne donnée on recherche la place de l'élément poussé depuis le haut*)
        c := 1;
        tant que ( (P[l,c]<>0) et (P[l,c]<v) ) faire
            c := c+1;
        fintantque
        (*l'élément à pousser est en colonne précédente*)
        c := c-1;
        Échanger(P[l,c],v);
        l := l-1; (*on recommence plus bas*)
    fintantque
    retourner (v);
FIN
```

5 La programmation dynamique

Il s'agit non pas d'un algorithme mais d'une stratégie permettant d'obtenir des algorithmes efficaces. Prenons l'exemple du calcul des cœfficients binomiaux :

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k} + \binom{n-1}{k-1} \\ 1 \text{ si } k = 0 \text{ ou } k = n \\ 0 \text{ si } k < 0 \text{ ou } k > n \end{cases}$$

Cette définition conduit à l'écriture naturelle de l'algorithme suivant ;

```
Algo Binome(n,k:entier):entier
DÉBUT
    si ( (k=0) ou (k=n) ) alors
        retourne (1);
    si ( (k<0) ou (k>n) ) alors
        retourner (0);
    sinon
        retourner (Binome(n-1,k) + Binome(n-1,k-1);
FIN
```

Le problème est que cet algorithme est très (trop) lent, et en plus nécessite beaucoup de mémoire. Montrons que dans le pire des cas le calcul de $\binom{n}{k}$ par cet algorithme est en $\mathcal{O}(2^n)$.
On compte le nombre d'appels récursifs à `Binome`.

	nombre d'appels pour i, j								somme
$\vdots$	$\vdots$								
$\frac{n}{2}$	$\cdots$	1	$\cdots$					1	$2^{\frac{n}{2}} \in \mathcal{O}(2^n)$
$\vdots$	$\vdots$								
n-3	$\cdots$		$\cdots$		1	3	3	1	8
n-2	$\cdots$		$\cdots$			1	2	1	4
n-1	$\cdots$		$\cdots$				1	1	2
n	$\cdots$		$\cdots$					1	1
	$\cdots$		$\cdots$	k-4	k-3	k-2	k-1	k	

On constate en remplissant ce tableau que de très nombreux calculs sont effectués plusieurs fois au cours du calcul. La stratégie « programmation dynamique » consiste simplement à éviter de refaire ces calculs déjà faits en enregistrant le résultat de tout calcul effectué une première fois afin de le retrouver sans effectuer la procédure lorsque nécessaire. Cela conduit à diminuer la complexité en temps mais au prix d'un espace mémoire supplémentaire.
La version « dynamique » de l'algorithme devient :

```
Algo Binome_dynamique(n,k:entiers):entiers
DÉBUT
    B:tableau[0..n][0..k] d'entiers;
    B[0,0] := 1;
    pour i de 1 à n faire
        B[i,0] := 1;
        pour j de 1 à k-1 faire
            B[i,j] := B[i-1][j]+B[i-1][j-1];
        finpour
        B[i,i] := 1;
    finpour
    retourner (B[n,k]);
FIN
```

Pour la suite de Fibonnaci l'algorithme naïf est :

```
Algo Fibonacci(n:entier):entier
DÉBUT
    si (n=0) alors
        retourne (1);
    si (n=1) alors
```

```
        retourne (1);
    retourne (Fibonnaci(n-1)+Fibonnaci(n-2));
FIN
```

Même problème que dans le cas précédent, cet algorithme effectue un nombre d'appels de l'ordre de $Fibonnaci(n)$. Sa version dynamique s'écrit :

