

# Algorithmique et arbres

Pierre Boudes  
creative common<sup>1</sup>

31 mars 2011

1. Cette création est mise à disposition selon le Contrat Paternité-Pas d'Utilisation Commerciale-Partage des Conditions Initiales à l'Identique 2.0 France disponible en ligne <http://creativecommons.org/licenses/by-nc-sa/2.0/fr/> ou par courrier postal à Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

## **Première partie**

# **Écriture et comparaison des algorithmes, tris**

# Chapitre 1

## Introduction



### 1.1 La notion d'algorithme

Un algorithme est un procédé permettant l'accomplissement mécanique d'une tâche assez générale. Cette tâche peut être par exemple la résolution d'un problème mathématique ou, en informatique, un problème d'organisation, de structuration ou de création de données. Un algorithme est décrit par une suite d'opérations simples à effectuer pour accomplir cette tâche. Cette description est finie et destinée à des humains. Elle ne doit pas produire de boucles infinies : étant donnée une situation initiale (une instance du problème, une donnée particulière), elle doit permettre d'accomplir la tâche en un nombre fini d'opérations.

Quelques exemples de problèmes pour lesquels on utilise des algorithmes : rechercher un élément dans une liste, trouver le plus grand commun diviseur de deux nombres entiers, calculer une expression algébrique, compresser des données, factoriser un nombre entier en nombres premiers, trier un tableau, trouver l'enveloppe convexe d'un ensemble de points, créer une grille de sudoku, etc.

Le problème résolu ou la tâche accomplie doivent être assez généraux. Par exemple, un algorithme de tri doit être capable de remettre dans l'ordre n'importe quelle liste d'éléments deux à deux comparables (algorithme généraliste) ou être conçu pour fonctionner sur un certain type d'éléments correspondant à des données usuelles (le type `int` du C, par exemple). Contre-exemple : le tri *chanceux*. Cet algorithme rend la liste passée en argument si celle-ci est déjà triée et il ne rend rien sinon. Autre contre-exemple : si on

se restreint au cas où la liste à trier sera toujours la liste des entiers de 0 à  $n - 1$  dans le désordre (une *permutation* de  $\{0, \dots, n - 1\}$ ) alors nul besoin de trier, il suffit de lire la taille  $n$  de la liste donnée en entrée et de rendre la liste  $0, \dots, n - 1$  sans plus considérer l'entrée. Il serait incorrect de dire de ce procédé qu'il est un algorithme de tri.

Un algorithme doit toujours terminer en un temps fini, c'est à dire en un nombre fini d'étapes, toutes de temps fini. Contre-exemple : tri *bogo* (encore dénommé tri stupide). Ce tri revient, sur un jeu de cartes, à les jeter en l'air, à les ramasser dans un ordre quelconque puis à vérifier si les cartes sont dans le bon ordre, et si ça n'est pas le cas, à recommencer. Cet algorithme termine presque sûrement mais il est toujours possible qu'il ne termine jamais (de même qu'un singe tapant à la machine aléatoirement pendant suffisamment de temps produira presque sûrement une série de pages contenant l'ensemble de l'œuvre de Shakespeare).

En général, un algorithme est *déterministe* : son exécution ne dépend que des entrées (ce n'est pas le cas du tri *bogo*).

En particulier, un algorithme déterministe réalise une fonction (au sens mathématique) : il prend une entrée et produit une sortie qui ne dépend que de l'entrée.

Toutefois, pour une même fonction mathématique il peut y avoir plusieurs algorithmes, parfois très différents. Il y a par exemple plusieurs algorithmes de tri, qui réalisent tous la fonction « ordonner les éléments d'un tableau ».

Nous verrons également des algorithmes *randomisés* (i.e. à méthodes aléatoires) qui ne sont pas déterministes. Toutefois la seule part de hasard dans l'exécution se ramènera à une modification de l'entrée sans conséquence sur la justesse du résultat. Par exemple, un algorithme de tri randomisé désordonne la liste d'éléments qu'on lui donne à trier avant de se lancer dans le tri.

## 1.1.1 Algorithmes et programmes

Dans ce cours, les algorithmes sont écrits en pseudo-code ou en langage C, comme des programmes.

Bien que les deux notions aient des points communs, il ne faut toutefois pas confondre algorithme et programme.

1. Un programme n'est pas forcément un algorithme. Un programme ne termine pas forcément de lui-même (c'est souvent la personne qui l'utilise qui y met fin). Un programme n'a pas forcément vocation à retourner un résultat. Enfin un programme est bien souvent un assemblage complexe qui peut employer de nombreux algorithmes résolvants des problèmes très variés.

2. La nature des algorithmes est plus mathématique que celle des programmes et la vocation d'un algorithme n'est pas forcément d'être exécutée sur un ordinateur. Les humains utilisaient des algorithmes bien avant l'ère de l'informatique et la réalisation de calculateurs mécaniques et électroniques. En fait, les algorithmes ont été au cœur du développement des mathématiques (voir [CEBMP<sup>+</sup>94]). Pour autant, ce cours porte sur les algorithmes pour l'informatique.

### 1.1.2 Histoire

Le terme algorithme provient du nom d'un mathématicien persan du IX<sup>e</sup> siècle, Al Kwarizmi (Abou Jafar Muhammad Ibn Mūsa al-Khuwārizmi) à qui l'ont doit aussi : l'introduction des chiffres indiens (communément appelés chiffres arabes) ainsi que le mot algèbre (déformation du titre d'un de ses ouvrages). Son nom a d'abord donné algorithme (via l'arabe) très courant au moyen-âge. On raconte que la mathématicienne Lady Ada Lovelace fille de Lord Byron (le poète) forgea la première le mot algorithme à partir d'algorithme. En fait il semble qu'elle n'ait fait qu'en systématiser l'usage. Elle travaillait sur ce qu'on considère comme le premier programme informatique de l'histoire en tant qu'assistante de Charles Babbage dans son projet de réalisation de machines différentielles (ancêtres de l'ordinateur) vers 1830. Le langage informatique Ada (1980, Défense américaine) a été ainsi nommé en hommage à la première informaticienne de l'histoire. Son portrait est aussi sur les hologrammes des produits Microsoft.

L'utilisation des algorithmes est antérieure : les babyloniens utilisaient déjà des algorithmes numériques (1600 av. JC).

En mathématiques le plus connu des algorithmes est certainement l'algorithme d'Euclide (300 av. JC) pour calculer le pgcd de deux nombres entiers.

« Étant donnés deux entiers naturels  $a$  et  $b$ , on commence par tester si  $b$  est nul. Si oui, alors le P.G.C.D. est égal à  $a$ . Sinon, on calcule  $c$ , le reste de la division de  $a$  par  $b$ . On remplace  $a$  par  $b$ , et  $b$  par  $c$ , et on recommence le procédé. Le dernier reste non nul est le P.G.C.D. » (wikipedia.fr).

Un autre algorithme très connu est le crible d'Ératosthène (III<sup>e</sup> siècle av. JC) qui permet de trouver la liste des nombres premiers plus petits qu'un entier donné quelconque. On écrit la liste des entiers de 2 à  $N$ . (i) On sélectionne le premier entier (2) on l'entoure d'un cercle et on barre tous ses multiples (on avance de 2 en 2 dans la liste) sans les effacer. (ii) Lorsqu'on a atteint la fin de la liste on recommence avec le premier entier  $k$  non cerclé et non barré : on le cerclé, puis on barre les multiples de  $k$  en progressant de  $k$  en  $k$ . (iii) On

recommence l'étape (ii) tant que  $k^2 < N$ . Les nombres premiers plus petits que  $N$  et supérieurs à 1 sont les éléments non barrés de la liste.

Il y a aussi le *pivot de Gauss* (ou de Gauss-Jordan) qui est en fait une méthode bien antérieure à Gauss. Un mathématicien Chinois, Liu Hui, avait déjà publié la méthode dans un livre au IIIe siècle.

Mais la plupart des algorithmes que nous étudierons datent d'après 1945. Date à laquelle John Von Neumann introduisit ce qui est sans doute le premier programme de tri (un tri fusion).

## 1.2 Algorithmique

L'*algorithmique* est l'étude mathématique des algorithmes. Il s'agit notamment d'étudier les problèmes que l'on peut résoudre par des algorithmes et de trouver les plus appropriés à la résolution de ces problèmes. Il s'agit donc aussi de comparer les algorithmes, et de démontrer leurs propriétés.

La nécessité d'étudier les algorithmes a été guidée par le développement de l'informatique. Ainsi l'algorithmique est une activité jeune qui s'est développée dans la deuxième moitié du XXe siècle, principalement à partir des années 60-70 avec le travail de Donald E. Knuth [Knu68, Knu69, Knu73]. Actuellement, un très bon livre de référence en algorithmique est le *Cormen* [CLRS02].

Parmi les critères de comparaison entre algorithmes, les plus déterminants d'un point de vue informatique sont certainement les consommations en temps et en espace de calcul. Nous nous intéresserons particulièrement à l'expression des coûts en temps et en espace à l'aide de la *notation asymptotique* qui permet de donner des ordres de grandeur indépendamment de l'ordinateur sur lequel l'algorithme est implanté.

À côté des études de coûts, les propriétés que nous démontrerons sur les algorithmes sont principalement la *terminaison* : l'algorithme termine ; et la *correction* : l'algorithme résout bien le problème donné. Pour cela une notion clé sera celle d'*invariant de boucle*.

### 1.2.1 La notion d'invariant de boucle

Souvent un algorithme exécute une boucle pour aboutir à son résultat. Un invariant de boucle est une propriété telle que :

**initialisation** elle est vraie avant la première itération de la boucle ;

**conservation** si elle est vérifiée avant une itération quelconque de la boucle elle le sera encore avant l'itération suivante ;

**terminaison** bien entendu il faut aussi que cette propriété soit utile à quelque chose, à la fin. La dernière étape consiste donc à établir une propriété intéressante à partir de l'invariant, en sortie de boucle.

La notion d'invariant de boucle est à rapprocher de celle de raisonnement par récurrence (voir exercice 20). Il s'agit d'une notion clé pour démontrer les propriétés d'un algorithme. Souvent, la propriété invariante est étroitement liée à l'idée même à l'origine de l'algorithme et l'invariant est construit en fonction du but de l'algorithme, au moment de sa mise au point.

## Invariant et tablette de chocolat

En guise de récréation, voici un exemple de raisonnement utilisant un invariant de boucle, pris en marge de l'algorithmique. Il s'agit d'un jeu, pour deux joueurs, le Joueur et l'Opposant. Au départ, les deux joueurs disposent d'une tablette de chocolat rectangulaire dont un des carrés au coin a été peint en vert. Tour à tour chaque joueur découpe la tablette entre deux rangées et mange l'une des deux moitiés obtenues. L'objectif est de ne pas manger le carré vert. Joueur commence. Trouver une condition sur la configuration de départ et une stratégie pour que Joueur gagne à tous les coups.

On peut coder ce problème comme un problème de programmation. On représente la tablette comme un couple d'entiers non nuls  $(p, q)$ . la position perdante est  $(1, 1)$ . On considère un tour complet de jeu (Joueur joue puis Opposant joue) comme une itération de boucle. Schématiquement, une partie est l'exécution d'un programme :

```
32 main(){
33     init();
34     while(1){ /* <----- Boucle principale */
35         arbitre("Joueur"); /* Faut-il déclarer Joueur perdant ? */
36         Joueur(); /* Sinon Joueur joue. */
37         arbitre("Opposant"); /* Faut-il déclarer Opposant perdant ? */
38         Opposant(); /* Sinon Opposant joue. */
39     }
40 }
```

où  $p$  et  $q$  sont deux variables globales, initialisées par une fonction appropriée `init()` en début de programme.

L'arbitre est une fonction qui déclare un joueur perdant et met fin à la partie si ce joueur reçoit la tablette  $(1, 1)$ .

```
11 arbitre(char *s){
12     if ( ( p == 1 ) && ( q == 1 ) ) {
13         printf("%s a perdu !", s);
14         exit(0); /* <----- fin de partie */
```

```

15     }
16 }

```

On peut supposer que Opposant joue au hasard, sauf lorsqu'il gagne en un coup.

```

18 opposant(){
19     if (p == 1) q = 1;      /* si p == 1 opposant gagne en un coup */
20     else if (q == 1) p = 1; /* de même si q == 1 */
21     else if ( random() % 2 ) /* Opposant choisit p ou q au hasard */
22         p == random() % (p - 1) + 1; /* croque un bout de p */
23     else q == random() % (q - 1) + 1; /* croque un bout de q */
24 }

```

L'objectif est de trouver une condition de départ et une manière de jouer pour le Joueur qui le fasse gagner contre n'importe quel Opposant. C'est ici qu'intervient notre invariant de boucle : on cherche une condition sur  $(p, q)$  qui, si elle est vérifiée en début d'itération de la boucle principale, le sera encore à l'itération suivante, et, bien sûr, qui permette à Joueur de gagner en fin de partie.

La bonne solution vient en trois remarques :

- la position perdante est une tablette carrée ( $p = q = 1$ );
- si un joueur donne une tablette carrée ( $p = q$ ) à l'autre, cet autre rend obligatoirement une tablette qui n'est pas carrée ( $p \neq q$ );
- lorsque qu'un joueur commence avec une tablette qui n'est pas carrée, il peut toujours la rendre carrée.

Il suffit donc à Joueur de systématiquement rendre la tablette carrée avant de la passer à Opposant. Dans ce cas, quoi que joue Opposant, celui-ci retourne une tablette qui n'est pas carrée à Joueur, et ce dernier peut ainsi continuer à rendre la tablette carrée.

Avec cette stratégie pour Joueur, l'invariant de boucle est : la tablette n'est pas carrée. Par les remarques précédents, l'invariant est préservé. Par ailleurs, Joueur ne perd jamais puisqu'il ne peut pas recevoir de tablette carrée. C'est donc bien que Opposant perd.

Il manque l'initialisation de l'invariant. Si la tablette n'est pas carrée au départ, il est vrai et Joueur gagne contre n'importe quel opposant. Par contre, si la tablette de départ est carrée, il suffit qu'Opposant connaisse la stratégie que nous venons de décrire pour gagner. Donc en partant d'une tablette carrée, il est possible que Joueur perde. Ainsi, nous avons trouvé une condition nécessaire et suffisante – le fait que la tablette ne soit pas carrée au départ – pour gagner à tous les coups au jeu de la tablette de chocolat.

Voici le code pour Joueur.

```

26  joueur(){
27      if (p > q) p = q;          /* Si la tablette n'est pas carrée */
28      else if (q > p) q = p;    /* rend un carré. */
29      else p--;                /* Sinon, gagner du temps ! */
30  }

```

En résumé on a trouvé une propriété qui est préservée par le tour de jeu (un invariant) et qui permet à Joueur de gagner. Ce type de raisonnement s'applique à d'autres jeux mais trouver le bon invariant est souvent difficile.

### Invariant de boucle et récurrence, un exemple

La notion d'invariant de boucle dans un programme itératif est l'équivalent de celle de propriété montrée par récurrence dans un programme récursif.

Considérons deux algorithmes différents, l'un itératif, l'autre récursif, pour calculer la fonction factorielle.

<b>Fonction Fact(<math>n</math>)</b>	/* Fonction fact en C */
<b>si</b> $n = 0$ <b>alors</b>   <b>retourner</b> 1; <b>sinon</b>   <b>retourner</b> $n \times \text{Fact}(n - 1)$ ;	unsigned int fact(unsigned int n){ if (n == 0) return 1; return n * fact(n - 1); }

Figure 1.1 – Factorielle récursive en pseudo-code et en C

Pour montrer que la version récursive calcule bien factorielle on raisonne par récurrence. C'est vrai pour  $n = 0$  puisque  $0! = 1$  et que  $\text{Fact}(0)$  renvoie 1. Supposons que c'est vrai jusqu'à  $n$ . Alors  $\text{Fact}(n + 1)$  renvoie  $(n + 1) \times \text{Fact}(n)$ . Par hypothèse de récurrence  $\text{Fact}(n)$  renvoie  $n!$ , donc  $\text{Fact}(n + 1)$  renvoie  $(n + 1) \times n!$  qui est bien égal à  $(n + 1)!$ .

Pour la version itérative on pose l'invariant de boucle : au début de la  $k$ -ième étape de boucle  $r = (k - 1)!$ . Initialisation : à la première étape de boucle  $r$  vaut 1 et  $j$  prend la valeur 1, l'invariant est vrai ( $(1 - 1)! = 1$ ). Conservation : supposons que l'invariant est vrai au début de la  $k$ -ième étape de boucle, on montre qu'il est vrai au début de la  $k + 1$ -ième étape. À la  $k$ -ième étape,  $j = k$  et  $r$  prend la valeur  $r \times j$  mais  $r = (k - 1)!$  donc en sortie de cette étape  $r = (k - 1)! \times j = k!$ . Ainsi au début de la  $k + 1$ -ième étape  $r$  vaut bien  $k!$ . Terminaison : la boucle s'exécute  $n$  fois, c'est à dire jusqu'au début de la  $n + 1$ -ième étape, qui n'est pas exécutée. Donc en sortie de boucle  $r = n!$  et comme c'est la valeur renvoyée par la fonction, l'algorithme est correct.

<b>Fonction</b> Fact( <i>n</i> )	/* Fonction fact en C */
<i>r</i> = 1;	unsigned int fact(unsigned int n){
<b>pour</b> <i>j</i> = 1 à <i>n</i> <b>faire</b>	int j, r = 1;
<i>r</i> = <i>r</i> × <i>j</i> ;	for (j = 1; j <= n; j++){
<b>retourner</b> <i>r</i> ;	r = r * j;
	}
	return r;
	}

Figure 1.2 – Factorielle itérative en pseudo-code et en C

## 1.2.2 De l’optimisation des programmes

Les programmes informatiques s’exécutent avec des ressources limitées en temps et en espace mémoire. Il est courant qu’un programme passe un temps considérable à effectuer une tâche particulière correspondant à une petite portion du code. Il est aussi courant qu’à cette tâche correspondent plusieurs algorithmes. Le choix des algorithmes à utiliser pour chaque tâche est ainsi très souvent l’élément déterminant pour le temps d’exécution d’un programme. L’optimisation de la manière dont est codé l’algorithme ne vient qu’en second lieu (quelles instructions utiliser, quelles variables stocker dans des registres du processeur plutôt qu’en mémoire centrale, etc.). De plus, cette optimisation du code est en partie prise en charge par les algorithmes mis en œuvre par le compilateur.

**Pour écrire des programmes efficaces, il est plus important de bien savoir choisir ses algorithmes plutôt que de bien connaître son assembleur !**

Le temps et l’espace mémoire sont les deux ressources principales en informatique. Il existe toutefois d’autres ressources pour lesquelles on peut chercher à optimiser les programmes. On peut citer la consommation électrique dans le cas de logiciels embarqués. Mais aussi tout simplement le budget nécessaire. Ainsi, pour ce qui est des tris d’éléments rangés sur de la mémoire de masse (des disques durs), il existe un concours appelé *Penny sort* où l’objectif est de trier un maximum d’éléments pour un penny US (un centième de dollar US). L’idée est de considérer une configuration matérielle particulière. On prend en compte le coût d’achat de ce matériel et on considère qu’il peut fonctionner trois années. On obtient alors la durée que l’on peut s’offrir avec un penny. Enfin on mesure sur ce matériel le nombre d’élément que l’on est capable de trier avec le programme testé au cours de cette durée.

### 1.2.3 Complexité en temps et en espace

Dans la suite nous nous intéresserons surtout au coût en temps d'un algorithme et nous travaillerons moins sur le coût en espace. À cela deux raisons.

Les règles d'études du coût en espace s'appuient sur les mêmes notions que celles pour le coût en temps, avec la particularité que si le temps va croissant au cours de l'exécution, il n'en va pas de même de l'utilisation de l'espace. On mesure alors le plus grand espace occupé au cours de l'exécution, en ne comptant pas la place prise par les données en entrée. Nous appellerons *empreinte mémoire* de l'algorithme cet espace.

Le coût en temps borne le coût en espace. En effet, il est réaliste d'estimer que chaque accès à une unité de la mémoire participe du coût en temps pour une certaine durée, minorée par une constante  $d$ . Ainsi en un temps  $t$  un algorithme ne pourra pas occuper plus de  $\frac{t}{d}$  espaces mémoires. Il n'aurait pas le temps d'accéder à plus d'espaces mémoires. Le coût en espace sera donc toujours borné par une fonction linéaire du coût en temps. En général, il sera même bien inférieur.

### 1.2.4 Pire cas, meilleur cas, moyenne

Le coût en temps (ou en espace mémoire) est fonction des données fournies en entrée.

Il y a bien sûr des bons cas et des mauvais cas : souvent un algorithme de tri sera plus efficace sur une liste déjà triée, par exemple. En général on classe les données par leurs tailles : le nombre d'éléments dans la liste à trier, le nombre de bits nécessaires pour coder l'entrée, etc.

On peut alors s'intéresser au *pire cas*. Pour une taille de donnée fixée, quel est le temps maximum au bout duquel cet algorithme va rendre son résultat ? Sur quelle donnée de cette taille l'algorithme atteint-il ce maximum ? Avoir une estimation correcte du temps mis dans le pire des cas est souvent essentiel dans le cadre de l'intégration de l'algorithme dans un programme. En effet, on peut rarement admettre des programmes qui de temps en temps mettent des heures à effectuer une tâche alors qu'elle est habituellement rapide.

On s'intéressera plus rarement aux études en *meilleur cas*, qui est comme le pire cas mais où on prend le minimum au lieu du maximum.

Il est particulièrement utile de faire une étude *en moyenne*. Dans ce cas, on travaille sur l'ensemble des données possibles  $D_n$  d'une même taille  $n$ . Lorsque l'ensemble  $D_n$  est fini, pour chacune de ces données,  $d \in D_n$ , on considère sa probabilité  $p(d)$  ainsi que le temps mis par l'algorithme  $t(d)$ . Le

temps moyen mis par l'algorithme sur les données de taille  $n$  est alors :

$$t_n = \sum_{d \in D_n} p(d) \times t(d).$$

Lorsque l'ensemble de données  $D_n$  est infini on se ramène en général à un ensemble fini, en posant des équivalences entre données.

### 1.2.5 Notation asymptotique

Jusqu'ici nous avons été évasif sur la manière de mesurer le temps d'exécution pour un algorithme en fonction de la taille des entrées.

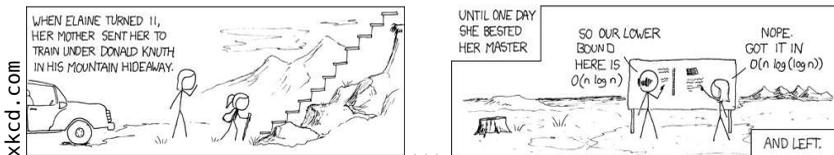
Nous pourrions implanter nos algorithmes sur ordinateur et les chronométrer. Il faut faire de nombreux tests sur des jeux de données importants pour pouvoir publier des résultats utiles. Pour les tailles de donnée non testée on extrapole ensuite les résultats. On obtient ainsi une courbe de l'évolution du coût mesuré en fonction de la taille des donnée (courbe de la moyenne des temps, courbe du temps en pire cas, etc.).

Si ce genre de mesure peut avoir un intérêt ce sera plutôt pour départager plusieurs implantations de quelques algorithmes, sélectionnés auparavant, pour une architecture donnée. Autrement, la mesure risque de rapidement devenir obsolète à cause des changements d'architecture.

Il est beaucoup plus intéressant de faire quelques approximations permettant de mener un raisonnement mathématique grâce auquel on obtient la forme générale de cette courbe exprimée sous la forme d'une fonction mathématique simple,  $f(N)$ , en la taille des données,  $N$ . On dit que la fonction exprime *la complexité* (en temps ou en espace, en pire cas ou en moyenne) de l'algorithme. On peut alors comparer les algorithmes en comparant les fonctions de complexité. S'il faut vraiment optimiser, alors seulement on compare différentes implantations.

Voyons comment on procède pour trouver une fonction de complexité et quelles approximations sont admises.

**Première approximation.** La première approximation qu'on va faire consiste à ne considérer que quelques *opérations significatives* dans l'algorithme et à en négliger d'autres. Bien entendu, il faut que ce soit justifié. Pour une mesure en temps par exemple, il faut que le temps passé à effectuer l'ensemble de toutes les opérations soit directement proportionnel au temps passé à effectuer les opérations significatives. En général, on choisira au moins une opération significative dans chaque étape de boucle.



**Deuxième approximation.** En général, on cherchera un cadre où les opérations significatives sont suffisamment élémentaires pour considérer qu'elles se font toujours à temps constant. Ceci nous amènera à compter le nombre d'opérations significatives élémentaires de chaque type pour estimer le coût en temps de l'algorithme. Il est ainsi courant que les résultats de complexité en temps soient exprimés par le décompte du nombre d'opérations significatives sans donner de conversion vers les unités de temps standards. Chaque opération significative sera ainsi considérée comme participant d'un *coût unitaire*. Autrement dit, notre unité de temps sera le temps d'exécution d'une opération significative et ne sera pas convertie en secondes.

Ainsi pour un tri, par exemple, on pourra se contenter de dénombrer le nombre de comparaisons entre éléments ainsi que le nombre d'échanges. Attention toutefois : ceci n'est valable que si la comparaison ou l'échange se font réellement en un temps borné. C'est le cas de la comparaison ou de l'échange de deux entiers. La comparaison de deux chaînes de caractères pour l'ordre lexicographique demande un temps qui dépend de la taille des chaînes, il est donc incorrect d'attribuer un coût unitaire à cette opération. L'opération significative sera par contre la comparaison de deux caractères qui sert dans la comparaison des chaînes.

Après dénombrement, on exprime le nombre d'opérations significatives effectuées sous la forme d'une fonction mathématique  $f(N)$  de paramètre la taille  $N$  de l'entrée. Selon ce qu'on cherche on peut se contenter d'une majoration ou d'une minoration du nombre d'opérations significatives.

**Troisième approximation.** On se contente souvent de donner une *approximation asymptotique* de  $f$  à l'aide d'une fonction mathématique simple, telle que :

- $\log N$  logarithmique
- $N$  linéaire
- $N \log N$  quasi-linéaire
- $N^{3/2} = N\sqrt{N}$
- $N^2$  quadratique
- $N^3$  cubique
- $2^N$  exponentielle

où le paramètre  $N$  exprime la taille des données.

La notion d'approximation asymptotique et les notations associées sont définies formellement comme suit.

**Définition 1.1.** Soient  $f$  et  $g$  deux fonctions des entiers dans les réels. On dit que  $f$  est asymptotiquement dominée par  $g$ , on note  $f = O(g)$  et on lit  $f$  est en « grand o » de  $g$ , lorsqu’il existe une constante  $c_1$  strictement positive et un entier  $n_1$  à partir duquel  $0 \leq f(n) \leq c_1g(n)$ , i.e.

$$\exists c_1 > 0, \exists n_1 \in \mathbb{N}, \forall n \geq n_1, 0 \leq f(n) \leq c_1g(n).$$

On dit que  $f$  domine asymptotiquement  $g$  et on note  $f = \Omega(g)$  ( $f$  est en « grand omega » de  $g$ ) lorsque

$$\exists c_2 > 0, \exists n_2 \in \mathbb{N}, \forall n \geq n_2, 0 \leq c_2g(n) \leq f(n).$$

On dit que  $f$  et  $g$  sont asymptotiquement équivalentes et on note  $f = \Theta(g)$  ( $f$  est en « grand theta » de  $g$ ) lorsque  $f = O(g)$  et  $f = \Omega(g)$  i.e.

$$\exists c_1 > 0, \exists c_2 > 0, \exists n_3 \in \mathbb{N}, \forall n \geq n_3, 0 \leq c_2g(n) \leq f(n) \leq c_1g(n).$$

Les notations asymptotiques à l’aide du signe égal, adoptées ici, sont trompeuses (par exemple, ce n’est pas parce que  $f = O(g)$  et  $h = O(g)$  que  $f = h$ ) mais assez répandues, il faut éviter de prendre cela pour une véritable égalité.

Pour comprendre pourquoi cette approche est efficace le mieux est de regarder quelques exemples.

Supposons que l’on ait affaire à plusieurs algorithmes et que leurs temps moyens d’exécution soient respectivement exprimés par les fonctions formant les lignes du tableau ci-dessous.

Pour fixer les idées, disons que nos algorithmes sont implantés sur un ordinateur du début des années 70 (premiers microprocesseurs sur quatre bits ou un octet) qui effectue mille opérations significatives par seconde (la fréquence du processeur est meilleure, mais il faut une bonne centaine de cycles d’horloge pour effectuer une opération significative).

Le tableau 1.1 exprime le temps d’exécution en approximation asymptotique de chacun de ces algorithmes en fonction de la taille des données en entrée. L’unité 1 C correspond à un siècle (cent ans). L’unité 1 U correspond à l’estimation courante de l’âge de l’Univers, c’est à dire 13,7 milliards d’années.

Les écarts entre les ordres de grandeurs des temps de traitement, rapportés au temps humain, sont tels qu’un facteur multiplicatif dans l’estimation du temps d’exécution sera généralement négligeable par rapport à la fonction à laquelle on se rapporte. Il faudrait de très grosses ou très petites constantes multiplicatives en facteur pour que celles-ci aient une incidence dans le choix des algorithmes. En négligeant ces facteurs, on ne perd donc que peu d’information sur un algorithme. Les coûts réels, fonction de l’implantation, serviront

$N$	10	50	100	1 000	10 000	100 000	1 000 000
$\log N$	3 ms	6 ms	7 ms	10 ms	13 ms	17 ms	20 ms
$\log^2 N$	11 ms	32 ms	44 ms	0,1 s	0,2 s	0,3 s	0,4 s
$N$	10 ms	50 ms	0,1 s	1 s	10 s	17 min	17 min
$N \log N$	33 ms	0,3 s	0,7 s	10 s	2 min	28 min	6h
$N^{(3/2)}$	32 ms	0,3 s	1 s	30 s	17 min	9h	12 j
$N^2$	0,1 s	2,5 s	10 s	17 min	1,2 j	4 mois	32 a
$N^3$	1 s	2 min	17 min	12 j	32 a	32 C	$10^7$ a
$2^N$	1 s	35 C	$10^9$ U				

Table 1.1 – Principaux équivalents asymptotiques traduits en temps d’exécution sur ordinateur des années 70

ensuite à départager des algorithmes de coûts asymptotiques égaux ou relativement proches.

Une autre constatation à faire immédiatement est que les algorithmes dont le coût asymptotique en temps est au delà du quasi-linéaire ( $N \log N$ ) ne sont pas praticables sur des données de grande taille et que le temps quadratique  $N^2$ , voire le temps cubique  $N^3$  sont éventuellement acceptables sur des tailles moyennes de données. Le coût exponentiel, quant à lui est inacceptable en pratique sauf sur de très petites données.

Ces constatations sont exacerbées par l’accroissement de la rapidité des ordinateurs.

Disons maintenant que nos algorithmes sont implantés sur un ordinateur très récent (2006, plusieurs processeurs 64 bits) qui effectue un milliard d’opérations significatives par seconde (on a gagné en fréquence mais aussi sur le nombre de cycles d’horloge nécessaire pour une opération significative). Nous avons alors le tableau 1.2 (les nombres sans unités sont en milliardième de seconde).

Il est à noter que même avec une très grande rapidité de calcul les algorithmes exponentiels ne sont praticables que sur des données de très petite taille (quelques dizaines d’unités).

### 1.2.6 Optimalité

Grâce à la notation asymptotique nous pouvons classer les algorithmes connus résolvant un problème donné, en vue de les comparer. Mais cela ne nous dit rien de l’existence d’autres algorithmes que nous n’aurions pas imaginé et qui résoudraient le même problème beaucoup plus efficacement. Faut-il chercher de nouveaux algorithmes ou au contraire améliorer l’implantation

$N$	90	$10^3$	$10^6$	$10^8$	$10^9$	$10^{12}$
$\log N$	6	10	20	27	30	40
$\log^2 N$	42	$0,1 \mu s$	$0,4 \mu s$	$0,7 \mu s$	$0,9 \mu s$	$1,5 \mu s$
$N$	90	$1 \mu s$	1 ms	100 ms	1 s	17 min
$N \log N$	584	$9 \mu s$	20 ms	3 s	30 s	11 h
$N^{(3/2)}$	854	$31 \mu s$	1 s	17 min	9 h	32 a
$N^2$	$8 \mu s$	1 ms	17 min	4 mois	32 a	$10^7$ a
$N^3$	$0,7$ ms	1 s	32 a	$10^7$ a	2 U	$10^9$ U
$2^N$	3 U					

Table 1.2 – Principaux équivalents asymptotiques traduits en temps d’exécution sur ordinateur actuel

de ceux qu’on connaît ? Peut-on seulement espérer en trouver de nouveaux qui soient plus efficaces ?

L’algorithmique s’attache aussi à répondre à ce type de questions, où il s’agit de produire des résultats concernant les problèmes directement et non plus seulement les algorithmes connus qui les résolvent. Ainsi, on a des théorèmes du genre : pour ce problème  $P$ , quelque soit l’algorithme employé (connu ou inconnu) le coût en temps/espace, en moyenne/pire cas/meilleur cas, est asymptotiquement borné inférieurement par  $f(N)$ /en  $\Omega(f(N))$ , où  $N$  est la taille de la donnée initiale.

Autrement dit, il arrive que pour un problème donné on sache décrire le coût minimal (en temps ou en espace, en pire cas ou en moyenne) de n’importe quel algorithme le résolvant.

Lorsque un algorithme  $A$  résolvant un problème  $P$  a un coût équivalent asymptotiquement au coût minimal du problème  $P$  on dit que l’algorithme  $A$  est *optimal* (pour le type de coût choisi : temps/espace, moyenne/pire cas).

Voici un exemple de résultat d’optimalité, nous en verrons d’autres.

**Proposition 1.2.** *Soit le problème  $P$  consistant à rechercher l’indice de l’élément maximum dans un tableau  $t$  de  $n$  éléments deux à deux comparables et donnés dans le désordre. Alors :*

1. *tout algorithme résolvant ce problème utilise au moins  $n - 1$  comparaisons ;*
2. *il existe un algorithme optimal pour ce problème, c’est à dire un algorithme qui résout  $P$  en exactement  $n - 1$  comparaisons.*

Pour démontrer la première partie de la proposition, on utilise le lemme suivant :

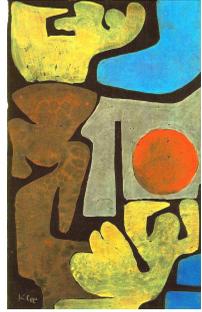
**Lemme 1.3.** Soit  $A$  un algorithme résolvant le problème  $P$  de recherche du maximum, soit  $t$  un tableau en entrée dont tous les éléments sont différents, et soit  $i_{\max}$  l'indice de l'élément maximum. Alors pour tout indice  $i$  du tableau différent de  $i_{\max}$ , l'élément  $t[i]$  est comparé au moins une fois avec un élément plus grand au cours de l'exécution de l'algorithme.

On déduit du lemme que si  $n$  est la taille du tableau, alors  $A$  effectue au moins  $n - 1$  comparaisons. En effet pour chaque indice  $i$  différent de  $i_{\max}$  le lemme établit l'existence d'une comparaison entre  $t[i]$  et un autre élément du tableau, que nous noterons  $k_i$ . Comme cela représente  $n - 1$  indices, il suffit de montrer que toutes ces comparaisons sont différentes les unes des autres pour en dénombrer au moins  $n - 1$ . Pour  $i \neq j$  pour que les deux comparaisons associées soient en fait la même il faudrait que  $i = k_j$  et  $j = k_i$ . Comme le lemme dit également que  $t[i] < t[k_i]$  et que  $t[j] < t[k_j]$ , on aurait une contradiction entre  $t[i] < t[k_i]$  et  $t[i] = t[k_j] < t[j] = t[k_i]$ . C'est donc que les deux comparaisons sont deux à deux différentes.

Il reste à prouver le lemme. Par l'absurde. Soit  $A$  un algorithme tel qu'au moins un élément, disons  $t[j]$ , différent de  $t[i_{\max}]$  n'est comparé avec aucun des éléments qui lui sont supérieurs. Alors changer la valeur de l'élément  $t[j]$  pour une valeur supérieure dans le tableau en entrée n'affecte pas le résultat de  $A$  sur cette entrée. Ainsi, il suffit de prendre  $t[j]$  plus grand que  $t[i_{\max}]$  pour que l'algorithme soit faux (il devrait rendre  $j$  et non  $i_{\max}$ ).

Voilà pour la première partie de la proposition. La seconde partie est immédiate, par écriture de l'algorithme : voir exercice 20 page 86 (il suffit d'inverser l'ordre pour avoir un algorithme qui trouve le maximum au lieu du minimum).

## Chapitre 2



# Les algorithmes élémentaires de recherche et de tri



---

Le Parc des idoles de Paul Klee, façon Art en bazar, Ursus Wehrli, éditions Milan jeunesse.

---

Dans ce chapitre nous nous intéressons à la recherche et au tri d'éléments de tableaux unidimensionnels. Les tableaux sont indexés à partir de 0.

Les éléments possèdent chacun une *clé*, pouvant servir de clé de recherche ou de clé de tri (dans un carnet d'adresse, la clé sera par exemple le nom ou le prénom associé à une entrée).

Les comparaisons entre éléments se font par comparaison des clés. On suppose que deux clés sont toujours comparables : soit la première est plus grande que la seconde, soit la seconde est plus grande que la première, soit elles sont égales (ce qui ne veut pas dire que les éléments ayant ces clés sont égaux).

Si on veut trier des objets par leurs masses, on considérera par exemple que la clé associée à un objet est son poids terrestre et on comparera les objets à l'aide d'une balance à deux plateaux.

Dans la suite on considérera une fonction de comparaison :

$$\text{Comparer}(a, b) \text{ qui rend : } \begin{cases} -1 \text{ si } a > b \\ 1 \text{ si } a < b \\ 0 \text{ lorsque } a = b \text{ (même masse).} \end{cases}$$

Un élément ne se réduit pas à sa clé, on considérera qu'il peut contenir des *données satellites* (un numéro de téléphone, une adresse, etc.).

## 2.1 La recherche en table

On considère le problème qui consiste à rechercher un élément par sa clé dans un groupe d'éléments organisés en tableau.

### 2.1.1 Recherche par parcours

Lorsque l'ordre des éléments dans le tableau est quelconque, il faut forcément comparer la clé sur laquelle s'effectue la recherche avec la clé de chacun des éléments du tableau. Si le tableau a une taille  $N$  et si un seul élément possède la clé recherchée, alors il faut effectuer : 1 comparaison en meilleur cas,  $N$  comparaisons en pire cas et  $N/2$  comparaisons en moyenne (sous l'hypothèse que l'élément recherché est bien dans le tableau et que toutes les places sont équiprobables). Ainsi rechercher un élément dans un tableau est un problème linéaire en la taille du tableau.

### 2.1.2 Recherche dichotomique

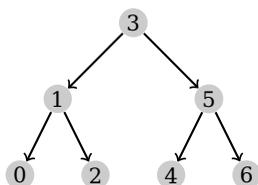
Lorsque le tableau, de taille  $N$  est déjà trié, disons par ordre croissant, on peut appliquer une recherche dichotomique. Dans ce cas, nous allons voir que la recherche est au pire cas et en moyenne en  $\Theta(\log N)$  (le meilleur cas restant en  $O(1)$ ). À cette occasion nous introduisons la notion d'*arbre de décision*, dont on se servira encore pour les tris.

Rappelons ce qu'est la recherche dichotomique. Étant donné le tableau (trié) et une clé pour la recherche on cherche l'indice d'un élément ayant cette clé dans le tableau. On compare la clé recherchée avec la clé de l'élément au milieu du tableau, disons d'indice  $m$ . Cet indice est calculé par division par deux de la taille du tableau puis arrondi par partie entière inférieure,  $m = \lfloor N/2 \rfloor$ . Si la clé recherchée est plus petite on recommence avec le sous-tableau des éléments entre 0 et  $m - 1$ , si la clé est plus grande on recommence

avec le sous-tableau des éléments de  $m + 1$  à  $N - 1$ . On s'arrête soit sur un élément ayant la clé recherchée et on rend son indice soit parce que le sous-tableau que l'on est en train de considérer est vide et on rend une valeur spéciale d'indice, disons  $-1$ , pour dire que l'élément n'est pas dans le tableau.

Dans cet algorithme, on considère la comparaison des clés comme seule opération significative.

On représente tous les branchements conditionnels possibles au cours de l'exécution sous la forme d'un arbre binaire. L'arbre obtenu s'appelle un *arbre de décision*. Ici les tests qui donnent lieu à branchement sont les comparaisons entre la clé de l'élément recherché et la clé d'un élément du tableau. On peut donc représenter le test en ne notant que l'indice de l'élément du tableau dont la clé est comparée avec la clé recherchée. Considérons le cas  $N = 2^3 - 1 = 7$ . L'arbre de décision est alors :



Cet arbre signifie qu'on commence par comparer la clé recherchée avec l'élément d'indice 3 (car  $\lfloor 7/2 \rfloor = 3$ ). Il y a ensuite trois cas : soit on s'arrête sur cet élément soit on continue à gauche (avec  $\lfloor 3/2 \rfloor = 1$ ), soit on continue à droite (avec l'élément d'indice  $\lfloor 3/2 \rfloor = 1$  du sous-tableau de droite, c'est à dire l'élément d'indice  $4 + 1$  dans le tableau de départ). Et ainsi de suite.

Si l'élément recherché n'est pas dans le tableau on parcourt toute une branche de l'arbre de la racine à une feuille.

Supposons que l'on applique cet algorithme à un tableau de taille  $N = 2^k - 1$ . La hauteur de l'arbre est  $k$ . Si l'élément n'est pas dans le tableau on fait donc systématiquement  $k$  comparaisons. De même, par exemple, lorsque l'élément est dans la dernière case du tableau. C'est le pire cas. Comme  $N = 2^k - 1$ ,  $k - 1 \leq \log N < k$  et on en déduit facilement que le pire cas est en  $\Theta(\log N)$  (pour  $N > 2$ ,  $\frac{1}{2} \log N \leq k \leq \log N$ ). Ceci lorsque  $N$  est de la forme  $2^k - 1$ .

On peut faire moins de comparaisons que dans le pire cas. Combien en fait on en moyenne lorsque la clé recherchée est dans le tableau, en un seul exemplaire, et que toutes les places sont équiprobables ?

Exactement :

$$\text{moy}(N) = \frac{\sum_{i=1}^k i \times 2^{i-1}}{N}$$

Puisque dans un arbre binaire parfait de hauteur  $k$  il y a  $2^{i-1}$  éléments de hauteur  $i$  pour chaque  $i \leq k$ .

On cherche une forme close pour exprimer  $\text{moy}(N)$ .

On remarque que  $\sum_{i=1}^k i \times 2^{i-1}$  est la série  $\sum_{i=1}^k i \times z^{i-1}$  où  $z = 2$ . On peut commencer la sommation à l'indice 0, puisque dans ce cas le premier terme est nul. En posant  $S(z) = \sum_{i=0}^k z^i$  il vient  $S'(z) = \sum_{i=0}^k i \times z^{i-1}$ . Mais  $S(z)$  est une série géométrique de raison  $z$  donc :

$$S(z) = \frac{z^{k+1} - 1}{z - 1} \quad \text{et, par conséquent}$$

$$S'(z) = \frac{(k+1)z^k(z-1) - (z^{k+1} - 1)}{(z-1)^2}$$

En fixant  $z = 2$  on obtient :

$$\text{moy}(N) = k - 1 + \frac{k}{N}$$

Ceci est une formule exacte. Comme  $k = \lfloor \log N \rfloor + 1$ , on en déduit sans trop de difficultés que  $\text{moy}(N) = \Theta(\log N)$  (prendre, par exemple, l'encadrement  $\frac{1}{2} \log N \leq \text{moy}(N) \leq 2 \log N$ ).

L'étude du nombre moyen de comparaisons effectuées par une recherche dichotomique, comme celle du pire cas, a été menée pour une taille particulière de tableau :  $N = 2^k - 1$ . Il est tout à fait possible de généraliser le résultat  $\text{moy}(N) = \Theta(\log N)$  à  $N$  quelconque en remarquant que pour  $N$  tel que  $2^{k-1} - 1 < N < 2^k - 1$  la moyenne du nombre de comparaisons est entre  $\Omega(\log(N-1))$  et  $O(\log N)$ , puis en donnant une minoration de  $\log(N-1)$  par un  $c \times \log N$ . De même pour le pire cas. Nous ne rentrons pas dans les détails de cette généralisation.

En général, on pourra supposer que les fonctions de complexité sont croissantes et ainsi déduire des résultats généraux à partir de ceux obtenus pour une suite infinie strictement croissante de tailles de données (ici la suite est  $u_k = 2^k - 1$ ).

## 2.2 Le problème du tri

Nous nous intéressons maintenant au problème du tri. Nous ne considérons pour l'instant que les tris d'éléments d'un tableau, nous verrons les tris de listes au chapitre sur les structures de données.

On cherche des algorithmes généralistes : on veut pouvoir trier des éléments de n'importe quelles sortes, pourvu qu'ils soient comparables. On dit

que ces tris sont par comparaison : les seuls tests effectués sur les éléments donnés en entrée sont des comparaisons.

Pour qu'un algorithme de tri soit correct, il faut qu'il satisfasse deux choses : qu'il rende un tableau trié, et que les éléments de ce tableau trié soient exactement les éléments du tableau de départ.

**En place.** Un algorithme est dit en place lorsque la quantité de mémoire qu'il utilise en plus de celle fournie par la donnée est constante en la taille de la donnée. Typiquement, un algorithme de tri en place utilisera de la mémoire pour quelques variables auxiliaires et procédera au tri en effectuant des échanges entre éléments directement sur le tableau fourni en entrée. Dans la suite, on utilisera une fonction `Échanger-Tableau( $T, i, j$ )` (`echangertab()` en C) qui prend en paramètre un tableau  $T$  et deux indices  $i$  et  $j$  et échange les éléments  $T[i]$  et  $T[j]$  du tableau. Le fait de n'agir sur le tableau  $T$  que par des appels à la fonction `Échanger-Tableau()` est une garantie suffisante pour que le tri soit en place mais bien entendu ce n'est pas strictement nécessaire. Lorsque un tri n'est pas en place, sa mémoire auxiliaire croît avec la taille du tableau passé en entrée : c'est typiquement le cas lorsqu'on crée des tableaux intermédiaires pour effectuer le tri ou encore lorsque le résultat est rendu dans un nouveau tableau.

**Stable.** Il arrive fréquemment que des éléments différents aient la même clé. Dans ce cas on dit que le tri est stable lorsque toute paire d'éléments ayant la même clé se retrouve dans le même ordre à la fin qu'au début du tri : si  $a$  et  $b$  ont mêmes clés et si  $a$  apparaît avant  $b$  dans le tableau de départ, alors  $a$  apparaît encore avant  $b$  dans le tableau d'arrivée. On peut toujours rendre un tri par comparaison stable, il suffit de modifier la fonction de comparaison pour que lorsque les clés des deux éléments comparés sont égales, elle compare l'indice des éléments dans le tableau.

On considère que la comparaison est l'opération la plus significative sur les tris généralistes. Sauf avis contraire, on considérera la comparaison comme une opération élémentaire (qui s'exécute en temps constant). L'échange peut parfois aussi être considéré comme une opération significative. Pour les tris qui ne sont pas en place, il faut aussi compter les allocations mémoires : l'allocation de  $n$  espaces mémoires de taille fixée comptera pour un temps  $n$  et un espace  $n$ .

## 2.3 Les principaux algorithmes de tri

### 2.3.1 Tri sélection

Souvent les tris en place organisent le tableau en deux parties : une partie dont les éléments sont triés et une autre contenant le reste des éléments. La partie contenant les éléments triés croît au cours du tri.

Le principe du tri par sélection est de construire la partie triée en rangeant à leur place définitive les éléments. La partie triée contient les  $n$  plus petits éléments du tableau dans l'ordre. Au départ,  $n = 0$ . La partie non triée contient les autres éléments, tous plus grands que ces  $n$  premiers éléments, dans le désordre. Pour augmenter la partie triée, on choisit le plus petit des éléments de la partie non triée et on le place en bout de partie triée (en  $n$  si le tableau est indexé à partir de 0, comme en C). La recherche du plus petit élément de la partie non triée se fait par parcours complet de la partie non triée. Si il y a plusieurs plus petit élément on choisit celui de plus petits indices.