```
Algo Fibonacci_dynamique(n:entier):entier
DÉBUT
    F:tableau[1..n] d'entier;
    F[0] := 1;
    F[1] := 1;
    pour i de 2 à n faire
        F[i] := F[i-1]+F[i-2];
    finpour
    retourne (F[n]);
FIN
```

6 Le « Master Theorem » pour la résolution des récurrences

Théorème 1 (Master Theorem) *Soit* $T(n) = aT(\frac{n}{b}) + f(n)$ *:*
– *si* f *est « petit » (*f *est dominé par* $n^{\log_b a}$*) alors* $T(n) \in \Theta(n^{\log_b a})$
– *si* f *est « moyen » (*$f \in \Theta(n^{\log_b a})$*) alors* $T(n) \in \Theta(n^{\log_b a} \log n)$
– *si* f *est « grand » (*f *domine strictement* $n^{\log_b a}$*) alors* $T(n) \in \Theta(f(n))$

Ce type de récurrence intervient souvent dans le calcul de la complexité d'algorithmes utilisant la stratégie du « diviser pour régner ». Dans ce type de cas, dans l'équation :
– a représente le nombre de sous-problèmes ;
– $\frac{n}{b}$ représente la taille des données des sous-problèmes ;
– $f(n)$ représente la complexité de l'opération qui reconstruit la solution totale à partir des solutions partielles.

Exemple : Pour le tri fusion on a l'équation $T(n) = 2T(\frac{n}{2}) + cn$. L'application du Master Theroem et l'observation du fait que l'on se trouve dans le cas moyen ($cn \in \Theta(n^{\log_b a}) \log n$) donne immédiatement $T(n) \in \Theta(n. \log n)$.

La multiplication des nombres

La multiplication naïve des nombre (telle qu'on l'a apprise à l'école élémentaire) est un algorithme.

Exercice 10 *Écrire un algorithme permettant de construire le résultat de la multiplication de deux nombres de n chiffres. Chaque nombre de n chiffres étant représenté par un tableau de taille n, dans lequel la case i représente le chiffre* $n - i + 1$.

Si A et B sont deux nombres de taille n, on peut découper ces deux nombres de la façon suivante : $A : A_h A_b$ tel que $A = A_h.2^{\frac{n}{2}} + A_b$ et $B : B_h B_b$ tel que $B = B_h.2^{\frac{n}{2}} + B_b$
Le produit de A par B s'exprime donc de la façon suivante

$$A.B = (A_h.2^{\frac{n}{2}} + A_b).(B_h.2^{\frac{n}{2}} + B_b)$$

donc

$$A.B = A_h.B_h.2^n + (A_h B_b + A_b B_h)2^{\frac{n}{2}} + A_b.B_b$$

La complexité vérifié l'équation

$$T(n) = 4.T(\frac{n}{2}) + cn$$

car le produit nécessite 4 produits de nombres de taille $\frac{n}{2}$ et quelques additions de complexité linéaire.
La complexité est obtenue par le Master Theorem :

$$T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$$

Pour gagner en performances on peut faire l'observation suivante (découverte par Karatsuba et Ofman) :

$$A.B = (A_h B_h)2^n + ((A_h + B_b).(A_b + A_h) - A_h B_h - A_b B_b)2^{\frac{n}{2}} + A_b B_b$$

où l'on observe qu'il n'y a que trois produits. La complexité s'exprime alors par l'équation

$$T(n) = 3.T(\frac{n}{2}) + cn$$

Par le Master Theorem en observant que f est $\ll$ petit $\gg$, on a $T(n) \in \Theta(n^{\log_2 3})$.

Exercice 11 *Écrire la multiplication de nombres (représentés de la même façon que dans l'exercice 10) en utilisant l'astuce de Karatsuba-Ofman.*

La multiplication de matrices

Soient A et B deux matrices carrées de taille n. Supposons que $n = 2^k$. On souhaite calculer le produit $C = A.B$. Décomposons les matrices en sous-matrices de taille $\frac{n}{2}$ de la façon suivante :

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Alors le produit s'exprime par la relation de récurrence suivante :

$$\begin{cases} C_{11} = A_{11}B_{11} + A_{12}B_{21} \\ C_{12} = A_{11}B_{12} + A_{12}B_{22} \\ C_{21} = A_{21}B_{11} + A_{22}B_{21} \\ C_{22} = A_{21}B_{12} + A_{22}B_{22} \end{cases}$$

La complexité est exprimée par la relation de récurrence

$$T(n) = 8.T(\frac{n}{2}) + cn^2$$