Voici, figure 2.1, une version en C du tri sélection (voir aussi l'exercice 20).

```
void triselection(tableau_t *t){
    int n, j, k;
    for (n = 0; n < taille(t) - 1; n++) {
        /* Les éléments t[0 .. n - 1] sont à la bonne place */
        k = n;
        /* On cherche le plus petit élément dans t[n .. N - 1] */
        for (j = k + 1; j < taille(t); j++){
            if ( 0 < comparer(t[j], t[k]) ) /* t[j] < t[k] */
                k = j;
        }
        /* Sa place est à l'indice n */
        echangertab(t, k, n);
    }
}
```

Figure 2.1 – Tri sélection

Il est facile de montrer que ce tri est toujours en  $\Theta(N^2)$  quelle que soit la forme de l'entrée.

### Arbres de décision des tris par comparaison

Comme pour la recherche dichotomique on peut associer un arbre de décision à un algorithme de tri pour chaque taille de donnée. Une fois que la taille de donnée est fixée, les branchements conditionnels sont obtenus par des comparaisons des éléments du tableau de départ. Pour simplifier, on ne

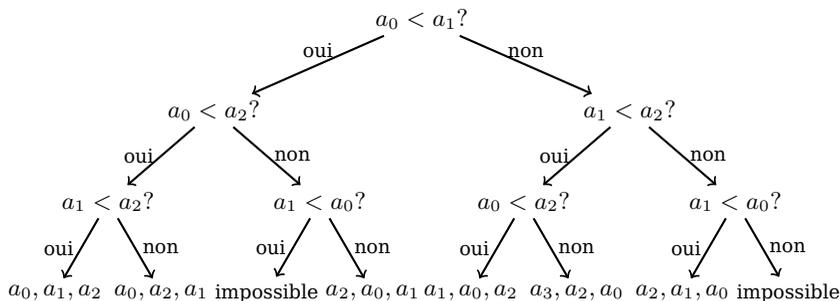


Figure 2.2 – Arbre de décision du tri par sélection

tiendra pas compte des cas d'égalité entre clés : on suppose que toutes les clés sont différentes.

Voici, figure 2.2, l'arbre de décision du tri par sélection dans le cas où le tableau contient trois éléments. Le tableau de départ contient les éléments  $a_0$ ,  $a_1$  et  $a_2$  (d'indices 0, 1, 2). Ce tableau est modifié au cours de l'exécution : dans l'arbre de décision, on regarde quel élément est comparé avec quel autre élément, non pas quel indice est comparé avec quel autre indice : ainsi si  $a_0$  se retrouve à l'indice 1, et  $a_2$  à l'indice 2, on notera  $a_0 < a_2$  le nœud de l'arbre de décision correspondant à la comparaison entre ces deux éléments, et non  $T[1] < T[2]$ . À chaque fois la branche de gauche correspond à la réponse oui et la branche de droite à la réponse non.

Comme on considère tous les cas possibles, toutes les permutations possibles de l'entrée apparaissent comme une feuille de l'arbre de décision. Et ce toujours en un seul endroit – à chaque permutation correspond une et une seule feuille – car une permutation détermine complètement l'ordre des éléments et donc le résultat des comparaisons.

Pour le tri sélection, certaines comparaisons faites sont inutiles : leurs résultats auraient pu être déduits des résultats des comparaisons précédentes. Dans ces comparaisons, un des deux branchements n'est donc jamais emprunté, il est noté ici comme impossible.

Il arrive fréquemment que des algorithmes fassent des tests inutiles (quelle que soit l'entrée). Ce sera encore le cas pour le tri bulle, par exemple.

### 2.3.2 Tri bulle

En anglais : *bubble sort*

L'idée du tri bulle est très naturelle. Pour tester si un tableau est trié on compare deux à deux les éléments consécutifs  $T[i]$ ,  $T[i + 1]$  : on doit toujours

```

void tribulle(tableau_t *t){
    int n, k, fin;
    for (n = taille(t) - 1; n >= 1; n--) {
        /* Les éléments d'indice > n sont à la bonne place. */
        fin = 1;
        for (k = 0; k < n; k++){
            if ( 0 > comparer(t[k], t[k + 1]) ){ /* t[k] > t[k + 1] */
                echangertab(t, k, k + 1);
                fin = 0;
            }
        }
        if (fin) break; /* Les éléments entre 0 et n - 1 sont bien ordonnés */
    }
}

```

Figure 2.3 – Tri bulle

avoir  $T[i] \leq T[i + 1]$ . Dans le tri bulle, on parcourt le tableau de  $i = 0$  à  $i = N - 2$ , en effectuant ces comparaisons. Et à chaque fois que le résultat de la comparaison est  $T[i] > T[i + 1]$  on échange les deux éléments. Si un parcours se fait sans échange c'est que le tableau est trié. Autrement, il suffit de recommencer sur le sous-tableau des  $N - 1$  premiers éléments. En effet, un tel parcours amène toujours par échanges successifs, l'élément maximum en fin de tableau. Ainsi on construit une fin de tableau, dont les éléments sont rangés à leurs places définitives et dont la taille croît de un à chaque nouvelle passe. Tandis que la taille du tableau des éléments qu'il reste à trier diminue de un, à chaque passe.

Voici, figure 2.3, une fonction C effectuant le tri bulle.

Mais le tri bulle est assez lent en pratique. Il s'agit d'un algorithme en  $\Theta(N^2)$  (pire cas et moyenne) qui est plus lent que d'autres algorithmes de même complexité asymptotique (tri insertion, tri sélection).

Pour illustrer un des principaux défauts du tri bulle, on parle parfois de tortues et de lièvres. Les tortues sont des éléments qui ont une petite valeur de clé, relativement aux autres éléments du tableau, et qui se trouvent à la fin du tableau. Le tri bulle est lent à placer les tortues : elles sont déplacées d'au plus une case à chaque passe. Symétriquement les lièvres sont des éléments au début du tableau dont les clés sont grandes relativement aux autres éléments. Ces éléments sont vite déplacés vers la fin du tableau par les premières passes du tri bulle.

Le tri bulle admet plusieurs variantes. Dans chacune de ces variantes, les tortues trouvent leur place plus vite.

## Tri bulle bidirectionnel

Le tri bulle bidirectionnel revient simplement à alterner les parcours du début vers la fin du tableau avec des parcours de la fin vers le début. Ceci a pour effet de rétablir une symétrie de traitement entre les lièvres et les tortues.

## Tri gnome

Le tri gnome s'apparente au tri bulle au sens où on compare et on échange uniquement des éléments consécutifs du tableau. Dans le tri gnome, on compare deux éléments consécutifs : s'ils sont dans l'ordre on se déplace d'un cran vers la fin du tableau (ou on s'arrête si la fin est atteinte) ; sinon, on les intervertit et on se déplace d'un cran vers le début du tableau (si on est au début du tableau alors on se déplace vers la fin). On commence par le début du tableau.

### 2.3.3 Tri insertion

Le tri insertion est le tri du joueur de cartes. On maintient une partie du jeu de carte triée, en y insérant les cartes une par une. Pour chaque insertion, on doit chercher la place de la carte qu'on ajoute. Le tri commence en mettant une carte dans la partie triée et il se termine lorsque toutes les autres cartes ont été insérées.

Dans le cas de tableaux, l'ajout nécessite de décaler les éléments de la partie triée plus grands que l'élément inséré. Il est alors facile de voir que ce tri est en  $\Theta(N^2)$  en pire cas. Voici, figure 2.4, une version C.

## Tri Shell

Le tri Shell, due à D. L. Shell (1959), et que nous ne verrons pas, peut être considéré comme une amélioration du tri insertion.

### 2.3.4 Tri fusion

En anglais : *merge sort*

Le tri fusion (von Neumann, 1945) est un très bon tri, sur le principe du diviser pour régner. Mais il a le défaut de ne pas être en place et de nécessiter une mémoire auxiliaire de la taille de la donnée. Ainsi l'empreinte mémoire du tri fusion est de l'ordre de  $N$ , où  $N$  est la taille du tableau fourni en entrée (attention : on ne compte pas la taille du tableau en entrée – uniquement

```

void triinsertion(tableau_t *t){
    int n, k;
    element_t e;
    for (n = 1; n < taille(t); n++) {
        /* --- Invariant: le sous-tableau entre 0 et n - 1 est trié --- */
        /* Insertion du n + 1 ième élément dans ce sous-tableau trié : */
        /* --1) Le n + 1 ième élément est sauvegardé dans e */
        e = t[n];
        /* --2) Décalage des éléments plus grands que e du sous-tableau */
        k = n - 1;
        while ( ( k >= 0) && (0 < comparer(e, t[k]) ) ){ /* e < t[k] */
            t[k + 1] = t[k];
            k--;
        }
        /* --3) La nouvelle place de l'élément e est à l'indice k + 1 */
        t[k + 1] = e;
    }
}

```

Figure 2.4 – Tri insertion

accessible en lecture – ni la taille du tableau en sortie – uniquement accessible en écriture).

Il s'agit de partager le tableau à trier en deux sous-tableaux de tailles (quasiment) égales. Une fois que les deux sous-tableaux seront triés il suffira de les interclasser pour obtenir le tableau trié. Le tri des deux sous-tableaux se fait de manière récursive.

Ainsi pour trier un tableau de taille quatre on commence par trier deux tableaux de taille deux. Et pour trier le premier d'entre eux on doit trier deux tableaux de taille un... qui sont déjà triés (un tableau de taille un est toujours trié). On interclasse ces deux derniers tableaux, puis on doit trier le second tableau de taille deux. Lorsque c'est fait on interclasse les deux tableaux de taille deux.

Pour l'interclassement de deux tableaux de taille identique  $n$ , on effectue en pire cas  $2n - 1$  comparaisons. Voir exercice 21.

On cherche un majorant du nombre maximum de comparaisons effectuées dans le tri fusion (c'est à dire un résultat en pire cas).

Considérons un tableau de taille  $2^k$  en entrée et notons  $u_k$  le nombre maximum de comparaisons effectuées par le tri fusion sur cette entrée. Le nombre maximum de comparaisons effectuées est majoré par deux fois le nombre de comparaisons nécessaires au tri d'un tableau de taille  $2^{k-1}$ , plus le nombre maximum de comparaisons nécessaire à l'interclassement de deux tableaux

de taille  $2^{k-1}$ , qui est  $2^k - 1$ . On a donc la relation :

$$u_k = 2u_{k-1} + 2^k - 1.$$

Pour  $k = 0$ , on effectue aucune comparaison, on devrait donc écrire  $u_0 = 0$ . Mais ce n'est pas très réaliste de compter un temps 0 pour une opération qui prend tout de même un peu de temps (le problème ici est qu'il n'est pas correct de ne compter que le nombre de comparaisons. On pose donc arbitrairement  $u_0 = 1$ , à interpréter comme : l'appel au tri fusion sur un tableau de un élément prend un temps de l'ordre d'une comparaison. De plus cela va bien nous arranger pour résoudre la récurrence.

On pose  $v_k = u_k - 1$ . On obtient :

$$v_k = 2v_{k-1} + 2^k.$$

On montre que  $v_k = k2^k$ . Parce qu'on a posé  $u_0 = 1$ , on a  $v_0 = u_0 - 1 = 0$  qui est bien égal à  $0 \times 2^0$ . Par ailleurs on a :

$$\begin{aligned} v_{k+1} &= 2(k2^k) + 2^{k+1} \\ &= (k+1)2^{k+1}. \end{aligned}$$

Ce qui prouve par récurrence que  $v_k = k2^k$ .

On en déduit  $u_k = k2^k + 1$ . Ainsi si  $N = 2^k$  on fait au maximum  $N \log N + 1$  comparaisons, ce qui est en  $O(N \log N)$ . Puisque cette majoration est correcte pour le pire cas, elle est encore une majoration pour le nombre moyen de comparaison.

Nous démontrons plus loin que le nombre moyen de comparaisons de n'importe quel tri généraliste est toujours (au moins en)  $\Omega(N \log N)$ .

En anticipant on conclut donc que le tri fusion est en  $\Theta(N \log N)$  en moyenne et en pire cas.

### 2.3.5 Tri rapide

En anglais : *quick sort*, C. A. R. Hoare, 1960

Le tri rapide fonctionne aussi comme un diviser pour régner. Il s'agit de choisir un élément du tableau appelé le pivot et de chercher sa place  $p$  en rangeant entre 0 et  $p - 1$  les éléments qui lui sont plus petits et entre  $p + 1$  et  $N - 1$  les éléments qui lui sont plus grands. Ainsi on fait une partition du tableau autour du pivot. Ensuite il suffit de trier par appel récursif ces deux sous-tableaux (entre 0 et  $p - 1$  et entre  $p + 1$  et  $N - 1$ ) pour achever le tri. On fait appel à une fonction Sous-tableau( $T, i, n$ ) (soustab() en C) prenant un

tableau  $T$ , un indice  $i$  et une taille  $n$ , qui rend le sous-tableau de  $T$  commençant à l'indice  $i$  et de longueur  $n$ , sans en faire de copie : une modification du sous-tableau entraîne une modification du tableau  $T$ .

Partitionner un tableau de taille  $n$  coûte un temps  $n$  (on fait comme dans l'exercice 6 sauf qu'au lieu de la couleur on utilise le résultat de la comparaison contre le pivot).

Il peut arriver que la partition soit complètement déséquilibrée. Supposons qu'on choisisse toujours le premier élément du tableau comme pivot. Alors le tableau des éléments déjà triés laisse le pivot à sa place à chaque appel, et dans ce cas, le tableau des valeurs inférieures au pivot est toujours vide. On fera donc  $N$  appels récursifs au tri, le premier sur tout le tableau, demandera  $N - 1$  comparaisons pour faire le partitionnement, le deuxième demandera  $N - 2$ , etc. Le nombre total de comparaisons est alors en  $\Theta(N^2)$ . La profondeur de pile d'appel en  $N$  implique une utilisation de la mémoire en  $\Theta(N)$ . C'est le pire cas.

Mais on peut montrer (on ne le fera pas ici) que le tri rapide est en moyenne en  $\Theta(N \log N)$  pour le temps et en  $\Theta(\log N)$  pour l'espace, lorsque toutes les permutations possibles sont équiprobables en entrée. Comme il utilise peu de mémoire, contrairement au tri fusion, cela fait de ce tri un très bon tri, qui donne d'ailleurs de bons résultats pratiques. Il est ainsi souvent employé.

Lorsqu'on n'est pas certain que les entrées sont équiprobables on peut *randomiser* l'entrée de manière à donner la même probabilité à chaque permutation. Pour cela, il suffit de changer l'ordre de la donnée en tirant au hasard la place de chaque élément. En fait, plutôt que de changer l'ordre de tous les éléments à l'avance, on peut se contenter de tirer le pivot au hasard à chaque appel.

Voici, figure 2.5, une version en C du tri rapide randomisé.

### 2.3.6 Tableau récapitulatif (tris par comparaison)

Nous venons de voir deux tris, le tri fusion et le tri rapide, qui fonctionnent sur le principe du diviser pour régner. Pour l'un, le tri fusion, la division est facile mais il y a du travail pour *régner* (la fusion par entrelacement) et pour l'autre c'est l'inverse, la division est plus difficile (le partitionnement) mais *régner* est simple (il n'y a rien besoin de faire).

On récapitule les résultats de complexité sur les principaux algorithmes de tri généralistes dans le tableau 2.1 (nous n'avons pas tout démontré).

Dans ce tableau, nous faisons aussi figurer le tri par tas, que nous ne verrons qu'au prochain chapitre.

```

void trirapide (tableau_t *t){
  if (taille(t) > 1) {
    int k;
    tableau_t *t1, *t2;
    int p = 0;
    /* Randomisation: on choisit le pivot au hasard */
    echangertab(t,0, random()%taille(t));
    /* Partition ----- */
    /* Invariant : pivot en 0, éléments plus petits entre 1 et p,
       plus grands entre p + 1 et k - 1, indéterminés au delà. */
    for (k = 1; k < taille(t); k++){
      if ( t > comparer(t[0], t[k]) ){ /* t[0] > t[k] */
        p++;
        echangertab(t, p, k);
      }
    }
    /* Range le pivot à sa place, p. ----- */
    echangertab(t, 0, p);
    /* Tri du sous-tableau [0..p - 1] ----- */
    t1 = soustab(t, 0, p);
    trirapide(t1);
    /* tri du sous-tableau [p + 1..N - 1] ----- */
    t2 = soustab(t, p + 1, taille(t) - p - 1);
    trirapide(t2);
  }
}

```

Figure 2.5 – Tri rapide (*quicksort*)

Algorithme	moyenne	pire cas	espace	remarque
bulle	$N^2$	$N^2$	en place	stable
sélection	$N^2$	$N^2$	en place	
insertion	$N^2$	$N^2$	en place	stable
rapide	$N \log N$	$N^2$	$N$ (pire cas)	pas stable
fusion	$N \log N$	$N \log N$	$N$	stable
par tas	$N \log N$	$N \log N$	en place	stable, non local

Table 2.1 – Principaux résultats de complexité sur les tris généralistes

## 2.4 Une borne minimale : $N \log N$

Nous démontrons maintenant que tout algorithme de tri généraliste, fondé sur la comparaison, est au minimum en  $N \log N$  en moyenne (et en pire cas).

Nous utilisons pour cela les arbres de décisions. Pour une taille de tableau fixée,  $N$ , les nœuds de ces arbres sont uniquement des comparaisons, deux à

deux des éléments du tableau en entrée.

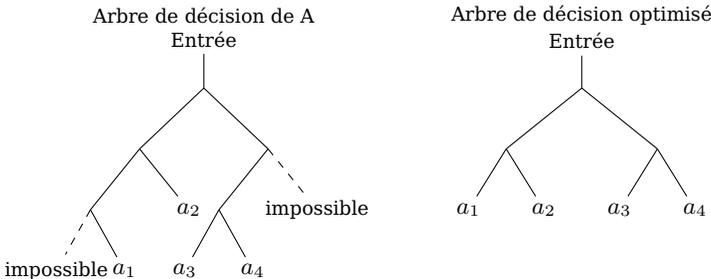
Pour simplifier nous nous restreignons aux tableaux dont les éléments sont tous deux à deux différents. De plus, nous identifions les tableaux qui correspondent à une même permutation de la liste triée : ainsi, sur des entiers, les entrées 11, 14, 13, 12 ou  $-10, 1, 20, 0$  ou 100, 20000, 10000, 1000 sont considérées comme équivalentes et nous les identifions à la permutation  $\sigma = 0, 3, 2, 1$ .

Enfin, on considère que toutes les permutations sont équiprobables.

**Compter les comparaisons grâce à l'arbre de décision.** Soit un algorithme de tri  $A$ . Considérons l'arbre de décision de  $A$  sur les entrées de taille  $N$ . Chaque nœud interne (un nœud qui n'est pas une feuille) est une comparaison. Comme  $A$  est un tri, chaque permutation de  $\{0, \dots, N - 1\}$  de la liste triée doit apparaître comme feuille de cet arbre. De plus, une permutation apparaît au plus dans une feuille, puisque la donnée de la permutation détermine précisément le résultat de chaque comparaison. Ainsi le nombre de feuilles est minoré par le nombre de permutations.

Pour une permutation, le nombre de comparaisons effectuées par  $A$  est précisément le nombre de nœuds internes traversés lorsqu'on va de la racine de l'arbre à la feuille qui correspond à cette permutation. C'est à dire la profondeur de la feuille.

Il est possible que certaines comparaisons dans l'arbre de décision soient inutiles et qu'elles fassent apparaître des branchements impossibles. On supprime ces comparaisons. Ainsi la profondeur d'une permutation est désormais un minorant du nombre réel de comparaisons effectuées (en un sens, on optimise  $A$ ). Dans l'arbre obtenu, chaque feuille est une permutation. De plus tous les branchements ont deux descendants : on dit que l'arbre binaire est *complet*. Il y a  $N!$  permutations donc  $N!$  feuilles. Comme on a supprimé des comparaisons, la plus grande profondeur de permutation minore le nombre de comparaisons en pire cas. Et la moyenne des profondeurs minore le nombre moyen de comparaisons.



**Lemme 2.1.** *Dans un arbre binaire, si la profondeur maximale des feuilles est  $k$  alors le nombre de feuilles est au plus  $2^k$ .*

Démonstration facile par récurrence. Laissez en exercice (exercice 41). Ainsi la profondeur maximale de permutation dans l'arbre est au moins  $\log(N!)$ . On peut démontrer que  $\log(N!)$  est en  $\Omega(N \log N)$  (voir exercice 19).

On en déduit immédiatement que le nombre de comparaisons en pire cas de n'importe quel tri est en  $\Omega(N \log N)$ .

Mais nous voulons améliorer ce résultat en trouvant un minorant pour le nombre moyen de comparaisons. Ce nombre est minoré par la moyenne de la profondeur des feuilles de l'arbre optimisé.

Lorsque toutes les profondeurs sont (à peu près) égales il est plus facile de trouver un minorant asymptotique serré.

**Définition 2.2.** Un arbre est *équilibré* lorsque la différence des profondeurs entre deux feuilles est au plus 1 quel que soit le choix de ces deux feuilles.

**Lemme 2.3.** *Un arbre binaire complet équilibré qui a  $K$  feuilles est tel que chaque feuille est de profondeur supérieure ou égale à  $\log(K) - 1$ .*

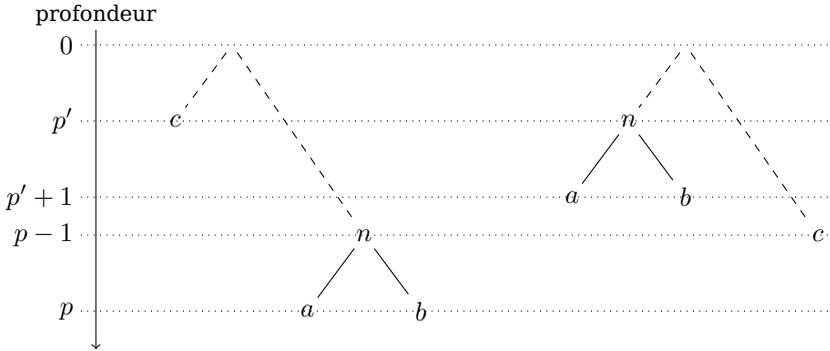
Par l'absurde, supposons qu'il y ait une feuille de profondeur strictement inférieure à  $\log(K) - 1$ . Comme l'arbre est équilibré cela signifie qu'il est de profondeur maximum strictement inférieure à  $\log K$ . Ainsi son nombre de feuilles est strictement inférieur à  $2^{\log K} - 1 = K - 1$ . Contradiction.

La moyenne des profondeurs dans un arbre binaire complet équilibré à  $K$  feuilles est donc minorée par  $\log(K) - 1$ . Ici  $K = N!$ . On a  $\log(N!) = \Omega(N \log N)$  et il est facile d'en déduire que  $\log(N!) - 1 = \Omega(N \log N)$ . Ainsi, dans le cas où l'arbre est équilibré, la moyenne des profondeurs est en  $\Omega(N \log N)$ .

Pour établir un résultat plus général, nous allons maintenant montrer qu'à nombre de feuilles fixé, la moyenne des profondeurs des feuilles dans un arbre binaire complet est minimale lorsque l'arbre est équilibré.

Pour cela on définit une opération qui transforme un arbre binaire complet non équilibré en un nouvel arbre binaire complet ayant les mêmes feuilles mais dont la moyenne des profondeurs des feuilles est strictement plus petite.

**Transformation des arbres binaires complets non équilibrés.** Soit un arbre binaire complet non équilibré. Soit une feuille  $a$  de profondeur maximale dans l'arbre et soit  $p$  cette profondeur. Soit  $n$  le nœud interne juste au dessus de cette feuille. Comme  $a$  est de profondeur maximale, les deux branchements issus de  $n$  sont des feuilles. Il y a donc  $a$  et une autre feuille  $b$  toutes les deux de profondeur  $p$ . Comme l'arbre n'est pas équilibré, il existe une feuille  $c$  de profondeur  $p' \leq p - 2$ . On échange  $c$  avec  $n$  et ses deux branches  $a$  et  $b$ .



La profondeur de  $c$  passe de  $p'$  à  $p - 1$  et la profondeur de  $a$  et de  $b$  passe de  $p$  à  $p' + 1$ . Si  $P$  est la somme des profondeurs des feuilles avant transformation et  $P'$  cette somme après transformation alors pour passer de  $P$  à  $P'$  on retranche :

$$P - P' = p' + 2p - (p - 1) - 2(p' + 1) = p - (p' + 1)$$

Comme  $p' < p - 1$ ,  $P - P'$  est un entier strictement positif. La moyenne des profondeurs décroît donc d'au moins 1 à chaque fois que l'on effectue la transformation.

Étant donné un arbre non équilibré on lui applique alors cette transformation tant que l'arbre obtenu n'est pas équilibré. Comme la somme des profondeurs des feuilles diminue d'au moins 1 à chaque transformation et qu'elle ne peut pas devenir négative, quel que soit l'arbre il ne peut y avoir qu'un nombre fini d'étapes de transformation. C'est donc qu'à un moment l'arbre devient équilibré. La moyenne des profondeurs des feuilles de l'arbre équilibré obtenu est alors un minorant strict de la moyenne des profondeurs des feuilles de l'arbre de départ.

Ce qui achève la démonstration du théorème suivant.

**Théorème 2.4.** *La complexité en moyenne (et en pire cas) d'un tri par comparaison, est minorée asymptotiquement par  $N \log N$  (c'est à dire en  $\Omega(N \log N)$ ).*

## 2.5 Tris en temps linéaire

Lorsque les clés obéissent à des propriétés particulières, les tris ne sont pas nécessairement fondés sur la comparaison. On peut alors trouver des résultats de complexité meilleurs que  $N \log N$ .

### 2.5.1 Tri du postier

Les lettres et colis postaux sont triés selon leurs adresses. Cette clé de tri est très particulière. Quel que soit la quantité de courrier, il est possible de faire un nombre borné de paquets par pays, puis de trier chaque paquet par ville (là encore le nombre est borné), puis par arrondissement, par rue, par numéro et enfin par nom (on simplifie). Le résultat est un tri linéaire en le nombre de lettres : à chaque étape répartir le courrier en paquets de destination différentes se fait en temps  $N$  et il y a un nombre borné d'étapes. Ce type de tri peut aussi s'appliquer à certaines données.

### 2.5.2 Tri par dénombrement

Pour trier des entiers (sans données satellites) dont on sait qu'ils sont dans un intervalle fixé, disons entre 0 et 9, il suffit de compter les entiers de chaque sorte, puis de reproduire ce décompte en sortie. Le comptage peut être effectué dans un tableau auxiliaire (ici de taille 10) en une passe sur le tableau en entrée. Ce qui fait un temps linéaire en la taille  $N$  du tableau. La production de la sortie se fait aussi en temps linéaire en  $N$ . Un tri comme celui-ci prend donc un temps linéaire. Ce type de tri, par dénombrement, peut être amélioré pour intégrer les données satellites tout en obtenant un tri en temps linéaire qui de plus est stable, et ce tant que l'espace des clés est linéaire en la taille de la donnée. Voir l'exercice 33.

### 2.5.3 Tri par base

Le tri par base permet de trier en temps linéaire des éléments dont les clés sont des entiers dont l'expression en base  $N$  est bornée par une constante  $k$ . Autrement dit si les entiers sont entre 0 et  $N^k - 1$  ce tri est linéaire. Il s'agit simplement d'appliquer un tri par dénombrement, stable, sur chaque terme successif de l'expression en base  $N$  de ces entiers, en commençant par les termes les moins significatifs.

**Deuxième partie**

**Structures de données,  
arbres**



## Chapitre 3

# Structures de données

Dans le chapitre précédent, nous avons utilisé des tableaux pour organiser des données ensemble. Il existe beaucoup d'autres structures de données que les tableaux qui répondent chacune à des besoins particuliers.

Une structure de données contient des données et est caractérisée par les opérations fondamentales (ou primitives) qui permettent d'accéder à ces données<sup>1</sup>. L'ajout et le retrait d'un élément sont deux opérations fondamentales que l'on retrouve dans toutes les structures de données. Le test à vide, qui répond *vrai* lorsque la structure ne contient aucun élément, est également une opération que l'on retrouvera dans toutes les structures de données.

D'autres primitives sont disponibles selon les structures. Par exemple dans les tableaux, on peut accéder à un élément à partir de son rang (son indice) pour lire ou modifier sa valeur.

Une *structure abstraite de données* est une structure de données pour laquelle la manière dont les données sont organisées en mémoire et la manière dont les primitives sont programmées ne sont pas renseignées. La seule information intéressante retenue des réalisations concrètes de ces structures (leurs implantations) est le coût en temps ou en espace des primitives, exprimé de manière asymptotique, c'est à dire leur complexité.

Cette complexité doit être faible : temps ou espace constant, sub-linéaire (log), plus rarement linéaire. Autrement l'opération ne mérite pas l'appellation *fondamentale*.

Par exemple, accéder à un élément dans un tableau à partir de son rang se fait en temps et en espace constant. Mais l'insertion et la suppression d'un

---

1. Pour les informaticiens, les structures de données entrent bien évidemment tout à fait dans le cadre de la programmation modulaire. En programmation objet, chaque structure de donnée sera une classe et les primitives seront ses méthodes publiques.

élément à un rang quelconque demande, en pire cas et en moyenne, un temps linéaire en la taille du tableau, car il faut décaler les éléments suivants.

Nous étudions dans ce chapitre trois structures de données élémentaires : la pile, la file et la file de priorité.

Si les tableaux sont directement implantés dans les langages de programmation, ce n'est pas nécessairement le cas des autres structures de données. Pour réaliser concrètement les piles et les files, il est courant d'utiliser des listes chaînées. Ce chapitre contient donc un rappel sur les listes chaînées et leur(s) implantation(s) en C, ainsi que les primitives qui leurs sont associées. On peut également utiliser des tableaux pour implanter les piles et les files (voir exercice 48).

Pour réaliser algorithmiquement, sinon concrètement, les files de priorités nous utilisons la structure arborescente de *tas* (encore appelée maximier, ou minimier). La présentation des *tas* étant liée à celle d'arborescence, les *tas* sont traités au chapitre suivant.

Ces trois structures de données, pile, file, file de priorité, ne fournissent pas d'opération fondamentale pour la recherche d'un élément quelconque.

Dans les implantations courantes de ces structures de données, l'opération de recherche serait en moyenne en  $\Theta(N)$ . Dans le chapitre suivant, nous verrons des structures de données (arbres de recherche et arbre rouge noir) qui optimisent la recherche d'un élément quelconque ( $\Theta(\log N)$  comme pour les tableaux triés) tout en préservant une insertion et une suppression en temps raisonnable ( $\Theta(\log N)$ , contrairement aux tableaux triés).

Comme pour les tableaux, pour chacune des trois structures de données, pile, file, file de priorité, on peut ajouter sans difficulté une opération  $\text{Taille}(S)$  en  $\Theta(1)$  qui renvoie le nombre d'élément contenus dans la structure de donnée  $S$  (il suffit de maintenir un compteur du nombre d'éléments à chaque ajout/retrait). Au niveau concret, on peut également compter comme primitive une opération de création (un constructeur) d'une nouvelle structure vide, en  $O(1)$ .

### 3.1 Structures de données abstraites : pile, file, file de priorité

En mathématiques, en particulier depuis l'axiomatisation de l'arithmétique par Peano, les entiers naturels (ensemble  $\mathbb{N}$ ) peuvent être définis comme les termes du langage :

$$n = 0 \mid S n$$

La notation  $S_n$  peut être vue comme l'application de la fonction successeur :  $S : n \mapsto n + 1$  à son argument, où les parenthèses  $S(n)$  ont été omises (pour faciliter la lecture). Ainsi  $SSSO$  désigne l'entier 3.

De même si  $A$  est un ensemble d'éléments  $a$ , les listes finies d'éléments de  $A$  (ou *mots* sur  $A$ ), dont l'ensemble est noté  $A^*$ , sont les termes du langage :

$$l := \varepsilon \mid a :: l$$

La notation  $\varepsilon$  désigne la liste vide. Ainsi, si  $A = \mathbb{N}$ , la notation  $2 :: 12 :: 0 :: \varepsilon$  désigne la liste d'entiers 2, 12, 0. Pour simplifier, on omet le mot vide  $a_1 :: \dots :: a_n :: \varepsilon = a_1 :: \dots :: a_n$ . Ceci permet également de former des termes  $l :: a$  (ajout de  $a$  en fin de liste) ou  $l :: l'$  (concaténation de deux listes).

### 3.1.1 Piles

En anglais : *stacks*

Une pile est une structure de donnée que l'on rencontre très souvent dans la vie de tous les jours : pile de papiers, pile d'assiettes (ou si on est programmeur, pile d'exécution). Sa principale caractéristique est que l'élément que l'on retire est toujours le dernier élément ajouté et qui n'a pas encore été retiré, que nous appellerons *sommet* de la pile. On parle de structure LIFO (*last in first out*). Les opérations fondamentales sont :

- le test à vide,  $\text{EstVide}(P)$ ,
- l'ajout d'un élément (appelé empilement),  $\text{Empiler}(e, P)$ ,
- le retrait (dépilement),  $\text{Dépiler}(P)$ , qui retire le dernier élément empilé non encore retiré et le renvoie,
- et la lecture de l'élément au sommet de la pile,  $\text{Sommet}(P)$ , qui rend la même valeur que  $\text{Dépiler}(P)$  sans modifier la pile.

Ces quatre opérations peuvent être réalisées de manière à ce que leurs temps d'exécution soit constant, c'est à dire en  $\Theta(1)$  et avec en espace constant.

En termes plus mathématiques, les piles peuvent être vues comme des listes d'éléments sur lesquelles les opérations disponibles sont définies par :

$$\begin{aligned} \text{EstVide}(\varepsilon) &= \text{vrai} \\ \text{EstVide}(e :: l) &= \text{faux} \\ \text{Empiler}(e, l) &= e :: l \\ \text{Dépiler}(e :: l) &= (e, l) \\ \text{Dépiler}(\varepsilon) &\text{ est indéfini} \\ \text{Sommet}(e :: l) &= e \\ \text{Sommet}(\varepsilon) &\text{ est indéfini.} \end{aligned}$$

Du point de vue mathématique, Dépiler renvoie une paire : élément, pile.<sup>2</sup> Mais nous utiliserons toujours la notation au sens informatique, comme dans :  $e = \text{Dépiler}(P)$ , en laissant aux effets de bord le soin de modifier la pile.

### 3.1.2 Files

En anglais : *queues*

Une file est une structure de donnée que l'on rencontre aussi très souvent dans la vie de tous les jours : la file d'attente. L'élément que l'on retire est toujours le premier élément ajouté et qui n'a pas encore été retiré. On l'appelle tête de la file. Le dernier élément ajouté est l'élément de queue. On parle alors de structure FIFO (*first in first out*). Les opérations fondamentales sont les mêmes que celles des piles :

- EstVide( $F$ ), le test à vide,
- AjouterFile( $e, F$ ), (ou InsérerFile( $e, F$ )) ajoute l'élément  $e$  dans la file,
- RetirerFile( $F$ ), retire l'élément de tête la file et le renvoie,
- TeteFile( $F$ ) renvoie l'élément de tête sans modifier la file.

Ces quatre opérations peuvent être réalisées de manière à ce que leurs temps d'exécution soit toujours constant, c'est à dire en  $\Theta(1)$ , et en espace constant.

En termes plus mathématiques, les files peuvent être vues comme des listes d'éléments sur lesquelles les opérations disponibles sont définies par :

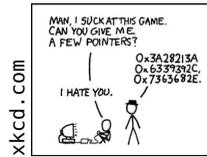
$$\begin{aligned} \text{EstVide}(\varepsilon) &= \text{vrai} \\ \text{EstVide}(e :: l) &= \text{faux} \\ \text{AjouterFile}(e, l) &= l :: e \\ \text{RetirerFile}(e :: l) &= (e, l) \\ \text{RetirerFile}(\varepsilon) &\text{ est indéfini} \\ \text{TeteFile}(e :: l) &= e \\ \text{TeteFile}(\varepsilon) &\text{ est indéfini.} \end{aligned}$$

### 3.1.3 File de priorité

Une file de priorité est une structure de donnée qui permet de stocker des éléments ayant une *priorité* – comme pour les tris chaque élément possède une clé, sa priorité, ainsi que des données satellites. La particularité des files de priorités est que l'opération de retrait d'un élément rend toujours un élément de priorité maximum (ainsi que l'opération de lecture). Les primitives

---

2. Ceci fait qu'il est mathématiquement correct d'écrire  $\text{Empiler}(\text{Dépiler}(l))$  qui vaut alors  $l$  lorsque  $\text{Dépiler}(l)$  est défini (i.e. lorsque  $l$  n'est pas vide).



sont similaires à celles des piles et des files, plus une primitive qui permet de modifier la priorité d'un élément :

- EstVide( $F$ ), test à vide ;
- Insertion( $e, F$ ), insertion ;
- RetraitMax( $F$ ), retrait
- LectureMax( $F$ ), lecture du prochain élément (renvoie l'élément maximum sans modifier la file de priorité) ;
- Modifier( $x, e, F$ ) qui permet de modifier un élément quelconque de la file, et en particulier sa priorité. Pour cela il faut connaître l'adresse  $x$  de l'élément dans la file. L'argument  $e$  est la nouvelle valeur de l'élément, elle contient en particulier sa nouvelle priorité.

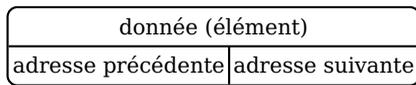
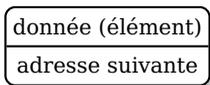
Une réalisation des files de priorités à l'aide de tas (vue au prochain chapitre) permet de rendre :

- constant  $\Theta(1)$  le temps d'exécution des opérations de test à vide et de lecture du maximum ;
- et en  $\Theta(\log N)$ , où  $N$  est le nombre d'éléments, le temps d'exécution en pire cas des opération d'insertion, de retrait et de modification ;
- de plus toutes ces opérations sont en espace constant.

## 3.2 Listes chaînées en C

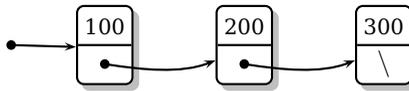
En anglais : *linked lists*

**Liste chaînée.** Une liste chaînée est une structure de donnée séquentielle qui permet de stocker une suite d'éléments en mémoire. Chaque élément est stocké dans une cellule et les cellules sont *chaînées* : chaque cellule contient, en plus d'un élément, l'adresse de la cellule suivante, et éventuellement l'adresse de la cellule précédente dans le cas des listes doublement chaînées. On peut représenter graphiquement une cellule de liste simplement chaînée par une boîte contenant deux champs et une cellule de liste doublement chaînée par une boîte contenant trois champs :



Au lieu d'écrire l'adresse mémoire d'une cellule, on représente cette adresse par une flèche vers cette cellule. La liste simplement chaînée des entiers 100,

200, 300 se représente ainsi :



On utilise une valeur spéciale, \ dans la notation graphique et NULL en C, pour signaler une adresse qui ne mène nulle part. Dans le cas des listes simplement chaînées cette valeur signale la fin de la liste.

La première cellule de la liste s'appelle la *tête* de la liste et le reste de la liste la *queue*. D'un point de vue mathématique, les listes chaînées sont définies par induction en disant qu'une liste est soit la liste vide, soit un élément (la tête) suivi d'une liste (la queue).

Le type des listes simplement chaînées peut être défini en C par les instructions :

```
struct cellsimple_s {
    element_t element;
    struct cellsimple_s * suivant;
};

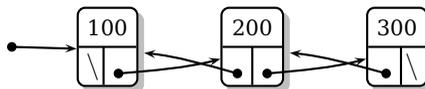
typedef struct cellsimple_s * listesimple_t;
```

Ces instructions déclarent :

1. qu'une cellule est une structure contenant un élément (element) et un pointeur sur une cellule (suivant);
2. qu'une liste est un pointeur sur une cellule (une variable contenant l'adresse d'une cellule).

La liste vide sera simplement égale au pointeur NULL.

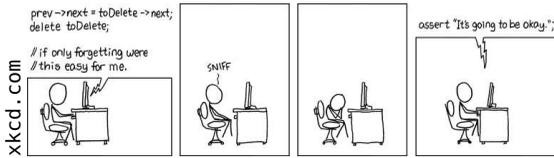
La variante doublement chaînée de la liste précédente se représente ainsi :



Le type des listes doublement chaînées peut être défini en C par :

```
struct celldouble_s {
    element_t element;
    struct celldouble_s * precedent;
    struct celldouble_s * suivant;
};

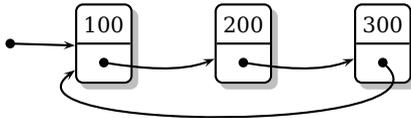
typedef struct celldouble_s * listedouble_t;
```



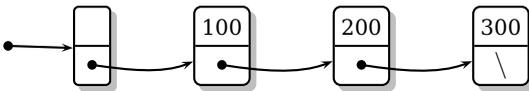
qui ne diffère du type des listes simplement chaînées que par l'ajout du champ precedant dans les cellules.

Dans les deux cas, listes simplement chaînées et listes doublement chaînées, on peut introduire des variantes dans la représentation des listes. En voici deux qui ne changent pas la manière de définir le type des listes en C, mais changent la manière de les manipuler.

**Listes circulaires.** On peut utiliser des listes circulaires, où après le dernier élément de la liste on revient au premier. Graphiquement on fait comme ceci :



**Utilisation d'une sentinelle.** Il est parfois plus facile de travailler avec des listes chaînées dont le premier élément ne change jamais. Pour cela on introduit une première cellule *sentinelle* dont l'élément ne compte pas. La liste vide contient alors cette unique cellule. L'avantage est que la modification (insertion, suppression) du premier élément de la liste (qui est alors celui dans la seconde cellule) ne nécessite pas de mettre à jour le pointeur qui indique le début de la liste (voir l'exemple de code C pour la suppression avec et sans sentinelle un peu plus loin). La liste de notre exemple, se représente alors comme ceci :



On ne tient en général jamais compte de la valeur de l'élément stocké dans la cellule sentinelle.

### 3.2.1 Opérations fondamentales

Les listes chaînées sont une réalisation concrète (plus ou moins complète selon les variantes) d'une structure de donnée abstraite *liste*. Les opérations primitives que nous retenons sur les listes chaînées sont : le test à vide ; l'insertion d'un élément en tête de liste ; la lecture de l'élément en tête de liste ; la suppression de l'élément en tête de liste. C'est à dire les opérations qui correspondent au fonctionnement d'une liste comme une pile. D'autres opérations

primitives sont possibles, par exemple avec des listes doublement chaînées : l'ajout en fin de liste et la concaténation de deux listes, mais nous n'en parlons pas dans ce cours.

Voici une réalisation des primitives à l'aide de fonctions en C, pour les listes simplement chaînées (non circulaires et sans sentinelle). Lorsque ces fonctions fonctionnent sans modification pour les listes doublement chaînées, nous utilisons un type générique `liste_t`.

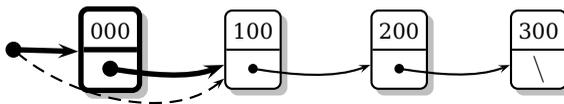
**Test à vide.** Le test à vide prend une liste en paramètre, renvoie vrai (ou 1) si la liste est vide et faux (0) sinon.

```
int listeVide(liste_t x){
    if (x == NULL) return 1;
    else return 0;
}
```

**Lecture.** La lecture de l'élément en tête de liste, prend une liste et renvoie la valeur de son élément de tête.

```
element_t lectureTete(liste_t x){
    return x->element;
}
```

**Insertion.** L'insertion d'un élément en tête d'une liste prend en paramètre une liste et un élément et ajoute une cellule en tête de la liste contenant cet élément. Le schéma est alors le suivant :



Ceci modifie la liste et il est nécessaire de récupérer sa nouvelle valeur. On peut renvoyer la nouvelle liste comme valeur de retour de la fonction (`insérerListe1`), où bien utiliser la liste comme un argument-résultat en passant son adresse à la fonction (`insérerListe2`).

```
listesimple_t insererListe1(listesimple_t x, element_t e){
    listesimple_t y;
    y = malloc(sizeof(*y)); /* Création d'une nouvelle cellule. */
    if (y == NULL) perror("échec malloc");
    y->element = e;
    y->suivant = x;
    // y->precedant = NULL; <-- Pour les listes doublement chaînées
    // x->precedant = y; <--
```

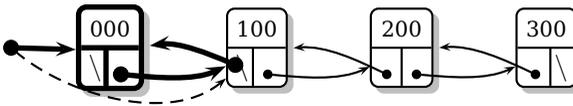
```

return y;
}

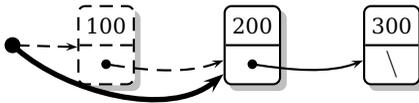
void insererListe2(listesimple_t * px, element_t e){
    listesimple_t y;
    y = malloc(sizeof(*y)); /* Création d'une nouvelle cellule. */
    if (y == NULL) perror("échec malloc");
    y->element = e;
    y->suivant = *px;
    // y->precedant = NULL; <-- Pour les listes doublement chaînées
    // (*px)->precedant = y; <--
    *px = y;
}

```

Pour les listes doublement chaînées le schéma serait le suivant :



### Suppression.



La suppression de l'élément en tête de liste prend une liste et supprime sa première cellule. Comme pour l'insertion, il faut mettre à jour la valeur de la liste. Soit on renvoie cette valeur en sortie de fonction (`suppressionListe1`) soit on passe la liste par adresse (`suppressionListe2`). À titre d'exemple, la fonction `suppressionListeSentinelle` montre le code de la fonction de suppression lorsque les listes ont une sentinelle (c'est alors la deuxième cellule qu'il faut supprimer).

```

listesimple_t supprimerListe1(listesimple_t x){
    listesimple_t y;
    y = x->suivant;
    free(x); /* Suppression de la première cellule */
    return y; /* On renvoie la cellule suivante */
}

void supprimerListe2(listesimple_t * px){
    listesimple_t y;
    y = *px;
    *px = y->suivant; /* Fait pointer px sur la liste commençant à
    la deuxième cellule */
}

```

```

    free(y); /* Suppression de la première cellule */
}

void supprimerListeSentinelle(listesimple_t x){
    listesimple_t y;
    y = x->suivant;
    x->suivant = y->suivant; /* On fait commencer la liste à l'élément
suivant */
    free(y); /* Suppression du premier élément (seconde cellule) */
}

```

**Autres opérations.** Selon les variantes, d'autres opérations fondamentales sont possibles. Ainsi si on accède en temps constant au dernier élément de la liste (variante doublement chaînées) on peut réaliser la concaténation de deux listes en temps constant.

### 3.2.2 Réalisation des piles avec des listes chaînées

Les listes chaînées peuvent être utilisées pour programmer les piles de manière à ce que leurs primitives s'exécutent en temps constant  $\Theta(1)$ . Le sommet de la pile correspond à la tête de la liste. Le test à vide correspond au test à vide des listes chaînées, l'empilement correspond à l'insertion en tête, la lecture du sommet de la pile correspond à la lecture de l'élément de tête et le dépilement à une combinaison de lecture de l'élément de tête et de suppression. La fonction de dépilement modifie la pile et rend un élément. Elle peut s'écrire comme ceci, en passant la pile par adresse (argument-résultat) :

```

typedef listesimple_t pile_t; /* les piles sont réalisées par les listes */

element_t depiler(pile_t * px){
    pile_t y;
    element_t e;
    y = *px; /* y pointe sur le haut de la pile */
    e = y->element;
    *px = y->suivant; /* la pile commence désormais à l'élément du dessous */
    free(y); /* libère la mémoire devenue inutilisée */
    return e; /* rend l'ancien haut de pile */
}

```

### 3.2.3 Réalisation des files avec des listes chaînées

On peut réaliser une file à l'aide d'une liste simplement chaînée de manière à ce que les opérations fondamentales se fassent en temps constant  $\Theta(1)$ . Il faut enrichir un peu la structure : l'ajout se faisant à un bout de la

liste et la suppression à l'autre bout, une file doit fournir l'adresse du premier et du dernier élément de la liste chaînée de ses éléments. La tête de la liste sera la tête de file (où se fait le retrait) et le dernier élément de la liste son élément de queue (où se fait l'ajout). Voici le code C d'une réalisation des files basée sur les listes simplement chaînées.

```
typedef struct {
    liste_t tete;
    liste_t fin;
} * file_t;

int fileVide(file_t x){
    if (x->tete == NULL) return 1;
    else return 0;
}

element_t teteFile(file_t x){
    return (x->tete)->element; /* valeur de l'élément en tête de file */
}

element_t retirerFile(file_t x){
    element_t e;
    liste_t y;
    y = x->tete; /* le début de la file */
    e = y->element; /* l'élément de tête */
    x->tete = y->suivant; /* la file débute à l'élément suivant */
    free(y);
    return e;
}

void ajouterFile(file_t x, element_t e){
    liste_t y;
    /* création de la cellule y, qui contiendra e */
    y = malloc(sizeof(*y));
    if (y == NULL) perror("panne mémoire");
    y->element = e;
    /* On place y à la fin de la liste */
    y->suivant = NULL;
    if (estVide(x)) x->tete = y; /* si x était vide, y devient la tete de liste */
    else (x->fin)->suivant = y; /* sinon, chaînage des deux derniers éléments */
    x->fin = y; /* Le pointeur de fin de file est désormais sur la cellule y */
}
```

## Chapitre 4



# Arborescences

Dans ce chapitre, nous présentons d'abord les arbres (ou arborescences) en général, en commençant par les arbres binaires.