car le produit est obtenu par le calcul de 8 produit de sous-matrices de taille $\frac{n}{2}$ et des additions de matrices dont l'algorithme est linéaire.
La complexité de l'algorithme naïf est obtenue en utilisant le Master Theorem :

$$T(n) \in \Theta(n^3)$$

Exercice 12 *Écrire l'algorithme de multiplication de matrices en utilisant la décomposition précédente.*

Mais Strassen a obtenu une amélioration en posant :

$$\begin{cases} M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 = (A_{21} + A_{22})B_{11} \\ M_3 = A_{11}(B_{12} - B_{22}) \\ M_4 = A_{22}(B_{21} + B_{11}) \\ M_5 = (A_{11} + A_{22})B_{22} \\ M_6 = (A_{21} + A_{11})(B_{11} + B_{12}) \\ M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) \end{cases}$$

puis en remarquant que les sous-matrices $C_{11}, C_{12}, C_{21}, C_{22}$ peuvent s'exprimer comme simple somme de certains M_i.

Exercice 13 *Retrouver les sommes de M_i permettant d'obtenir les sous-matrices C_{ij}.*

De la sorte, on obtient les sous-matrices C en utilisant seulement 7 produits de sous-matrices et non plus 8. La complexité vérifie donc l'expression

$$T(n) = 7.T(\frac{n}{2}) + cn^2$$

En utilisant le Master Theorem, on obtient

$$T(n) \in \Theta(n^{\log_2 7})$$

7 Solutions des exercices

Exercice 1, page 3

Montrer que $\log_2 n \in \Theta \log_3 n$.
Il est évident que $\log_3 n \in \mathcal{O}(\log_2 n)$ puisque $\forall n \geq 1$, $\log_3 n < \log_2 n$ (on prend $n_0 = 1$ et $c = 1$).
Dans l'autre sens, $\forall n \geq 1$, $\log_2 n = \log_2 3. \log_3 n$ donc $\forall n \geq 1$, $\log_2 n < (\log_3 n + 1) \log_3 n$ (on prend $n_0 = 1$ et $c = (\log_3 n + 1)$.

Exercice 2, page 3

$2^n \in \mathcal{O}(3^n)$ puisque en prenant $n_0 = 1$ et $c = 1$ on a $\forall n > 1$, $2^n < 3n$.
On va montrer que $3^n \notin \mathcal{O}(2^n)$ par l'absurde. Si $\forall n > n_0 \exists c$, $3^n < c.2^n$ on aurait $\forall n > n_0$, $(\frac{3}{2})^n < c$ ce qui est faux.

Théorème 2 *Si* $\lim_{n\to\infty} \frac{f(x)}{g(x)}$ *existe alors*
- *si cette limite est* ∞ *alors* $g \in \mathcal{O}(f)$ *et* $f \notin \mathcal{O}(g)$
- *si cette limite est 0 alors* $f \in \mathcal{O}(g)$ *et* $g \notin \mathcal{O}(f)$
- *si cette limite est* $l \neq 0$ *alors* $f \in \mathcal{O}(g)$ *et* $g \in \mathcal{O}(f)$

Exercice 3, page 5

	Échanges	Comparaisons	Affectations (`position-du-minimum := j`)
Meilleur cas	$n-1$	$\frac{n(n-1)}{2}$	0 (tableau trié croissant)
Pire cas	$n-1$	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$ (tableau trié décroissant)

Exercice 4, page 8

Une idée possible est la suivante.

On place d'abord le pivot en fin de la partition (pour cela on peut l'échanger avec n'importe quelle autre valeur).

Ensuite, il suffit de s'assurer qu'à tout moment le reste du tableau (tous sauf le pivot) est découpé en deux parties (on notera l'indice de ce découpage j) : à gauche de j on a les éléments déjà « partitionnés », à droite de j les élément à « partitionner ». La partie à gauche de j est ultérieurement découpée en deux parties (l'indice de ce découpage est noté i), la première (à gauche de i) avec des éléments tous plus petit que le pivot et la seconde (à droite de i) avec des éléments tous plus grands ou égaux au pivot. Les éléments à droite de j sont encore en vrac. À chaque tour, il suffit de piocher un élément dans la partie en vrac (le premier, donc celui d'indice j), puis de l'insérer dans la bonne partie en partie gauche. Si l'élément est plus petit que le pivot, il suffit de le placer en position i (et dans ce cas, déplacer i vers la droite).

A la fin, un simple échange entre $A[f]$ et $A[i]$ permet de placer le pivot à sa position définitive.