Nous nous focalisons ensuite sur les arbres binaires quasi-parfaits, sur lesquelles repose la structure algorithmique de tas. Les tas permettent d'implanter les files de priorités du chapitre précédent. Plus précisément les tas forment un matériel algorithmique parfaitement adapté pour implanter la structure de données abstraite *file de priorité* et l'implantation concrète dépend encore de l'implantation des arbres binaires quasi-parfaits sur machine. Les tas permettent également de construire un algorithme de tri en place en  $O(N \log N)$  en pire cas (le tri par tas).

Les arbres binaire quasi-parfaits sont tellement particuliers que l'on peut les mettre en œuvre sans vraiment avoir recours aux structures de données concrètes utilisées habituellement pour les arbres : de simples tableaux suffisent. En ce sens, il s'agit d'une notion très dégénérée d'arbres. Par exemple, pour un nombre de nœuds  $n$  donné, un seul arbre binaire est quasi-parfait alors qu'il y a habituellement un nombre très important d'arbres binaires à  $n$  nœuds (ce nombre est plus qu'exponentiel en  $n$ ).

La dernière partie du chapitre est consacrée aux arbres binaires de recherche ce qui nous permettra de revenir plus sérieusement sur la notion d'arbre binaire.

**Définition 4.1.** Une *arborescence binaire* est une structure de donnée contenant un nombre fini d'éléments rangés dans des *nœuds*, telle que :

- lorsque la structure n'est pas vide, un nœud particulier, unique, appelé *racine* sert de point de départ ;

- tout nœud  $x$  autre que la racine fait référence de manière unique à un autre nœud, appelé son *parent* et la racine n'a pas de parent;
- chaque nœud  $x$  peut faire référence à zéro, un ou deux nœuds fils : un fils gauche et un fils droit, chacun de ces fils a alors  $x$  pour parent;
- Tous les nœuds ont la racine pour ancêtre commun (parent, ou parent de parent, ou parent de parent de parent, etc.).

Comme pour les listes chaînées il est utile de se donner une représentation graphique sous forme de cellules (ici on parlera de nœud). Cette représentation, le type en C correspondant et un exemple sont données figure 4.1.

Il y a au moins quatre opérations de lecture (qui ne modifient pas la structure) communes à toutes les structures de données basées sur les arborescences binaires : le test à vide qui prend un arbre en argument et rend vrai s'il ne contient pas de nœud et les trois opérations de lecture des références entre les nœuds. Ces trois opérations prennent en entrée un nœud. L'une rend son nœud parent (lorsqu'il existe), et les deux autres rendent respectivement le fils gauche et le fils droit (lorsqu'ils existent).

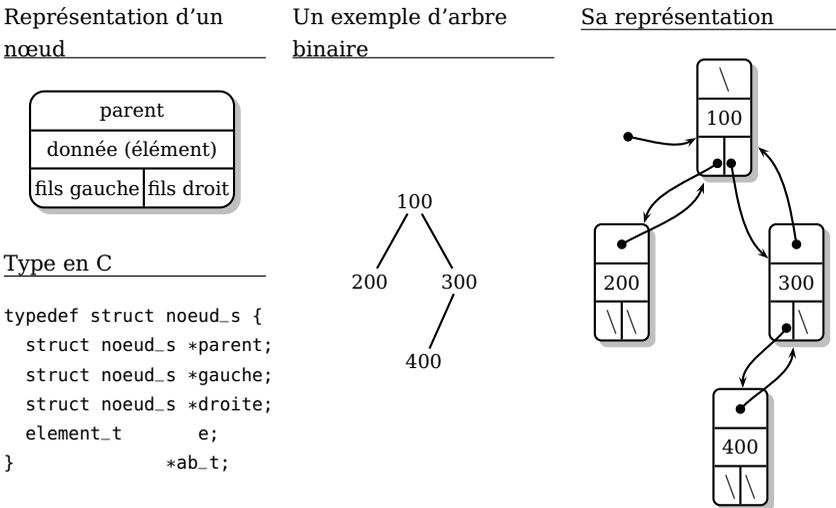
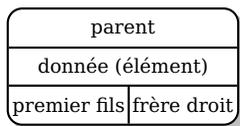


Figure 4.1 – Représentation des arborescences binaires

**Définition 4.2.** Plus généralement, une *arborescence* est comme une arborescence binaire mais où les fils d'un nœud peuvent aussi être en nombre supérieur à deux et sont donnés par une liste ordonnée.

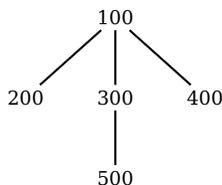
### Représentation d'un nœud



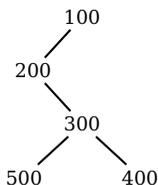
### Type en C

```
typedef struct noeud_s {
    struct noeud_s *parent;
    struct noeud_s *fils;
    struct noeud_s *frere;
    element_t      e;
}                 *arbre_t;
```

### Un exemple d'arbre



### Arbre binaire correspondant



### Sa représentation

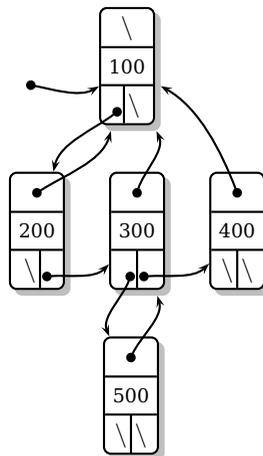


Figure 4.2 – Arbres en représentation fils gauche frère droit

Attention : dans une arborescence binaire on fait la distinction entre un nœud ayant seulement un fils gauche et un nœud ayant seulement un fils droit mais dans une arborescence (non binaire) on ne la fait pas (il s'agit deux fois d'un nœud ayant un seul fils).

Pour représenter les arbres binaires plutôt que de manipuler les deux structures que sont le nœud d'un arbre (élément, référence au parent, liste des fils) et la cellule de liste chaînée (nœud contenu, référence à la cellule suivante) utilisée pour représenter la liste des fils, il est plus facile de regrouper tout cela dans une même structure. La représentation obtenue s'appelle *fils gauche frère droit*. Elle revient en fait à représenter n'importe quelle arborescence à l'aide d'une arborescence binaire. La notion de fils gauche correspond à la notion de premier fils dans la liste des fils. Et la notion de fils droit correspond à celle de nœud suivant dans la liste des fils, autrement dit, à la notion de frère droit. La représentation graphique, le type en C et un exemple sont données figure 4.2.

En mathématiques, on peut donner une autre définition des notions d'arborescence et d'arborescence binaire équivalentes à celles-ci. Nous ne rentrerons pas dans ces détails. Une chose importante à retenir est qu'en mathématiques on parle plus volontiers d'arbre et que dans ce cas, en générale, on désigne une structure dans laquelle on a pas encore choisi de nœud particulier

pour jouer le rôle de la racine. Ainsi il y a une différence subtile entre la définition mathématique d'arbre et celle d'arborescence : une arborescence est un arbre dans lequel on a choisi une racine. Quelques mathématiciens parlent plutôt d'algue pour les arbres sans racine. Dans ce cours nous prenons le parti d'appeler arbre une structure arborescente et donc de donner une racine aux arbres.

La distance d'un nœud  $x$  à la racine est le nombre de fois où il faut remonter à un parent pour passer de  $x$  à la racine : si ce nœud est la racine cette distance est 0, si le parent de  $x$  est la racine c'est 1, etc.

la *profondeur* d'un nœud  $x$  dans un arbre est sa distance à la racine. La profondeur de la racine est donc 0, de ses fils éventuels 1, etc. Nous définissons la *hauteur* d'un arbre comme le maximum des profondeurs de ses nœuds, plus un (ce « plus un » est affaire de convention, et cette convention n'est pas la même partout). La hauteur d'un nœud est la différence entre la hauteur de l'arbre et la profondeur du nœud.

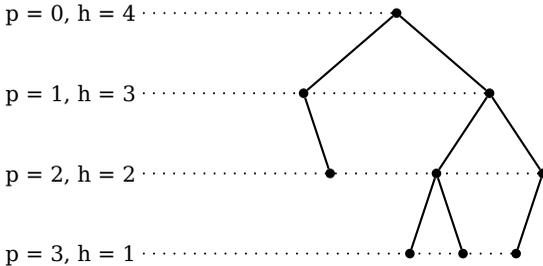


Figure 4.3 – Exemple d'arbre binaire avec hauteur et profondeur des nœuds

Un nœud qui n'as pas de descendant est une *feuille*. Un nœud qui n'est pas une feuille est parfois dénommé nœud interne (ou encore nœud *strict*).

## 4.1 Arbres binaires parfaits et quasi-parfaits

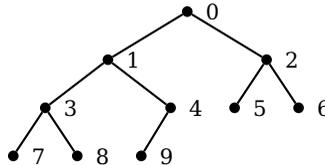
Rappelons qu'un arbre binaire est complet lorsque ses nœuds internes ont leurs deux descendants.

On appelle arbre binaire *parfait* un arbre binaire ayant  $2^h - 1$  nœuds où  $h$  est sa hauteur.

Un arbre binaire parfait est complet. À profondeur  $p \leq h - 1$  on a  $2^p$  nœuds. Si  $p = h - 1$  ces nœuds sont les feuilles de l'arbre.

**Arbre binaire quasi-parfait.** Ordonnons les nœuds d'un arbre binaire parfait selon leur profondeur puis de gauche à droite. En premier on a la racine,

puis ensuite ses deux fils (de profondeur 1), le gauche, puis le droit, ensuite les nœuds de profondeur 2, d'abord les fils du fils gauche de la racine, le gauche puis le droit, puis les fils du fils droit de la racine *etc.* Si on supprime un nombre quelconque de nœuds, en partant des derniers pour cet ordre, on obtient ce qu'on appelle un arbre binaire *quasi-parfait*. Voici un exemple d'arbre quasi-parfait où les nœuds sont numérotés dans l'ordre, en commençant à zéro. On ne représente pas les éléments contenus dans les nœuds.



Un arbre binaire quasi-parfait n'est pas nécessairement complet : le parent du dernier nœud peut ne pas avoir de fils droit, comme dans l'exemple.

Étant donné un nombre  $N$  fixé, il n'y a qu'un seul arbre parfait à  $N$  nœuds. La hauteur de cet arbre est  $h = \lceil \log N \rceil + 1$  donc en  $\Theta(\log N)$ .

On peut stocker un arbre binaire quasi-parfait dans un tableau en plaçant ses éléments dans l'ordre (la racine à l'indice 0, *etc.*). On fait alors référence aux nœuds par leurs indices dans le tableau. Partant d'un nœud d'indice  $i$  on trouve : son nœud parent à l'indice  $\text{Parent}(i)$  son nœud fils gauche à l'indice  $\text{Gauche}(i)$  et son nœud fils droit à l'indice  $\text{Droite}(i)$ , avec :

$$\text{Parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$$

$$\text{Gauche}(i) = 2i + 1$$

$$\text{Droite}(i) = 2i + 2.$$

Voici un exemple d'implantation des arbres quasi-parfaits en C (voir aussi le TD pour des variantes dans le choix de l'indice de la racine, et de l'ordre dans lequel sont écrits les nœuds – ordre direct ou ordre inverse). On adjoint à la donnée du tableau quelques données auxiliaires : le nombre d'éléments de l'arbre et l'espace mémoire disponible dans le tableau. Ces données permettent de gérer plus efficacement la mémoire lors de l'ajout et de la suppression d'éléments (voir fichier C inclus en fin de chapitre).

```

typedef struct {
    element_t *tab; // le tableau
    int mem;       // Nombre maximum d'éléments dans le tableau
    int taille;    // nombre d'éléments de l'arbre (mem => taille)
}

```

```

} *arbrebqp_t;

int parent(int i){// parent du noeud i
    return (i - 1)/2;
}

int gauche(int i){// gauche du noeud i
    return 2*i + 1;
}

int droite(int i){// descendant droit du noeud i
    return 2*i + 2;
}

int racine(arbrebqp_t x){// racine de l'arbre
    return 0;
}

int dernier(arbrebqp_t x){// indice du dernier élément
    return x->taille - 1;
}

int taillearbrebqp(arbrebqp_t x){// nb d'éléments dans l'arbre
    return x->taille;
}

```

## 4.2 Tas et files de priorité

Un tas est une structure de donnée basée sur les arbres binaires quasi-parfaits, qui réalise efficacement les files de priorité, c'est à dire avec de bons résultats de complexité algorithmique sur les opérations fondamentale : ajout, retrait, modification de la priorité d'un élément se font en  $O(\log N)$ .

**Définition 4.3.** Un *tas max* (respectivement *tas min*), également appelé *maximier* (resp. *minimier*), est un arbre binaire quasi-parfait tel que tout nœud différent de la racine possède une clé (ou priorité) plus petite (resp. plus grande) que celle de son parent ( $T[i] \leq T[\text{Parent}(i)]$ ).

Les deux notions, tas max et tas min, sont symétriques, il suffit d'inverser l'ordre des clés pour passer de l'une à l'autre. On travaille ici avec des tas max.

**Propriétés:** Première conséquence de la définition de tas : dans un tas max non vide, l'élément de clé maximum est toujours à la racine.

Tout sous-arbre obtenu à partir d'un tas en sélectionnant un nœud et tous ses descendants est encore un tas.

*Remarque 4.4.* Un tableau trié en ordre décroissant est un tas max. Mais il existe plusieurs manières différentes de structurer une liste d'éléments donnée sous la forme d'un tas.

Pour la suite, concernant l'implantation en C, on se donne une macro pour le pré-processeur pour simplifier l'adressage des éléments d'un arbre quasi-parfait cette macro ne marche que si l'arbre est noté  $x$ ). On se donne également une fonction d'échange similaire à celle des tableaux.

```
/* Macro pour simplifier l'accès aux tableau. */
#define x(indice) (x->tab[indice])
/* Exemple : x(i/2) sera remplacé par (x->tab[i/2]) */

void echangetas(arbrebqp_t x, int i, int j){
    element_t e;
    e = x(i);
    x(i) = x(j);
    x(j) = e;
}
```

### 4.2.1 Changement de priorité d'un élément dans un tas

Changer la priorité (clé) d'un élément dans un tas peut rendre un arbre qui ne satisfait plus la propriété de tas. On doit alors modifier l'arrangement des éléments dans l'arbre pour rétablir cette propriété. On dit qu'on maintient la propriété de tas. La forme même de l'arbre reste inchangée, puisque le nombre d'éléments ne change pas.

**Augmentation : maintien vers le haut.** Si la priorité d'un élément  $e$  dans un nœud  $i$  augmente, il est possible qu'elle dépasse celle de son parent (lorsqu'il existe). Dans ce cas, on fait appel à une procédure `MaintienHaut( $x, i$ )`. Cette procédure fait remonter l'élément en l'échangeant avec l'élément  $e'$  de son parent  $j = \text{Parent}(i)$ . Les descendants du nœud  $i$  ont bien des priorités inférieures à celle de  $e'$  puisque c'était le cas avant le changement de la priorité de l'élément  $e$ . Par contre il est possible que l'élément  $e$  qui est maintenant dans le nœud  $j$  n'ait pas une priorité inférieure à celle de son parent. C'est pourquoi, `MaintienHaut` fait appel récursivement à elle-même sur le nœud  $j$ .

Ceci a pour effet de faire remonter l'élément  $e$  vers la racine par échanges, jusqu'à ce qu'il se trouve dans un nœud dont le parent a une priorité supérieure, ou bien qu'il est atteint à la racine. Le temps d'exécution du maintien vers le haut est borné par la hauteur de l'arbre, il est donc en  $O(\log N)$  où  $N$  est le nombre d'éléments du tas.

La fonction `maintienHaut` est une implantation en C de cette opération de maintien vers le haut.

```
void maintienHaut(arbrebqp_t x, int i){
    if ( ( i != racine(x)) && (x(parent(i)).cle < x(i).cle) ){
        echangetas(x, parent(i), i);
        maintienHaut(x, parent(i));
    }
}
```

**Diminution : maintien vers le bas.** Si la priorité d'un élément  $e$  dans un nœud  $i$  diminue, il est possible qu'elle devienne inférieure à la priorité de l'un de ses deux descendants, ou des deux. Le maintien est alors effectué par une procédure `MaintienBas( $x, i$ )` qui fonctionne de la manière suivante. On cherche parmi les nœuds  $i$ , `Gauche( $i$ )` et `Droite( $i$ )` lequel renferme l'élément de priorité maximale. Appelons  $j$  ce nœud. Il y a deux cas. Si  $j = i$ , il n'y a rien à faire la propriété de tas n'a pas été violée par la diminution de la priorité de  $e$ . Si  $j$  est l'un des fils de  $i$ , on échange le contenu des nœuds  $i$  et  $j$ . Cela a pour effet de rétablir localement la propriété de tas entre les nœuds  $i$ , `Gauche( $i$ )` et `Droite( $i$ )`. Mais cette propriété peut maintenant être violée par le nœud  $j$  (l'un des fils), parce que sa priorité a diminué. On rappelle donc récursivement la procédure `MaintienBas` sur  $j$ . Ceci a pour effet de faire descendre  $e$  dans l'arbre par échanges successifs, jusqu'à ce qu'il se trouve dans un nœud dont chacun des fils a une priorité inférieure à celle de  $e$ . Le temps d'exécution du maintien vers le bas est borné par la hauteur de l'arbre, il est donc en  $O(\log N)$  où  $N$  est le nombre d'éléments.

La fonction `maintienBas` est une implantation en C de cette opération de maintien. Deux exemples d'exécution de cette fonction sont donnés ci-dessous : le nœud grisé est celui d'indice  $i$  (dans les deux exemples, au départ c'est la racine).

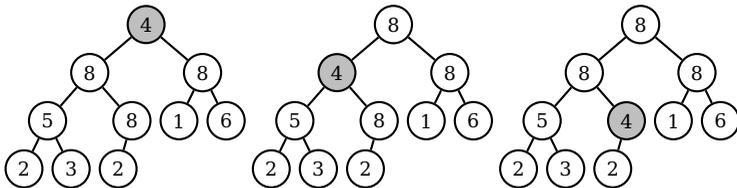
```
void maintienBas(arbrebqp_t x, int i){
    int imax;
    imax = i; /* option affichage */ AFFICHAGE_TAS_X_i;
    /* On cherche l'indice de l'élément maximum parmi i, gauche(i) et
       droite(i), lorsqu'ils existent */
```

```

if ( (gauche(i) <= dernier(x)) && (x(gauche(i)).cle > x(imax).cle) )
    imax = gauche(i);
if ( (droite(i) <= dernier(x)) && (x(droite(i)).cle > x(imax).cle) )
    imax = droite(i);
/* Si ce maximum n'est pas en i on procède à un échange et on
   relance sur le noeud qui contenait le maximum */
if ( imax != i ) {
    echangetas(x, i, imax);
    maintienBas(x, imax);
}
}

```

Voici un exemple de maintien bas, appliqué à la racine de l'arbre (puis récursivement au nœud grisé).



## 4.2.2 Ajout et retrait des éléments

**Ajout d'un élément.** Pour ajouter un élément  $e$  dans un tas ( $\text{InsérerTas}(x, e)$ ), on ajoute en fin de tableau un nœud  $n$  dans lequel on stocke  $e$  (sans modifier le contenu des autres nœuds). Le nœud  $n$  devient ainsi la dernière feuille de l'arbre. Le nœud  $n$  n'ayant pas de descendant la seule violation possible de la propriété de tas qu'ait pu introduire cette insertion est entre le nœud  $n$  et son parent. Il est en effet, possible que la priorité de  $e$  soit supérieure à celle du parent de  $n$ . On appelle donc la procédure de maintien vers le haut sur le nœud  $n$ . Ainsi l'opération d'insertion est en  $O(\log N)$  où  $N$  est le nombre d'éléments du tas.

```

void insererTas(arbrebqp_t x, element_t e){
    augmenterArbreBQP(x); /* Fait : x->taille++ et gestion mémoire */
    x(dernier(x)) = e;
    maintienHaut(x, dernier(x));
}

```

**Retrait du maximum.** Le retrait d'un élément dans un tas ( $\text{RetirerMax}(x)$ ) se fait toujours par retrait de l'élément de priorité maximum, c'est dire l'élément à la racine de l'arbre. Pour que la structure d'arbre quasi-parfait reste correcte, après avoir retiré l'élément à la racine, on le remplace par l'élément de la dernière feuille de l'arbre (le dernier élément du tableau) et on supprime cette feuille. Comme la nouvelle priorité de la racine est inférieure à l'ancienne, on appelle sur la racine, la procédure de maintien vers le bas de la propriété de tas. Le retrait du maximum est ainsi en  $O(\log N)$ .

```

element_t retirerMax(arbrebqp_t x){
    element_t e;
    /* On prend l'élément maximum à la racine */
    e = x(racine(x));
    /* On remplace la racine par le dernier élément */
    x(racine(x)) = x(dernier(x));
    /* On diminue le nombre d'élément de l'arbre de un */
    diminuerArbreBQP(x); /* Fait : x->taille-- et gestion mémoire */
    /* On reforme le tas */
    maintienBas(x, racine(x));
    /* On sort en rendant l'élément maximum */
    return e;
}

```

### 4.2.3 Formation d'un tas

**Par une série d'insertions.** Pour former un tas à partir d'une liste de  $N$  éléments, on peut commencer par un tas vide et lui ajouter les éléments un à un. Cet algorithme a une complexité en pire cas en  $O(N \log N)$  échanges.

En effet, l'ajout d'un élément dans un tas de  $k - 1$  éléments prend au pire des cas  $h - 1$  échanges où  $h$  est la hauteur de l'arbre obtenu après l'ajout. Mais  $2^{h-1} - 1 < k$  donc  $2^{h-1} \leq k$  ce qui donne :  $h - 1 \leq \log k$ , par passage au log. Le nombre d'échanges en pire cas d'un ajout est donc inférieur à  $\log k$ .

On effectue l'ajout  $N$  fois, en commençant avec un tas à zéro élément puis en augmentant de un son nombre d'éléments à chaque fois. Le nombre total d'échanges est borné supérieurement par :

$$\sum_{k=1}^N \log k = \log \prod_{k=1}^N k = \log(N!) = O(N \log N).$$

**Algorithme en temps linéaire en pire cas.** Il est toutefois possible de faire mieux que  $O(N \log N)$ , en changeant d'algorithme. On part de l'arbre quasi-

parfait  $A$  contenant tous les éléments et on applique à ses nœuds internes, en allant du dernier au premier, la procédure de maintien vers le bas de la structure de tas.

À la fin de ce nouvel algorithme, on obtient un tas. En effet, un arbre réduit à un seul nœud est toujours un tas. Donc dans  $A$  chaque feuille est déjà un tas. Et si on applique l'algorithme de maintien vers le bas à un nœud dont les fils sont déjà des racines de tas, alors l'arbre qui a pour racine ce nœud devient à son tour un tas. Ainsi, en procédant du dernier au premier nœud interne, à chaque étape, le nœud qui vient d'être traité est la racine d'un tas. Comme le dernier élément traité est la racine,  $A$  est bien transformé en tas. De plus, les seules modifications apportées à  $A$  le sont par échange d'éléments entre des nœuds. L'ensemble des éléments à la fin est donc bien le même qu'au début.

```
void formerTas(arbrebqp_t x){
    int i;
    if ( taillearbrebqp(x) > 1 ){
        for ( i = parent(dernier(x)); i >= racine(x); i--){
            maintienBas(x, i);
        }
    }
}
```

**Complexité.** On montre facilement que cet algorithme est en  $O(N \log N)$  en pire cas. En fait, on peut serrer plus la borne. On va montrer que le nombre d'échanges requis par ce nouvel algorithme pour former un tas de  $N$  éléments est linéaire en  $N$  en pire cas.

Pour simplifier les calculs, on se place dans le cas où l'arbre est parfait : dans ce cas  $N = 2^h - 1$  où  $h$  est la hauteur de l'arbre. Il est ensuite facile de généraliser comme on l'avait fait pour la recherche dichotomique (voir section 2.1.2 page 19).

En pire cas, le nombre d'échanges requis par l'application de l'algorithme de maintien vers le bas à un nœud  $m$  est  $h' - 1$  où  $h'$  est la hauteur du sous-arbre de racine  $m$ .

Dans l'arbre, il y a  $2^{h-1}$  feuilles toutes de hauteur 1. Pour chaque  $i$  tel que  $1 < i \leq h$ , il y a  $2^{h-i}$  nœuds internes de hauteur  $i$ . Si  $m$  est un nœud de hauteur  $i$ , la hauteur d'un sous-arbre de racine  $m$  est  $i$ .

Le nombre total d'échanges est donc majoré en pire cas par la somme :

$$E(h) = \sum_{i=1}^h i 2^{h-i} = 2^{h-1} \cdot \underbrace{\sum_{i=1}^h i \left(\frac{1}{2}\right)^{i+1}}_{S_h}$$

On cherche une majoration de  $S_h$ .

**Première méthode.** On pose :

$$f_h(z) = \sum_{i=1}^h z^i$$

On a alors  $S_h = f'_h(1/2)$ , il reste à évaluer  $f'_h(1/2)$ .

On a :

$$f_h(z) = \frac{z^{h+1} - 1}{z - 1}$$

D'où :

$$f'_h(z) = \frac{(h+1)z^h(z-1) - (z^{h+1} - 1)}{(z-1)^2}$$

$$f'_h\left(\frac{1}{2}\right) = -\frac{(h+1)(1/2)^{h+1} + ((1/2)^{h+1} - 1)}{(1/2)^2}$$

Comme  $S_{h+k} \geq S_h$  pour  $k \geq 0$  on obtient :

$$f'_h\left(\frac{1}{2}\right) \leq \lim_{h \rightarrow \infty} \frac{-(h+1)(1/2)^{h+1} - (1/2)^{h+1} + 1}{(1/2)^2} = 4$$

D'où  $S_h \leq 4$ . Ce qui donne  $E(h) \leq 4 \times 2^{h-1} = 4 \times (N+1)/2$  de quoi l'on déduit facilement que  $E(h) = O(N)$ , ce qui conclut.

**Autre méthode.** On sait que la somme  $s = 1 + \frac{1}{2} + \frac{1}{4} + \dots = \sum_{i=1}^{+\infty} \frac{1}{2^i}$  vaut 2. On a  $S_h \leq S_{+\infty}$  et on remarque que

$$S_{+\infty} = s + \frac{1}{2}s + \frac{1}{4}s + \dots = s \times \sum_{i=1}^{+\infty} \frac{1}{2^i} = s^2 = 4.$$

Donc  $S_h$  est majorée par 4 et on conclut de la même manière que précédemment.

#### 4.2.4 Le tri par tas

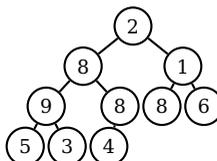
En anglais : *heap sort, Williams, 1961*

La structure de tas, permet d'écrire un algorithme de tri en place optimal en temps, c'est à dire en  $\Theta(N \log N)$ . Le principe est de prendre un tableau en entrée, d'y former un tas ( $O(N)$ ) et de retirer un à un les maximums successifs en les rangeant à la fin du tableau ( $N$  fois  $O(\log N)$ ). En C, cela donne le code suivant :

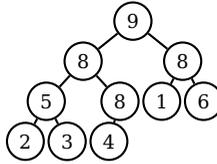
```
void triTas(tableau_t *t){
    /* 1) On tranforme le tableau en un arbre bqp */
    arbrebqp_t x;
    x = newArbreBQP(); /* allocation mémoire */
    x->tab = t->tab; /* On fait juste référence au tableau sans le
        recopier. On travaille donc "en place".*/
    x->mem = t->taille;
    x->taille = t->taille;
    /* 2) On forme le tas */
    formerTas(x); /* option affichage */ AFFICHAGE_TAS_X;
    /* 3) On extrait les maximums successifs en les plaçant en fin de
        tableau */
    while (x->taille > 1) {
        /* a) On place le maximum à la fin du tas */
        echangetas(x, racine(x), dernier(x));
        /* b) On diminue de un le nombre d'éléments du tas, en préservant le
            reste du tableau */
        x->taille--;
        /* c) On reforme le tas */
        maintienBas(x, racine(x));
    }
    /* Fin) le tableau t->tab est trié, t->taille a la bonne valeur, on
        peut libérer l'espace mémoire pris par x. */
    free(x);
}
```

Prenons par exemple le tableau : [2 8 1 9 8 8 6 5 3 4]. L'exécution du tri par tas sur cet exemple fonctionne comme suit.

1. On regarde le tableau comme un arbre quasi-parfait :

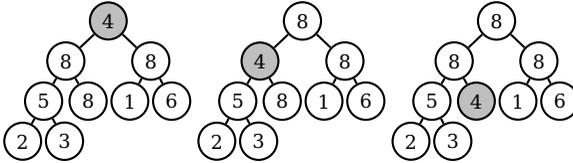






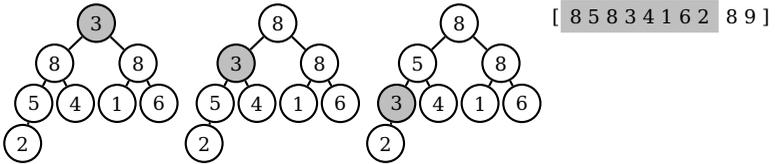
Le tableau est donc devenu : [ 9 8 8 5 8 1 6 2 3 4 ]. Il contient exactement les mêmes éléments que le tableau de départ, organisés en tas.

3. On extrait l'élément maximum (à la racine) en l'échangeant avec le dernier élément du tableau (la dernière feuille) ; on sort ce dernier élément de l'arbre ; et on reforme le tas.

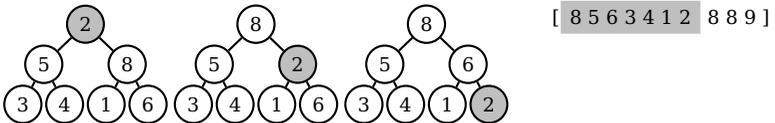


Le tableau est maintenant : [ 8 8 8 5 4 1 6 2 3 9 ] (en grisé, les éléments encore dans l'arbre).

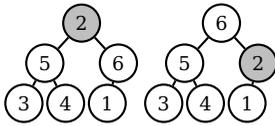
On recommence avec le nouveau maximum :



Encore...

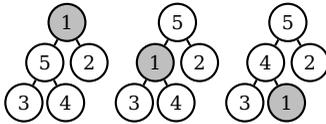


... et encore...



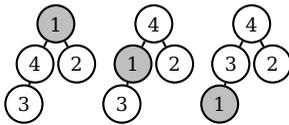
[ 6 5 2 3 4 1 8 8 8 9 ]

... et encore...



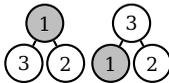
[ 5 4 2 3 1 6 8 8 8 9 ]

... et encore...



[ 4 3 2 1 5 6 8 8 8 9 ]

... et encore...



[ 3 1 2 4 5 6 8 8 8 9 ]

... et encore...



[ 2 1 3 4 5 6 8 8 8 9 ]

... jusqu'au dernier élément.



[ 1 2 3 4 5 6 8 8 8 9 ]

4. On obtient le tableau trié [1 2 3 4 5 6 8 8 8 9].

Le tri par tas est un algorithme de tri efficace : en place et en  $O(N \log N)$ , mais ses performances se dégradent lorsque le tableau à trier ne tient pas en entier dans la mémoire de travail. Dans ce cas, quel que soit le tri, il faut que le processus charge les portions (pages mémoires) de tableau en mémoire au moment où il a besoin d'y accéder (lecture ou écriture). Mais contrairement

à d'autres tris qui travaillent plus localement, le tri par tas fait des accès successifs au tableau dans des zones très éloignées les unes des autres ce qui provoque de nombreux chargements (fautes de pages).

### 4.3 Arbres binaires de recherche

**Définition 4.5.** Un *arbre binaire de recherche* est un arbre binaire, dont les nœuds contiennent des éléments munis d'une clé dans un ensemble totalement ordonné, et qui vérifie la propriété suivante :

Soit  $x$  un nœud d'un arbre binaire de recherche. Si  $y$  est un nœud du sous-arbre gauche de  $x$ , alors  $\text{clé}(y) \leq \text{clé}(x)$ . Si  $y$  est un nœud du sous-arbre droit de  $x$ , alors  $\text{clé}(y) \geq \text{clé}(x)$ .

**Propriété 4.6:** *Un sous-arbre quelconque d'un arbre binaire de recherche est encore un arbre binaire de recherche.*

Par sous-arbre quelconque on entend un sous-arbre où on ne sélectionne pas nécessairement tous les descendants d'un nœud mais seulement certains.

---

#### Fonction Rechercher( $n$ )

---

**Entrées :** Le nœud racine  $r$  d'un arbre binaire de recherche et la clé d'un élément  $c$

**Sorties :** Le nœud de l'arbre contenant l'élément de clé  $c$  s'il en existe un, ou le nœud vide  $N$  sinon.

**si** EstVide( $r$ ) **alors**

  | **retourner**  $N$  ;

**sinon**

  | **si**  $c = \text{Clé}(r)$  **alors**

    | **retourner**  $r$  ;

  | **sinon**

    | **si**  $c < \text{Clé}(r)$  **alors**

      | **retourner** Rechercher(Gauche( $r$ ),  $c$ ) ;

    | **sinon**

      | **retourner** Rechercher(Droite( $r$ ),  $c$ ) ;

---

Les opérations élémentaires sont la recherche d'un élément à partir de sa clé (voir le pseudo-code page 64), l'insertion et la suppression d'un élément, ainsi que la recherche de l'élément maximum et la recherche de l'élément minimum. Ces opérations prennent un temps en  $O(h)$  où  $h$  est la hauteur de l'arbre. On compte également dans les opérations élémentaires le test à vide qui s'exécute en temps constant.

La hauteur  $h$  peut varier entre  $\log n$  et  $n$  où  $n$  est le nombre d'éléments de l'arbre. La coloration rouge noir vue plus loin est une manière d'assurer le fait que  $h$  soit en  $\Theta(\log n)$  et ainsi, que toutes les opérations élémentaires soient en  $O(\log n)$ .

Le parcours infixe est un parcours séquentiel des nœuds de l'arbre, qui pour chaque nœud commence par parcourir son sous-arbre gauche, puis traite le nœud, puis parcourt le sous-arbre droit.

**Propriété 4.7:** *Le parcours infixe d'un arbre binaire de recherche donne la liste des éléments qu'il contient par ordre croissant (et ceci en un temps  $\Theta(n)$ ).*

Ainsi un arbre binaire de recherche peut être considéré comme une manière de garder rangée une liste d'éléments (penser à la recherche dichotomique dans un tableau).

On considère également deux opérations supplémentaires en  $O(h)$ , successeur et prédécesseur, qui, étant donné un nœud de l'arbre donnent le nœud contenant l'élément qui lui succède, respectivement qui le précède, dans l'ordre du parcours infixe, c'est à dire l'ordre naturel des éléments. Ces deux opérations sont en  $O(h)$ .

Le code C d'une implémentation des arbres binaires de recherche sera vu en cours d'amphi. Dans cette implémentation la clé d'un élément est l'élément lui-même (simplification).

On peut accepter ou non les répétitions de clés dans un arbre binaire de recherche. Ici on considère que les arbres sont sans répétitions : l'insertion d'un élément déjà présent dans l'arbre ne modifie pas l'arbre.

**Convention :** Pour la suite, on considère que les nœuds d'un arbre binaire ont soit deux fils, soit aucun fils. Les nœuds sans fils sont les feuilles de l'arbre, notées N ou NULL, et ne contiennent pas d'élément. On ne compte pas ces feuilles dans la hauteur de l'arbre.

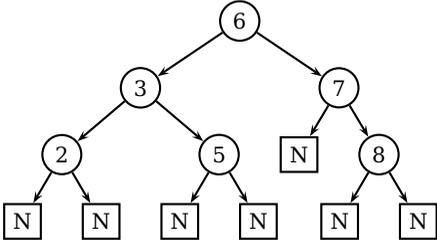


Figure 4.4 - Un exemple d'arbre binaire de recherche

Les algorithmes pour Insérer, Supprimer, Parcours-infixe, Maximum, Minimum, ainsi que Successeur et Prédécesseur seront détaillés au moment de l'implantation. Nous donnons juste ici quelques indications.

- Pour Maximum, qui renvoie l'élément maximum de l'arbre il suffit, partant de la racine, de descendre toujours à droite jusqu'à ce que ça ne soit plus possible. Le maximum est le dernier élément trouvé.
- Pour la fonction Successeur y a deux cas de figure :
  - soit le sous-arbre droit du nœud n'est pas vide, et dans ce cas il suffit de rechercher l'élément de clé minimale (le plus à gauche) dans ce sous-arbre droit.
  - soit le sous-arbre droit du nœud est vide, et dans ce cas il faut rechercher le premier ancêtre dont le fils gauche est un ancêtre du nœud.
- Insérer. L'insertion ajoute toujours l'élément comme une nouvelle feuille de l'arbre dont la place est trouvée de la même manière qu'on effectue la recherche d'une clé.
- Pour Supprimer, il y a plusieurs cas. Si le nœud à supprimer n'a pas de fils, on l'ôte. S'il n'a qu'un fils, on ôte le nœud et on rebranche son fils à la place. S'il a deux fils, on l'échange avec son successeur (ou son prédécesseur, ça marche aussi), dont on sait qu'il n'a pas de fils (voir Successeur), et on l'ôte.

Pour une forme d'arbre binaire donnée, il y a une seule manière de ranger dans ses nœuds des éléments deux à deux comparables de manière à en faire un arbre binaire de recherche. La répartition des éléments dans les nœuds mets en correspondance le parcours infixé des nœuds et la liste triée des éléments.

Par contre pour une liste donnée d'éléments, n'importe quelle forme d'arbre ayant le même nombre de nœuds qu'il y a d'éléments convient pour les accueillir.

Une suite d'insertions aléatoires de  $n$  éléments différents à partir d'un arbre vide produit un arbre binaire recherche dont la hauteur est en général plus proche de  $\log n$  que de  $n$ . Mais il peut être utile de s'assurer que la hauteur reste proche de  $\log n$ , comme avec la coloration rouge noir vue plus loin. Pour agir sur la forme de l'arbre, on utilise des rotations qui sont des opérations préservant la propriété d'être un arbre binaire de recherche.

### 4.3.1 Rotations

Les rotations des arbres binaires sont données dans la figure 4.5. Elles se lisent comme ceci. Une rotation à droite de centre  $x$  consiste en : prendre l'élément  $a$  de  $x$ , le sous-arbre droite  $E$  de  $x$ , la racine  $y$  du sous-arbre gauche de  $x$ , l'élément  $b$  contenu dans  $y$ , et  $C$  et  $D$  les deux sous-arbres gauche et

droite de  $y$  ; remplacer  $a$  par  $b$  dans  $x$ , remplacer le sous-arbre gauche de  $x$  par  $C$ , et le sous-arbre droite de  $x$  par un arbre de racine contenant  $a$  (on utilise  $y$  pour des raisons d'implantation) et dont les sous-arbres gauche et droite sont respectivement  $D$  et  $E$ . La rotation à gauche de centre  $x$  est l'opération réciproque.

Avec cette manière d'écrire les rotations la référence de  $x$  à son parent n'a pas besoin d'être mise à jour. Il est assez standard de trouver une version qui échange la place de  $x$  et de  $y$  plutôt que d'échanger leurs éléments mais nous ne l'utilisons pas ici.

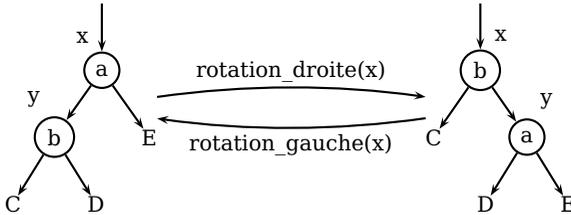


Figure 4.5 – Rotations gauche et droite. Dans cette version les éléments  $a$  et  $b$  sont échangés entre les nœuds  $x$  et  $y$  mais  $x$  garde sa place dans l'arbre.

Il faut montrer la préservation de la propriété d'arbre de recherche. On se contente de montrer que si l'arbre à gauche de la figure est un arbre binaire de recherche alors l'arbre à droite en est un, et réciproquement. En effet, pour l'arbre qui contient l'un de ces deux arbres comme sous-arbre, le fait de remplacer l'un par l'autre ne fait aucune différence : les deux sous-arbres ont mêmes ensembles d'éléments. Pour chacun des deux arbres de la figure on montre qu'être un arbre de recherche, sous l'hypothèse que  $C$ ,  $D$  et  $E$  en sont, est équivalent à la propriété :  $\forall c \in C, \forall d \in D, \forall e \in E, \text{Clé}(c) \leq \text{Clé}(b) \leq \text{Clé}(d) \leq \text{Clé}(a) \leq \text{Clé}(e)$ , ce qui conclue.

### 4.3.2 Arbres rouge noir

Nous donnons uniquement la définition des arbres rouge noir, le reste est esquissé. Une étude plus précise sera faite en cours et en TD. On montrera notamment en exercice que la hauteur d'un arbre rouge noir est logarithmique en son nombre de nœuds.

**Définition 4.8.** Un *arbre rouge noir* est un arbre binaire de recherche comportant un champ supplémentaire par nœud : sa *couleur*, qui peut valoir soit ROUGE, soit NOIR. En outre, un arbre rouge noir satisfait les propriétés suivantes :

1. Chaque nœud est soit rouge, soit noir.

2. Chaque feuille est noire.
3. Si un nœud est rouge, alors ses deux fils sont noirs.
4. Pour chaque nœud de l'arbre, tous les chemins descendants vers des feuilles contiennent le même nombre de nœuds noirs.
5. La racine est noire.

Un exemple est donné figure 4.6.

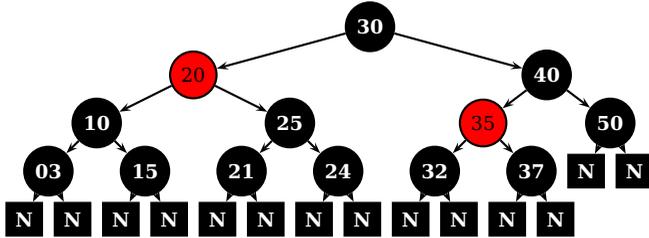


Figure 4.6 – Un exemple d'arbre rouge noir

Soit  $x$  un nœud d'un arbre rouge noir. On appelle *hauteur noire* de  $x$ , notée  $H_n(x)$ , le nombre de nœuds noirs présents dans un chemin descendant de  $x$  (sans l'inclure) vers une feuille de l'arbre. D'après la propriété 4, cette notion est bien définie.

Les opérations élémentaires sont les mêmes que celles des arbres binaires de recherche, sauf que l'insertion et la suppression supposent des opérations de maintien de la bonne coloration rouge noir de l'arbre. Ces opérations de maintien font appel à des rotations et sont assez compliquées à détailler, mais elles restent en  $O(h)$  où  $h$  est la hauteur.

## **Troisième partie**

# **Éléments de calculabilité**

## Chapitre 5

# Éléments de calculabilité

Jusqu'à maintenant nous nous sommes intéressés à la manière de résoudre des problèmes par un calcul (tri, organisation de données, recherche etc.), en particulier à l'aune du temps d'exécution. Mais tous les problèmes ne peuvent pas être résolus par un calcul. En particulier, il existe des problèmes qui s'énoncent sur des données finies et parfaitement informatisables dont on sait (on l'a démontré) qu'ils ne sont pas solubles par un calcul. Ce type de résultats est du domaine de la calculabilité. Ils supposent de se donner un modèle de calcul simple et précis (souvent les machines de Turing), mais l'exposé de ces modèles nécessiterait un cours complet. Nous faisons donc un compromis et nous gardons comme modèle de calcul un langage de programmation du type langage C et des programmes s'exécutants sur nos ordinateurs.

Ce chapitre expose quelques uns des résultats les plus spectaculaires de calculabilité. Les modèles de calcul usuels en calculabilité datent pour la plupart d'avant l'informatique des ordinateurs. Le résultat d'indécidabilité de l'arrêt est due à Alan Turing, et date de 1936 (il s'agit d'une formulation plus calculatoire du célèbre théorème d'incomplétude de Kurt Gödel). Le théorème de Rice est quant à lui daté de 1953. Pour l'un et l'autre de ces résultats, il a fallu beaucoup de temps pour que leurs formulations modernes s'imposent et que leurs implications soient comprises dans la communauté informatique.

Nous commençons par quelques rappels sur les notions de dénombrement et de fonction (au sens mathématique).

**Définition 5.1.** Une *fonction* au sens mathématique est la donnée de deux ensembles, un ensemble d'entrées  $E$ , un ensemble de sorties  $S$ , et d'une relation  $f \subset E \times S$ , c'est à dire d'un ensemble de couples  $(e, s)$ , telle que la relation  $f$  est déterministe, c'est à dire que si  $(e, s)$  et  $(e, s')$  sont tout deux dans la relation alors  $s = s'$ .

Lorsque  $(e, s)$  est dans la relation  $f$ , on écrit  $f : e \mapsto s$ , ou encore  $f(e) = s$ . Lorsque pour tout  $e \in E$ ,  $f(e)$  est défini (c'est à dire qu'il existe un  $s \in S$  tel que  $f(e) = s$ ) la fonction est *totale*, autrement la fonction est *partielle*. Il suffit d'ajouter un élément distingué  $\perp$  à l'ensemble  $S$  pour représenter les fonctions partielles de  $E \rightarrow S$  par les fonctions totales de  $E \rightarrow S \cup \{\perp\}$  (on envoie les  $e$  pour lesquels  $f$  n'est pas définie sur  $\perp$ ).

**Définition 5.2.** On dit d'un ensemble  $E$  qu'il est *dénombrable* lorsque il existe une bijection entre  $E$  et  $\mathbb{N}$ .

Toute partie infinie de  $\mathbb{N}$  est dénombrable. On dit d'un ensemble qu'il est au plus dénombrable lorsqu'il existe une bijection entre  $E$  et une partie de  $\mathbb{N}$ .

**Proposition 5.3** (Cantor). *L'ensemble  $\mathcal{P}(\mathbb{N})$  des parties de  $\mathbb{N}$  est non-dénombrable.*

La preuve introduit le célèbre argument de la diagonale, de Cantor.