```
Algo Partition(A:tableau[1..n] d'entiers; d,f,p:entiers):entier
DÉBUT
    Échanger(A[p],A[f]); (*Le pivot est placé à la fin*)
    (*à gauche de i est <pivot et tout ce qui est à droite jusqu'à j est >=pivot*)
    i := d;
    pour j de d à f-1 faire
        si (A[j]<A[f]) alors
            Échanger(A[i],A[j])
            i := i+1
```

```
    finpour
    Échanger(A[i],A[f]);
    retourner (i);
FIN
```

Voici l'extrait d'une exécution de l'algorithme `Partition`

$\vdots$

3	7 i	8	5	2 j	1	9	5	4 p	en j c'est un élément $<$ pivot
3	2	8 i	5	7	1 j	9	5	4 p	en j c'est un élément $<$ pivot
3	2	1	5 i	7	8	9 j	5	4 p	en j c'est un élément $\geq$ pivot
3	2	1	5 i	7	8	9	5 j	4 p	en j c'est un élément $\geq$ pivot
3	2	1	4 i	7	8	9	5 j	5 p	placement final du pivot

Exercice 5, page 9

$\log(n!) = \log(1.2.3.4.\cdots.n)$
Pour n pair on a

$$\begin{array}{rcl} \log(n!) & = & \log(1.n) + \log(2(n-1)) + \log(3(n-2)) + \cdots + \log(\frac{n}{2}\frac{n-1}{2}) \\ & \geq & \frac{n}{2}\log(1.n) \\ & = & \frac{n}{2}\log n \end{array}$$

Pour n impair on a

$$\begin{array}{rcl} \log(n!) & = & \log(1.n) + \log(2(n-1)) + \log(3(n-2)) + \cdots + \log(\frac{n-1}{2}(\frac{n-1}{2}+2)) + \log\frac{n+1}{2} \\ & \geq & \frac{n-1}{2}\log(1.n) + \log\frac{n+1}{2} \\ & = & \frac{n}{2}\log n - \log\sqrt{n} + \log\frac{n+1}{2} \\ & \geq & \frac{n}{2}\log n \end{array}$$

Exercice 7, page 10

`Tri_insertion` est stable car on stoppe les échanges dès que `A[i]<=A[i-1]`, donc en cas d'égalité l'ordre des éléments est préservé et deux éléments de même valeur ne sont jamais échangés.

`Tri_selection` n'est pas stable car dans le cas où $x < y$ et que la partie non triée est par exemple de la forme $y_1, y_2, \cdots, x, \cdots$ (où y_1 représente la première occurrence de y et y_2 la deuxième). La sélection produira $x, y_2, \cdots, y_1, \cdots$ (on sélectionne x et on l'échange avec le premier élément de la partie non triée) ainsi l'ordre relatif des deux y n'aura pas été préservé.

`Tri_selection` n'est stable qu'à condition que la procédure de `Fusion` fusionne (recopie) d'abord les éléments de la partie gauche en cas d'égalité des éléments à fusionner. Le test dans la procédure de fusion devrait être `if (G[i]<=D[j]`.

`Tri_rapide` n'est pas stable car dans le cas de l'Exercice 4, où le pivot est 4, on effectue à un échange (entre 4 et 5) qui ne préserve pas l'ordre relatif des deux 5.

Exercice 8, page 14

Le parcours préfixe de l'arbre en question visite dans l'ordre les nœuds suivants : 8, 9, 1, 3, 10, 4, 5, 11, 12, 6, 2, 7.
Le parcours postfixe est : 1, 3, 5, 11, 4, 10, 9, 2, 6, 7, 12, 8.

Exercice 9, page 16

Rappel : le type tas est un type structuré en deux champs `contenu` (un tableau) et `taille` (un entier).

```
Algo Suppression_tas(T:tas):element
DÉBUT
    valeur : élément;
    valeur := T.contenu[1]; (*on récupère le plus petit*)
    T.contenu[1] = T.contenu[T.taille]; (*on déplace le dernier*)
    T.taille := T.taille-1; (*on supprime un élément donc la taille diminue*)
    i := 1; (*position initiale de la descente/percolation*)
    y := T.contenu[1];
    (*considérons d'abors les cas où le noeud i a deux fils*)
    tant que ( (2*i+1<=T.taille et (y>min(T.contenu[2*i],T.contenu[2ï+1]) ) faire
      si (T.contenu[2*i]<T.contenu[2*i+1]) alors
        Échanger(T.contenu[2*i],T.contenu[i];
        i = 2*i;
      sinon
        Échanger(T.contenu[2*+1i],T.contenu[i];
        i = 2*i+1;
    fintantque
    (*il reste le cas où i est un noeud avec un seul fils*)
    si ( (2*i<=T.taille) et (y<T.contenu[T.taille]) ) alors
        Échanger(T.contenu[i],T.contenu[T.taille]);
    retourner (valeur);
FIN
```

Exercices supplémentaires

1. Pour chacune des paires de fonctions suivantes dire si $f \in \mathcal{O}(g)$ et si $g \in \mathcal{O}(f)$:
   - $f(n) = \log(2n)$ et $g(n) = \log n^3$
   - $f(n) = 2^{\frac{n}{2}}$ et $g(n) = n^3$
2. Évaluer la complexité de l'algorithme suivant :

```
Algo Machin(T:tableau[1..n] d'entiers)
DÉBUT
    k,i,j,x:entiers;
    pour i de 1 à n faire
        pour j de 1 à i faire
             pour k de 1 à n-j faire
                (*ici n'importe quelle opération en temps constant*)
            finpour
        finpour
    finpour
 FIN
```

3. Calculer à l'aide d'un algorithme le nombre de manières de gravir un escalier de m marches (partir de la première et arriver pile sur la dernière) avec des sauts de hauteurs choisies parmi $\{a_1, \cdots, a_p\}$. Pour commencer on peut faire le calcul en se limitant à s sauts (plus simple). On remarquera trivialement que pour une suite de sauts donnés si $m = s_1 + s_2 + \ldots$ alors $m - s_1 = s_2 + \ldots$

Index