*Démonstration.* Par contradiction. Si  $\mathcal{P}(\mathbb{N})$  est dénombrable, il existe une énumération (bijective) :  $e : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N})$ . On considère alors la partie de  $\mathbb{N}$  suivante :

$$D = \{k \in \mathbb{N} \mid k \notin e(k)\}$$

Cette partie  $D$  doit avoir un numéro  $n$  dans l'énumération  $e(n) = D$ . Est-ce que  $n \in D$ ? Si oui,  $n \notin e(n) = D$  ce qui n'est pas possible. Si non, alors  $n$  est tel que  $n \notin e(n)$  donc  $n \in D$  ce n'est pas possible non plus. Il y a contradiction dans les deux cas, c'est donc que l'énumération bijective n'existe pas c'est à dire que  $\mathcal{P}(\mathbb{N})$  n'est pas dénombrable.  $\square$

**Proposition 5.4.** *L'ensemble  $\mathbb{N}^{\mathbb{N}}$  des fonctions mathématiques de  $\mathbb{N} \rightarrow \mathbb{N}$  est non-dénombrable. Ceci est également vrai pour l'ensemble  $(\mathbb{N} \cup \{\perp\})^{\mathbb{N}}$  des fonctions partielles de  $\mathbb{N} \rightarrow \mathbb{N}$ .*

*Démonstration.* Il suffit de considérer le sous ensemble

$$X = \{\chi_A \mid A \subseteq \mathbb{N}\}$$

des fonctions  $\chi_A : n \mapsto \begin{cases} 1 & \text{si } n \in A \\ 0 & \text{sinon} \end{cases}$  qui testent l'appartenance à une partie

$A$  de  $\mathbb{N}$ . Cet ensemble a autant d'éléments que  $\mathcal{P}(\mathbb{N})$ , il est donc non dénombrable, et il s'en suit que  $\mathbb{N}^{\mathbb{N}}$  ne peut pas être dénombrable.  $\square$

On peut toujours considérer qu'un programme calcule une fonction au sens mathématique : il associe de manière déterministe une sortie d'un certain ensemble  $S$  à une entrée d'un certain ensemble  $E$ . Cette fonction est en général partielle : le programme boucle sur certaines entrées, et ne renvoie pas de résultat, la fonction n'est donc pas définie sur ces entrées. Par opposition à partielle, une fonction qui à chaque entrée associe une sortie est dite totale. On peut totaliser la fonction (la rendre totale) en ajoutant un symbole  $\perp$  pour dénoter que le programme boucle sur l'entrée  $e$ .

**Définition 5.5.** Une *fonction calculable* est une fonction (au sens mathématique) telle qu'il existe au moins un programme la calculant.

Pour une même fonction calculable il est possible d'écrire une infinité de programmes qui la calcule (il suffit de partir de l'un de ces programmes et d'y ajouter des blocs d'instructions arbitraires n'ayant pas d'incidence sur le résultat du calcul).

On oppose parfois l'intension, (ou la compréhension) des programmes à l'extension des fonctions. Intuitivement, une extension est une liste exhaustive souvent infinie : la liste des couples  $(e, s)$  mis en relation par une fonction, la liste des entiers pairs, tandis que l'intension est une représentation finie d'un ensemble potentiellement infinie d'objets, par exemple un programme calculant  $f(e)$  pour tout  $e$ , ou l'écriture  $\{n \in \mathbb{N} \mid 2 \text{ divise } n\}$ .

**Proposition 5.6.** *L'ensemble des mots finis d'entiers est dénombrable par une fonction calculable de réciproque calculable. L'ensemble  $\mathcal{P}_{\text{fin.}}(\mathbb{N})$  des parties finies de  $\mathbb{N}$  est dénombrable.*

*Démonstration.* Il est plus facile de démontrer que l'ensemble des mots finis non-vides d'entiers est dénombrable (voir plus bas). On ajoute le mot vide en utilisant un décalage  $n \mapsto n + 1$  dans l'énumération précédente. Pour ce qui est de la deuxième partie de la proposition, on peut alors rapidement conclure par le théorème de Cantor-Bernstein mais c'est hors du programme de ce cours. Ce théorème très pratique établit que s'il existe une injection de  $f : A \hookrightarrow B$  et une injection  $g : B \hookrightarrow A$  alors il existe une bijection entre  $A$  et  $B$ .

Montrons que l'ensemble des mots finis non-vides d'entiers est dénombrable. Rappelons que  $\mathbb{N} \times \mathbb{N}$  est dénombrable c'est à dire qu'il existe une bijection calculable dans les deux sens  $f : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ . Pour les septiques, prendre l'inverse de la fonction  $(p, q) \mapsto 2^p(2q + 1)$  (notez que cet inverse est calculable). À un entier  $k$  on peut alors associer une paire  $f(k) = (n, c)$  où  $n + 1$  sera la longueur du mot recherché. Il suffit alors d'appeler  $f$ ,  $n$  fois sur le second terme de la paire trouvée à chaque étape, pour construire un mot d'entiers : si  $n = 3$  on fait par exemple  $f(c) = (a_1, c_1)$ ,  $f(c_1) = (a_2, c_2)$ ,  $f(c_2) = (a_3, c_3)$ ,  $a_4 = c_3$ , ce qui donne le mot  $a_1a_2a_3a_4$ .  $\square$

Un *problème de décision* sur un ensemble  $E$  est un énoncé  $D(e)$  qui pour chaque élément  $e$  de  $E$  est soit vrai soit faux. C'est donc une fonction  $D : E \rightarrow \{\text{vrai, faux}\}$ . Un problème de décision est trivial lorsque il est vrai pour tous les éléments de  $E$  (l'ensemble  $\{e \in E \mid D(e)\}$  est égal à  $E$ ) ou bien lorsqu'il est faux pour tous les éléments de  $E$  (l'ensemble  $\{e \in E \mid D(e)\}$  est vide).

D'un point de vue algorithmique, un problème de décision est dit *décidable* lorsqu'il existe un programme qui le calcule ( $D : E \rightarrow \{\text{vrai, faux}\}$  est une fonction calculable). Autrement il est dit *indécidable*.

Attention, pour qu'un problème soit décidable il faut que le programme qui le calcule termine pour chaque entrée  $e \in E$  en donnant la réponse exacte. Un problème indécidable  $D$  est dit *semi-décidable* lorsqu'il existe un programme qui termine et répond vrai sur chaque entrée  $e \in E$  pour laquelle  $D(e)$  est vrai, et lorsqu'il termine sur les autres entrées, répond faux. Remarquez qu'il existe nécessairement une entrée  $e$ , telle que  $D(e)$  est faux, pour laquelle le programme ne termine pas (autrement le problème serait décidable). D'un point de vue calculatoire, la semi-décidabilité n'a que peu d'intérêt.

**Proposition 5.7.** *Pour un langage donné, l'ensemble des programmes qui calculent une fonction partielle de  $\mathbb{N} \rightarrow \mathbb{N}$  est dénombrable, et donc l'ensemble des fonctions calculables est dénombrable.*

Les deux dernières propositions montrent qu'il y a beaucoup plus de fonctions non-calculables que de fonctions calculables.

**Lemme 5.8.** *Pour un langage donné, on peut énumérer l'ensemble des programmes, et cette énumération peut être choisie calculable et de réciproque calculable.*

**Théorème 5.9** (Indécidabilité de l'arrêt). *Il n'existe pas de programme  $H$  prenant en argument n'importe quel programme  $p$  et des arguments pour  $p$ , et qui déterminerait avec certitude si l'exécution de  $p$  ses arguments s'arrêtera ou bouclera indéfiniment.*

*Démonstration.* On se donne une énumération des programmes de  $\mathbb{N} \rightarrow \mathbb{N}$ .

On suppose que l'on peut programmer une fonction  $h(p, k)$  qui teste l'arrêt du  $p$ -ième programme sur l'argument entier  $k$  : si le  $p$ -ième programme sur l'entrée  $k$  termine alors  $h(p, k) = 1$  sinon  $h(p, k) = 0$ . Par l'argument de la diagonale, on arrive à une contradiction. Celle-ci est simplement construite en écrivant le programme  $g$  qui prend en entrée un entier  $p$ , calcule  $h(p, p)$  et si le résultat est 1 boucle éternellement, sinon renvoie 0. Par construction,  $g$  est un programme de  $\mathbb{N} \rightarrow \mathbb{N}$ . Donc  $g$  a un numéro, disons  $n$  dans notre énumération. Est-ce que  $g(n)$  termine ? Si oui c'est que  $h(n, n) = 0$ , mais

alors le  $n$ -ième programme,  $g$ , boucle sur l'entrée  $n$ , ce qui est impossible. Si non c'est que  $h(n, n) = 1$  mais alors le  $n$ -ième programme  $g$ , termine sur l'entrée  $n$ , ce qui est également impossible. C'est impossible dans les deux cas. Contradiction. Donc  $h$  n'existe pas.

Jusque là nous avons simplement utilisé le fait que nous pouvions associer un numéro unique à chaque programme, par contre rien n'exige que la numérotation soit surjective ou calculable ou de réciproque calculable.

Pour conclure à l'impossibilité de l'existence de  $H$ , il nous faut montrer que l'existence de  $H$  implique celle de  $h$ .

Si le programme  $H$  existe il fonctionne sur les programmes de  $\mathbb{N} \rightarrow \mathbb{N}$ . Or d'après le lemme 5.8, il existe une énumération de ces programmes dont la réciproque est calculable. On a donc une fonction  $e$  qui à chaque entier associe un programme  $e(p)$  de type  $\mathbb{N} \rightarrow \mathbb{N}$ . On peut donc écrire le programme  $h(p, k)$  comme étant le programme qui renvoie la valeur de  $H(e(p), k)$ . Mais puisque  $h$  n'existe pas c'est que  $H$  non plus.  $\square$

**Proposition 5.10** (Indécidabilité de l'arrêt sur un argument (temps)). *Parmi les algorithmes calculant une fonction partielle de  $\mathbb{N} \rightarrow \mathbb{N}$  savoir lesquels terminent sur l'argument zéro est indécidable.*

*Remarque 5.11.* Si les algorithmes sont exécutés sur une machine à mémoire finie, alors le nombre d'états de n'importe quel programme est fini. Dans ce cas il est théoriquement possible de décider de l'arrêt de n'importe quel programme s'exécutant sur cette machine. En effet, sur une machine à  $N$  états si un programme change plus de  $N$  fois d'état avant l'arrêt du programme, alors le programme ne s'arrêtera jamais (il passe par une suite d'états cycliques). Nos ordinateurs sont à états finis mais attention  $N$  est en général tellement grand que cet argument de finitude est inapplicable en pratique. Pensez aux  $2^{8\,589\,934\,592}$  états d'un ordinateur ayant un Gibi octet de mémoire (sans parler des mémoires de masse)!

Un propriété des algorithmes ou des programmes qui s'énonce sur les fonctions calculées est dite *extensionnelle*. La notion est assez souple car on peut faire varier ce qui est considéré comme une entrée et ce qui est considéré comme une sortie d'un programme. Par exemple, le fait d'afficher «bonjour» est une propriété extensionnelle des programmes, pour la notion de sortie qui inclue les sorties écran.

**Théorème 5.12** (Théorème de Rice). *Toute propriété extensionnelle sur les programmes est soit triviale soit indécidable.*

*Démonstration.* On se ramène à l'indécidabilité de l'arrêt.  $\square$

**Proposition 5.13** (Indécidabilité de l'égalité extensionnelle (espace)). *Si on restreint notre champs d'étude aux algorithmes de type  $\mathbb{N} \rightarrow \mathbb{N}$  qui terminent toujours (sur chaque argument), savoir si deux de ces algorithmes calculent la même fonction mathématique est un problème indécidable (il n'existe pas de programme scrutateur qui donne la bonne réponse à tous les coups).*

*Démonstration.* On simule facilement l'exécution d'un programme  $p$  sur une entrée particulière, qui ne termine pas forcément, à l'aide d'un algorithme  $\mathbb{N} \rightarrow \mathbb{N}$  qui terminent tout le temps. Soit  $f_p$  l'algorithme qui prend en entrée  $n$  et lance l'exécution de  $p(0)$  pendant  $n$  cycles d'horloge. Si  $p(0)$  a terminé en  $n$  cycles ou moins il renvoie 1, et sinon il renvoie 0. On forme ainsi un ensemble de programmes  $f_p$  qui terminent toujours. Supposons que nous sachions toujours décider le problème de la proposition. Alors nous serions capables de savoir si la fonction calculée par un  $f_p$  prend la valeur 1 au bout d'un certain temps, ce qui reviendrait à pouvoir décider si le programme  $p$  termine sur l'entrée 0. Contradiction.  $\square$

*Remarque 5.14.* Le problème de l'arrêt est semi-décidable.

*Remarque 5.15.* Si  $E$  est un ensemble fini de données disons une partie finie de  $\mathbb{N}$ , et que l'on considère uniquement les algorithmes  $E \rightarrow \mathbb{N}$  qui terminent toujours, alors il est facile de tester si deux de ces algorithmes calculent la même fonction de  $E \rightarrow \mathbb{N}$  (en calculant effectivement sur toutes les valeurs de  $E$ ).

Quelques conséquences du théorème de Rice :

- Il est impossible de déterminer automatiquement, pour tous les programmes, si un programme va générer une erreur et planter.
- Savoir détecter tous les programmes qui contiennent un virus est un problème indécidable (l'anti-virus parfait n'existe pas).
- Si on donne par défaut des droits en écriture sur un compte utilisateur à un langage de script de pages web, alors il est impossible qu'un programme (le navigateur, un pare-feu etc.) puisse toujours détecter si un script risque d'utiliser ce droit de manière dangereuse ou malveillante.
- Il est impossible d'écrire un ramasse-miette qui évite à tous les coups les fuites de mémoire.
- L'analyse syntaxique des programmes Perl 5 est indécidable (plus précisément, la question indécidable est de savoir si un programme Perl 5 correspond à un arbre syntaxique donné).

On peut étendre facilement le théorème de Rice et la proposition 5.13 de manière à prendre en compte le temps de calcul. Il est par exemple indécidable de déterminer si deux programmes qui terminent et ont asymptotiquement le même temps d'exécution en pire cas calculent la même fonction

(Asperti 2006). Par exemple, parmi les programmes qui s'exécutent en temps  $\Theta(N \log N)$ , il est impossible de savoir à coup sûr lesquels font un tri sur leur entrée.

En terme de complexité en temps ou en espace des programmes, on définit des classes de programmes, qui ont l'avantage d'être closes par composition<sup>1</sup>. Par exemple la classe  $\mathbf{P}$  des programmes qui calculent en temps polynomial est l'ensemble des programmes  $f$  pour lesquels il existe un polynôme réel  $Q(X)$  tel que sur une donnée  $d$ ,  $f$  calcule en pire cas en temps  $O(Q(N))$  où  $N$  est la taille de la donnée  $d$ . Si deux programmes  $f$  et  $g$  sont dans  $\mathbf{P}$ , alors le programme qui calcule  $g(f(d))$  pour toute donnée  $d$  est également dans  $\mathbf{P}$ .

**Théorème 5.16** (Folklore, Terui). *Savoir si un programme qui termine toujours calcule en temps polynomial est indécidable.*

---

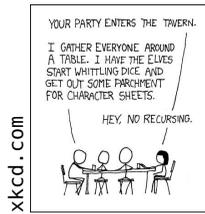
1. Ces classes sont très connues. D'ailleurs vous avez peut être déjà entendu parler de la grande question à un million de dollars  $\mathbf{P} \neq \mathbf{NP}$  ?

## **Quatrième partie**

### **Exercices**

# **Chapitre 6**

# **Exercices**



## 6.1 Exercices du premier chapitre

### 6.1.1 Récursivité

#### Exercice 1 (Récursivité).

1. Que calcule la fonction suivante (donnée en pseudo-code et en C)?

<b>Fonction</b> Toto( $n$ )	/* Fonction toto en C */
<b>si</b> $n = 0$ <b>alors</b>   <b>retourner</b> 1; <b>sinon</b>   <b>retourner</b> $n \times \text{Toto}(n - 1)$ ;	unsigned toto(unsigned n){ if (n == 0) return 1; return n * toto(n - 1); }

2. La suite de Fibonacci est définie récursivement par la relation  $u_n = u_{n-1} + u_{n-2}$ . Cette définition doit bien entendu être complétée par une condition d'arrêt, par exemple :  $u_1 = u_2 = 1$ . Écrire une fonction qui calcule et renvoie le  $n$ -ième terme de la suite de Fibonacci ( $n \in \mathbb{N}^*$  donné en argument de la fonction).
3. Écrire une fonction récursive qui calcule le pgcd de deux nombres entiers positifs.
4. Que calcule la fonction suivante ?

<b>Fonction</b> Tata( $n$ )	/* Fonction tata en C */
<b>si</b> $n \leq 1$ <b>alors</b>   <b>retourner</b> 0; <b>sinon</b>   <b>retourner</b> $1 + \text{Tata}(\lfloor \frac{n}{2} \rfloor)$ ;	unsigned tata(unsigned n){ if (n <= 1) return 0; return 1 + tata(n / 2); }

5. Il n'est parfois pas suffisant d'avoir un bon cas de base, voici un exemple. En C, que vaut `Morris(1, 0)` ?

```
int Morris(int a, int b) {
    if (a == 0) return 1;
    return Morris(a - 1, Morris(a, b));
}
```

**Exercice 2** (La fonction 91 de McCarthy).

Les fonctions récursives mêmes simples donnent parfois des résultats difficiles à prévoir. Pour s'en convaincre voici un exemple. Pour  $n > 100$  la fonction 91 de McCarthy vaut  $n - 10$ . Mais pour  $n \leq 100$ ? Tester sur un exemple... pas trop mal choisi, puis prouver le résultat en toute généralité.

---

**Fonction** Tata( $n$ )

---

```
si  $x > 100$  alors
  | retourner  $x - 10$ ;
sinon
  | retourner McCarthy(McCarthy( $x + 11$ ));
```

---

```
int McCarthy(int x)
{
    if (x > 100) return(x - 10);

    return McCarthy(McCarthy(x + 11));
}
```

**Exercice 3** (Tours de Hanoï).

On se donne trois piquets,  $p_1, p_2, p_3$  et  $n$  disques percés de rayons différents enfilés sur les piquets. On s'autorise une seule opération : Déplacer-disque( $p_i, p_j$ ) qui déplace le disque du dessus du piquet  $p_i$  vers le dessus du piquet  $p_j$ . On s'interdit de poser un disque  $d$  sur un disque  $d'$  si  $d$  est plus grand que  $d'$ . On suppose que les disques sont tous rangés sur le premier piquet,  $p_1$ , par ordre de grandeur avec le plus grand en dessous. On doit déplacer ces  $n$  disques vers le troisième piquet  $p_3$ . On cherche un algorithme (en pseudo-code ou en C) pour résoudre le problème pour  $n$  quelconque.

L'algorithme consistera en une fonction Déplacer-tour qui prend en entrée l'entier  $n$  et trois piquets et procède au déplacement des  $n$  disques du dessus du premier piquet vers le troisième piquet à l'aide de Déplacer-disque en utilisant si besoin le piquet intermédiaire. En C on utilisera les prototypes suivants sans détailler le type des piquets, piquet\_t ni le type des disques.

```
void deplacertour(unsigned n, piquet_t p1, piquet_t p2, piquet_t p3);
void deplacerdisque(piquet_t p, piquet_t q); /* p --disque--> q */
```

1. Indiquer une succession de déplacements de disques qui aboutisse au résultat pour  $n = 2$ .
2. En supposant que l'on sache déplacer une tour de  $n - 1$  disques du dessus d'un piquet  $p$  vers un autre piquet  $p'$ , comment déplacer  $n$  disques ?
3. Écrire l'algorithme en pseudo-code ou en donnant le code de la fonction deplacertour.
4. Combien de déplacements de disques fait-on exactement (trouver une forme close en fonction de  $n$ ) ?
5. Est-ce optimal (le démontrer) ?

**Exercice 4** (Le robot cupide).

Toto le robot se trouve à l'entrée Nord-Ouest d'un damier rectangulaire de  $N \times M$  cases. Il doit sortir par la sortie Sud-Est en descendant vers le Sud et en allant vers l'Est. Il a le choix à chaque pas (un pas = une case) entre : descendre verticalement ; aller en diagonale ; ou se déplacer horizontalement vers l'Est. Il y a un sac d'or sur chaque case, dont la valeur est lisible depuis la position initiale de Toto. Le but de Toto est de ramasser le plus d'or possible durant son trajet.

On veut écrire en pseudo-code ou en C, un algorithme *Robot-cupide*( $x, y$ ) qui, étant donné le damier et les coordonnées  $x$  et  $y$  d'une case, rend la quantité maximum d'or (gain) que peut ramasser le robot en se déplaçant du coin Nord-Ouest jusqu'à cette case. En C, on pourra considérer que le damier est un tableau bidimensionnel déclaré globalement et dont les dimensions sont connues.

A	B
C	D

1. Considérons quatre cases du damier comme ci-dessus et supposons que l'on connaisse le gain maximum du robot pour les cases A, B et C, quel sera le gain maximum pour la case D ?
2. Écrire l'algorithme.
3. Si le robot se déplace d'un coin à l'autre d'un damier carré  $4 \times 4$  combien de fois l'algorithme calcule-t-il le gain maximum sur la deuxième case de la diagonale ? Plus généralement, lors du calcul du gain maximum sur la case  $x, y$  combien y a-t'il d'appels au calcul du gain maximum d'une case  $i, j$  ( $i \leq x, j \leq y$ ).

### 6.1.2 Optimisation

**Exercice 5** (Exponentiation rapide).

L'objectif de cet exercice est de découvrir un algorithme rapide pour le calcul de  $x^n$  où  $x$  est un nombre réel et  $n \in \mathbb{N}$ . On cherchera à minimiser le nombre d'appels à des opérations arithmétiques sur les réels (addition, soustraction, multiplication, division) et dans une moindre mesure sur les entiers.

1. Écrire une fonction de prototype `double explent(double x, unsigned n)` qui calcule  $x^n$  (en C, ou bien en pseudo-code mais sans utiliser de primitive d'exponentiation).
2. Combien de multiplication sur des réels effectuera l'appel `explent(x, 4)` ?
3. Calculer à la main et en effectuant le moins d'opérations possibles :  $3^4, 3^8, 3^{16}, 3^{10}$ . Dans chaque cas combien de multiplications avez-vous effectué ?
4. Combien de multiplications suffisent pour calculer  $x^{256}$  ? Combien pour  $x^{32+256}$  ?

On note  $\overline{b_{k-1} \dots b_0}$  pour l'écriture en binaire des entiers positifs, où  $b_0$  est le bit de poids faible et  $b_{k-1}$  est le bit de poids fort. Ainsi

$$\overline{10011} = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 19.$$

De même que pour l'écriture décimale,  $b_{k-1}$  est en général pris non nul (en décimal, on écrit 1789 et non 00001789 – sauf sur le tableau de bord d'une machine à voyager dans le temps).

5. Comment calculer  $x^{\overline{10011}}$  en minimisant le nombre de multiplications ?
6. Plus généralement pour calculer  $x^{\overline{b_{k-1}\dots b_0}}$  de combien de multiplications sur les réels aurez-vous besoin (au maximum) ?

Rappels. Si  $n$  est un entier positif alors  $n \bmod 2$  (en C :  $n \% 2$ ) donne son bit de poids faible. La division entière par 2 décale la représentation binaire vers la droite :  $\overline{10111}/2 = \overline{10110}/2 = \overline{1011}$ .

7. Écrire une fonction double expzapide(double x, unsigned n) qui calcule  $x^n$ , plus rapidement que la précédente.
8. Si on compte une unité de temps à chaque opération arithmétique sur les réels, combien d'unités de temps sont nécessaires pour effectuer  $x^{1023}$  avec la fonction expzap ? Et avec la fonction expzapide ?
9. Même question, en général, pour  $x^n$  (on pourra donner un encadrement du nombre d'opérations effectuées par expzapide).
10. L'algorithme d'exponentiation rapide peut être considéré comme optimal asymptotiquement (ce résultat demanderait à être explicité mais il est trop difficile à établir pour ce cours). On peut toutefois faire un peu mieux sur certains cas (ça ne remet pas en cause le résultat asymptotique). Par exemple, on peut calculer  $x^{15}$  avec cinq multiplications, voyez-vous comment ?

### Exercice 6 (Drapeau, Dijkstra).

Les éléments d'un tableau (indexé à partir de 0) sont de deux couleurs, rouges ou verts. Pour tester la couleur d'un élément, on se donne une fonction  $\text{Couleur}(T, j)$  qui rend la couleur du  $j + 1$  ième élément (d'indice  $j$ ) du tableau  $T$ . On se donne également une fonction  $\text{Échange}(T, j, k)$  qui échange l'élément d'indice  $i$  et l'élément d'indice  $j$  et une fonction  $\text{Taille}(T)$  qui donne le nombre d'éléments du tableau.

En C, on utilisera les fonctions :

- int couleur(tableau\_t T, unsigned int j) rendant 0 pour rouge et 1 pour vert ;
- échange(tableau\_t T, unsigned int j, unsigned int k) ;
- unsigned int taille(tableau\_t T)

où le type des tableaux tableau\_t n'est pas explicité.

1. Écrire un algorithme (pseudo-code ou C) qui range les éléments d'un tableau en mettant les rouges en premiers et les verts en dernier. Contrainte : on ne peut regarder qu'une seule fois la couleur de chaque élément.
2. Même question, même contrainte, lorsqu'on ajoute des éléments de couleur bleue dans nos tableaux. On veut les trier dans l'ordre rouge, vert, bleu. On supposera que la fonction couleur rend 2 sur un élément bleu.

### Exercice 7 (rue Z).

Vous êtes au numéro zéro de la rue  $\mathbb{Z}$ , une rue infinie où les numéros des immeubles sont des entiers relatifs. Dans une direction, vous avez les immeubles numérotés 1, 2,

3, 4, ... et dans l'autre direction les immeubles numérotés  $-1, -2, -3, -4, \dots$ . Vous vous rendez chez un ami qui habite rue  $\mathbb{Z}$  sans savoir à quel numéro il habite. Son nom étant sur sa porte, il vous suffit de passer devant son immeuble pour le trouver (on suppose qu'il n'y a des immeubles que d'un côté et, par exemple, la mer de l'autre). On notera  $n$  la valeur absolue du numéro de l'immeuble que vous cherchez (bien entendu  $n$  est inconnu). Le but de cet exercice est de trouver un algorithme pour votre déplacement dans la rue  $\mathbb{Z}$  qui permette de trouver votre ami à coup sûr et le plus rapidement possible (d'un point de vue asymptotique).

1. Montrer que n'importe quel algorithme sera au moins en  $\Omega(n)$  pour ce qui est de la distance parcourue.
2. Trouver un algorithme efficace, donner sa complexité en distance parcourue sous la forme d'un  $\Theta(g)$ . Démontrer votre résultat.

### 6.1.3 Notation asymptotique

**Exercice 8** (Notation asymptotique (devoir 2006)).

1. Ces phrases ont elles un sens (expliquer) :
  - le nombre de comparaisons pour ce tri est au plus  $\Omega(n^3)$ ;
  - en pire cas on fait au moins  $\Theta(n)$  échanges.
2. Est-ce que  $2^{n+1} = O(2^n)$  ? Est-ce que  $2^{2^n} = O(2^n)$  ?
3. Démontrer :

$$\text{si } f(n) = O(g(n)) \text{ et } g(n) = O(h(n)) \text{ alors } f(n) = O(h(n)) \quad (6.1)$$

$$\text{si } f(n) = O(g(n)) \text{ alors } g(n) = \Omega(f(n)) \quad (6.2)$$

$$\text{si } f(n) = \Omega(g(n)) \text{ alors } g(n) = O(f(n)) \quad (6.3)$$

$$\text{si } f(n) = \Theta(g(n)) \text{ alors } g(n) = \Theta(f(n)) \quad (6.4)$$

**Exercice 9** (Partiel juin 2006).

Répondre par oui ou par non, sans justification et pour la seconde question donner seulement la borne.

1. Si  $f(n) = 10n + 100$ , est-ce que  $f(n) = O(n)$  ?  $f(n) = O(n \log n)$  ?  $f(n) = \Theta(n^2)$  ?  $f(n) = \Omega(\log n)$  ?  0.5 pt  
4 min
2. En notation asymptotique quelle est la borne minimale en temps des tris par comparaison, en pire cas et en moyenne ?  0.5 pt  
4 min

**Exercice 10** (Juin 2007).

Pour chacune des assertions suivantes, dire si elle est vraie ou fausse et **justifier** votre réponse en rappelant la définition.

1.  $\frac{n \log n}{2} = \Omega(n)$   1 pt  
9 min
2.  $\log(n!) = O(n \log n)$   1 pt  
9 min
3.  $n^3 + n^2 + n + 1 = \Theta(n^3)$   1 pt  
9 min

**Exercice 11** (Partiel mai 2008).

Rappeler les définitions utilisées et justifier (démontrer) vos réponses.

1. Est-ce que  $n^2 - 2n + 1 = O(n^2)$  ?
2. Est-ce que  $\sum_{i=1}^n \log i = \Omega(n)$  ?
3. Est-il vrai que si  $f = O(g)$  et  $f = \Omega(h)$  alors  $g = \Omega(h)$  ?

1 pt  
9 min

**Exercice 12** (Notation asymptotique).

Rappeler les définitions utilisées et justifier (démontrer) vos réponses à partir de ces définitions.

1. Est-ce que  $n \log n = O(n^2)$  ?
2. Est-ce que  $\log(n!) = O(n \log n)$  ?
3. Est-ce que  $\log(n!) = \Theta(n^2)$  ?

**Exercice 13** (Partiel mi-semestre 2006).

Rappeler les définitions utilisées et justifier (démontrer) vos réponses.

1. Est-ce que  $(n + 3) \log n - 2n = \Omega(n)$  ?
2. Est-ce que  $2^{2n} = O(2^n)$  ?

**Exercice 14** (Partiel mars 2008).

Rappeler les définitions utilisées et justifier vos réponses.

1. Est-ce que  $\sum_{i=1}^n i = \Theta(n^2)$  ?
2. Est-ce que  $n^2 = \Omega(2^n)$  ?
3. Est-ce que  $\sum_{i=0}^n \left(\frac{2}{3}\right)^i = O(1)$  ?

**Exercice 15** (Partiel 2007).

Rappeler les définitions utilisées et justifier vos réponses.

1. Est-ce que  $\log(n/2) = \Omega(\log n)$  ?
2. Est-ce que  $n = \Omega(n \log n)$  ?
3. Est-ce que  $\log(n!) = O(n \log n)$  ?
4. Soit un algorithme  $A$ . Est-il correct de dire du temps d'exécution de  $A$  que : si le pire cas est en  $O(f(n))$  et le meilleur cas en  $\Omega(f(n))$  alors en moyenne  $A$  est en  $\Theta(f(n))$  ?

**Exercice 16** (Septembre 2007).

En rappelant les définitions, démontrer chacune des assertions suivantes.

1.  $n^2 = \Omega(n \log n)$
2. Si  $f = \Theta(h)$  et  $g = O(h)$  alors  $f + g = O(h)$
3. Il est faux de dire que, en général, si  $f = O(g)$  et  $g = \Omega(h)$  alors  $f = \Omega(h)$ .  
Donner un contre-exemple.

**Exercice 17.**

En rappelant les définitions, démontrer chacune des assertions suivantes.

1.  $1 + \sum_{i=2}^n \log i = \Omega(n)$
2. Si  $f = \Theta(h)$  et  $g = O(h)$  alors  $f + g = O(h)$
3. Il est faux de dire que, en général, si  $f = O(g)$  et  $g = \Omega(h)$  alors  $f = \Omega(h)$   
(Donner un contre-exemple argumenté).

**Exercice 18** ((colle 2007)).

On suppose que `f_aux(k)` est une procédure qui requiert un temps d'exécution en  $\Theta(k)$ .

```
void f(int n){
    int i, j;
    for (i = 0; i < n; i++){
        for ( j = i; j < n; j++) {
            f_aux(j);
        }
    }
}
```

En supposant que les seules opérations significatives pour le temps d'exécution sont effectuées par `f_aux`, donner un équivalent asymptotique du temps d'exécution de `f`.

 2 pt  
18 min

**Exercice 19.**

Montrer que :

$$\log(n!) = \Omega(n \log n). \quad (6.5)$$

## 6.2 Exercices du second chapitre

### Exercice 20 (Tri sélection).

Soit un tableau indexé à partir de 0 contenant des éléments deux à deux comparables. Par exemple des objets que l'on compare par leurs masses. On dispose pour cela d'une fonction

$$\text{Comparer}(T, j, k) \text{ qui rend : } \begin{cases} -1 \text{ si } T[j] > T[k] \\ 1 \text{ si } T[j] < T[k] \\ 0 \text{ lorsque } T[j] = T[k] \text{ (même masses).} \end{cases}$$

1. Écrire un algorithme *Minimum* qui rend le premier indice auquel apparaît le plus petit élément du tableau  $T$ .
2. Combien d'appels à la comparaison effectue votre fonction sur un tableau de taille  $N$  ?

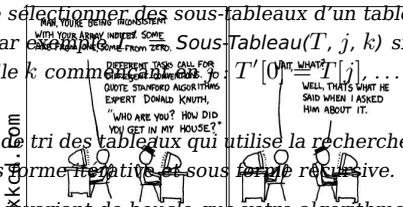
On dispose également d'une fonction *Échanger*( $T, j, k$ ) qui échange  $T[j]$  et  $T[k]$ . On se donne aussi la possibilité de sélectionner des sous-tableaux d'un tableau  $T$  à l'aide d'une fonction *Sous-Tableau*. Par *Sous-Tableau*( $T, j, k$ ) signifie que  $T'$  est le sous-tableau de  $T$  de taille  $k$  commençant à l'indice  $j$  :  $T'[0] = T[j], \dots, T'[k-1] = T[j+k-1]$ .

3. Imaginer un algorithme de tri des tableaux qui utilise la recherche du minimum du tableau. L'écrire sous forme itérative et sous forme récursive.
4. Démontrer à l'aide d'un invariant de boucle que votre algorithme itératif de tri est correct.
5. Démontrer que votre algorithme récursif est correct. Quelle forme de raisonnement très courante en mathématiques utilisez-vous à la place de la notion d'invariant de boucle ?
6. Combien d'appels à la fonction *Minimum* effectuent votre algorithme itératif et votre algorithme récursif sur un tableau de taille  $N$  ? Combien d'appels à la fonction *Comparer* cela représente-t-il ? Combien d'appels à *Échanger* ? Donner un encadrement et décrire un tableau réalisant le meilleur cas et un tableau réalisant le pire cas.
7. Vos algorithmes fonctionnent-ils dans le cas où plusieurs éléments du tableau sont égaux ?

### Exercice 21 (Interclassement).

Soient deux tableaux d'éléments comparables  $t_1$  et  $t_2$  de tailles respectives  $n$  et  $m$ , tous les deux triés dans l'ordre croissant.

1. Écrire un algorithme d'interclassement des tableaux  $t_1$  et  $t_2$  qui rend le tableau trié de leurs éléments (de taille  $n + m$ ).



On note  $N = n + m$  le nombre total d'éléments à interclasser. En considérant le nombre de comparaisons, en fonction de  $N$ , discuter l'optimalité de votre algorithme en pire cas et en meilleur cas à l'aide des questions suivantes (démontrer vos résultats).

2. À votre avis, sans considérer un algorithme en particulier, dans quel cas peut-il être nécessaire de faire le plus de comparaisons :  $n$  grand et  $m$  petit ou bien  $n$  et  $m$  à peu près égaux ?

Dans la suite, on suppose que  $n$  et  $m$  sont égaux (donc  $N = 2n$ ).

3. Dans le pire des cas, combien de comparaisons faut-il faire pour réussir l'interclassement ? Cela correspond t-il au nombre de comparaisons effectuées par votre algorithme dans ce cas ?
4. Toujours sous l'hypothèse  $n = m$ , quel est le meilleur cas ? En combien de comparaisons peut-on le résoudre ? Comparer avec votre algorithme.

**Exercice 22** (Complexité en moyenne du tri bulle (devoir 2006)).

Le but de cet exercice est de déterminer le nombre moyen d'échanges effectués au cours d'un tri bulle.

On considère l'implémentation suivante du tri bulle :

```
0 void tribulle(tableau_t *t){
1   int n, k, fin;
2   for (n = taille(t) - 1; n >= 1; n--) {
3     /* Les éléments d'indice > n sont à la bonne place. */
4     fin = 1;
5     for (k = 0; k < n; k++){
6       if ( 0 > comparer(t[k], t[k + 1]) ){ /* t[k] > t[k + 1] */
7         echangertab(t, k, k + 1);
8         fin = 0;
9       }
10    }
11    if (fin) break; /* Les éléments entre 0 et n - 1 sont bien ordonnés */
12  }
13 }
```

On considère le tableau passé en entrée comme une permutation des entiers de 0 à  $n - 1$  que le tri remettra dans l'ordre 0, 1, 2, ...,  $n - 1$ . Ainsi, pour  $n = 3$ , on considère qu'il y a 6 entrées possibles : 0, 1, 2; 0, 2, 1; 1, 0, 2; 1, 2, 0; 2, 0, 1 et 2, 1, 0.

On fait l'hypothèse que toutes les permutations sont équiprobables.

Une inversion dans une entrée  $a_0, \dots, a_{n-1}$  est la donnée de deux indices  $i$  et  $j$  tels que  $i < j$  et  $a_i > a_j$ .

1. Combien y a-t-il d'inversions dans la permutation 0, 1, ...,  $n - 1$  ? Et dans la permutation  $n - 1, n - 2, \dots, 0$  ?
2. Montrer que chaque échange dans le tri bulle élimine exactement une inversion.

3. En déduire une relation entre le nombre total d'inversions dans toutes les permutations de  $0, \dots, n-1$  et le nombre moyen d'échanges effectués par le tri bulle sur une entrée de taille  $n$ .

L'image miroir de la permutation  $a_0, a_1, \dots, a_{n-1}$  est la permutation  $a_{n-1}, a_{n-2}, \dots, a_0$ .

4. Montrer que l'ensemble des permutations de  $0, \dots, n-1$  est en bijection avec lui-même par image miroir.
5. Si  $(i, j)$  est une inversion dans la permutation  $a_0, a_1, \dots, a_{n-1}$ , qu'en est-il dans son image miroir? Réciproquement? En déduire le nombre moyen d'inversions dans une permutation des entiers de  $0$  à  $n-1$  et le nombre moyen d'échanges effectués par le tri bulle.

**Exercice 23** (Complexité en moyenne du tri gnome (partiel mi-semester 2006)).

Le but de cet exercice est d'écrire le tri gnome en C et de déterminer le nombre moyen d'échanges effectués au cours d'un tri gnome.

tot: 7 pt

Rappel du cours. « Dans le tri gnome, on compare deux éléments consécutifs : s'ils sont dans l'ordre on se déplace d'un cran vers la fin du tableau (ou on s'arrête si la fin est atteinte); sinon, on les intervertit et on se déplace d'un cran vers le début du tableau (si on est au début du tableau alors on se déplace vers la fin). On commence par le début du tableau. »

1. Écrire une fonction C de prototype `void trignome(tableau_t t)` effectuant le tri gnome.

Une inversion dans une entrée  $a_0, \dots, a_{n-1}$  est la donnée d'un couple d'indices  $(i, j)$  tel que  $i < j$  et  $a_i > a_j$ .

Rappel. Un échange d'éléments entre deux indices  $i$  et  $j$  dans un tableau est une opération qui intervertit l'élément à l'indice  $i$  et l'élément à l'indice  $j$ , laissant les autres éléments à leur place.

2. Si le tri gnome effectue un échange entre deux éléments, que peut on dire de l'évolution du nombre d'inversions dans ce tableau avant l'échange et après l'échange (démontrer)?

On suppose que le nombre moyen d'inversions dans un tableau de taille  $n$  est  $\frac{n(n-1)}{4}$ .

3. Si un tableau  $t$  de taille  $n$  contient  $f(n)$  inversions, combien le tri gnome effectuera d'échanges sur ce tableau (démontrer)? En déduire le nombre moyen d'échanges effectués par le tri gnome sur des tableaux de taille  $n$ .

**Exercice 24** (Arbre de décision, meilleur cas (partiel 2007)).

Rappel : le tri bulle s'arrête lorsqu'il a fait une passe au cours de laquelle il n'y a eu aucun échange.

1. Dessiner l'arbre de décision du tri bulle sur un tableau de trois éléments  $[a, b, c]$ .
2. On note  $C(N)$  le nombre de comparaisons faites par le tri bulle dans le meilleur des cas sur un tableau de taille  $N$ . Quel tableau en entrée donne le meilleur cas du tri bulle? Combien vaut  $C(N)$  exactement? (En fonction de  $N$ .)

tot: 4,5 pt

1 pt  
9 min

0,5 pt  
4 min

On raisonne maintenant sur les algorithmes de tri généralistes (par comparaison).

3. Est-il possible qu'un algorithme de tri fasse moins que  $N - 1$  comparaisons en meilleur cas ? (Démontrer.)  1 pt  
9 min
4. Rappeler la borne asymptotique inférieure du nombre de comparaisons nécessaires dans un tri généraliste.  0,5 pt  
4 min
5. Sans utiliser le résultat que vous venez de rappeler, montrer que pour  $N > 2$ , il n'y a pas d'algorithme  $A$  qui trie n'importe quel tableau de taille  $N$  en au plus  $N - 1$  comparaisons. (Considérer les  $N!$  permutations de  $0, \dots, N - 1$  en entrée et raisonner sur la hauteur de l'arbre de décision de  $A$ .)  1,5 pt  
13 min

**Exercice 25** (Min et max (partiel 2007)).

On se donne un tableau d'entiers  $T$ , non trié, de taille  $N$ . On cherche dans  $T$  le plus petit entier (le minimum) ainsi que le plus grand (le maximum). Si vous écrivez en  $C$  : ne vous souciez pas de la manière de rendre les deux entiers : `return(a, b)` où  $a$  est le minimum et  $b$  le maximum sera considéré comme correct.

tot: 3 pt

1. Écrire un algorithme  $Minimum(T)$  ( $C$  ou pseudo-code) qui prend en entrée le tableau  $T$  et rend le plus petit de ses entiers.  1 pt  
9 min

Pour trouver le maximum, on peut d'écrire l'algorithme  $Maximum(T)$  équivalent (en inversant simplement la relation d'ordre dans  $Minimum(T)$ ).

On peut alors écrire une fonction  $MinEtMax(T)$  qui renvoie  $(Minimum(T), Maximum(T))$ .

2. Combien la fonction  $MinEtMax$  fera-t-elle de comparaisons, exactement, sur un tableau de taille  $N$  ?  1 pt  
9 min

On propose la méthode suivante pour la recherche du minimum et du maximum. Supposons pour simplifier que  $N$  est pair et non nul.

On considère les éléments du tableau par paires successives :  $(T[0], T[1])$  puis  $(T[2], T[3])$ ,  $(T[4], T[5])$ ,  $\dots$   $(T[N - 2], T[N - 1])$ .

- On copie la plus petite valeur entre  $T[0]$  et  $T[1]$  dans une variable  $Min$  et la plus grande dans une variable  $Max$ .
- Pour chacune des paires suivantes,  $(T[2i], T[2i + 1])$  :
  - on trouve le minimum entre  $T[2i]$  et  $T[2i + 1]$  et on le range dans  $MinLocal$  de même le maximum entre  $T[2i]$  et  $T[2i + 1]$  est rangé dans  $MaxLocal$  ;
  - On trouve le minimum entre  $Min$  et  $MinLocal$  et on le range dans  $Min$  de même le maximum entre  $MaxLocal$  et  $Max$  est rangé dans  $Max$ .
- On rend  $Min$  et  $Max$ .

3. On réalise cet algorithme avec un minimum de comparaisons pour chaque étape : expliquer quelles seront les comparaisons (mais inutile d'écrire tout l'algorithme). Combien fait-on de comparaisons au total ?  1 pt  
9 min

**Exercice 26.**

Étant donné un tableau  $T$  de  $N$  entiers et un entier  $x$ , on veut déterminer s'il existe deux éléments de  $T$  dont la somme est égale à  $x$ .

3 pt  
27 min

1. Donner un algorithme le plus simple possible, basé sur la comparaison, sans faire appel à des algorithmes du cours. Quel est le pire cas ? Donner un équivalent asymptotique du nombre de comparaisons dans le pire cas. (Justifier)
2. Pouvez-vous donner un algorithme en  $O(N \log N)$  comparaisons en pire cas ? (Justifier) Vous pouvez utiliser des algorithmes vus en cours et les résultats de complexité sur ces algorithmes.

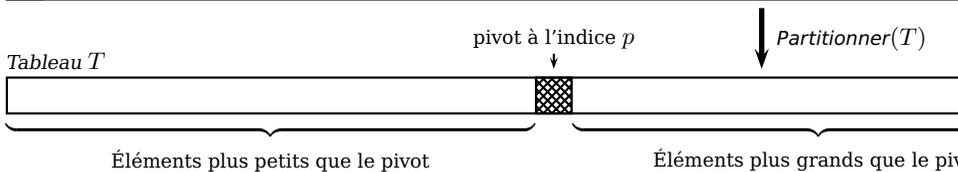
**Exercice 27.**

Soit un tableau  $T$  de  $N$  éléments deux à deux comparables. On souhaite partitionner le tableau autour de son premier élément  $T[0]$ , appelé pivot. Après partition, le pivot est à l'indice  $p$ , les éléments plus petits que le pivot sont (dans le désordre) aux indices inférieurs à  $p$  et les éléments strictement plus grands que le pivot sont (dans le désordre) aux indices supérieurs à  $p$ .

Tableau  $T$



Tableau  $T$



Écrire l'algorithme de partitionnement de manière à ce qu'il effectue  $N - 1$  (ou  $N$ ) comparaisons.

**Exercice 28** (Invariant de boucle).

Étant donné un tableau  $T$  de  $N > 0$  entiers, l'algorithme *Maximum* renvoie l'indice de l'élément maximum de  $T$ . Démontrer le à l'aide d'un invariant de boucle.

---

**Fonction** Maximum( $T$ )

---

$m = 0;$

**pour**  $i \leftarrow 1$  à Taille( $T$ ) - 1

**faire**

**si**  $T[i] \geq T[m]$  **alors**  
         $m = i;$

**retourner**  $m;$

---

**Exercice 29.**

Le but de cet exercice est trouver un bon algorithme pour la recherche de valeurs médianes. En économie, le revenu médian est le revenu qui partage exactement en deux la population : la moitié de la population dispose d'un revenu plus élevé que le

2 pt  
18 min

1.5 pt  
13 min

tot: 5 pt

revenu médian, l'autre moitié d'un revenu moins élevé. Plus généralement, dans un tableau l'élément médian (ou valeur médiane) est l'élément qui serait situé au milieu du tableau si le tableau était trié. Lorsque le nombre d'éléments est pair, il n'y a pas d'élément exactement au milieu. Par convention nous prendrons, pour tout  $N$ , comme médian l'élément d'indice  $\lfloor \frac{N}{2} \rfloor$  dans le tableau trié (indexé à partir de 0).

Par exemple, dans le tableau suivant le revenu médian est de 1200 €.

individus	A	B	C	D	E	F	G	H
revenus mensuels	1400	2000	1300	300	700	5000	1200	800

Pour trouver le médian d'un tableau d'éléments deux à deux comparables on peut trier le tableau puis renvoyer la valeur de son élément d'indice  $\lfloor \frac{N}{2} \rfloor$ .

1. Pour cette solution, quelle complexité en moyenne (en temps) peut on obtenir; au mieux? Quel algorithme de tri choisir?

 .5 pt  
4 min

On cherche un algorithme plus rapide. Il n'est sans doute pas nécessaire de trier tout le tableau pour trouver le médian. Le professeur Hoare suggère d'utiliser l'algorithme de partition de l'exercice 27 pour diviser les éléments à considérer. Après partition le tableau  $T$  est fait de trois parties : la première partie est un tableau  $T'$  des éléments de  $T$  plus petits que le pivot la deuxième partie ne contient que l'élément pivot et la troisième partie est un tableau  $T''$  des éléments plus grands que le pivot.

2. En fonction de l'indice  $p$  du pivot, dans quelle partie chercher le médian?

 .5 pt  
4 min

En général, on ne trouve pas le médian après le premier partitionnement. L'idée de Hoare est de continuer à partitionner la partie dans laquelle on doit chercher le médian, jusqu'à le trouver.

3. Dans quelle partie chercher le médian à chaque étape? Répondre en donnant l'algorithme complet. Si nécessaire vous pouvez considérer que l'algorithme de partitionnement renvoie un triplet  $(T', p, T'')$  où  $T'$  et  $T''$  sont les deux sous-tableaux de  $T$  évoqués plus haut et  $p$  est l'indice dans  $T$  du pivot.

 2 pt  
18 min

On suppose que le partitionnement d'un tableau de  $N$  éléments se fait exactement en  $N$  comparaisons ( $N - 1$  serait également possible). On s'intéresse à la complexité asymptotique de notre algorithme de recherche du médian, exprimée en nombre de comparaisons.

4. Quel est le meilleur cas? Pour quelle complexité? Quel est le pire cas? Pour quelle complexité?

 1 pt  
9 min

Pour estimer si la moyenne est plus proche du meilleur cas ou du pire cas, on fait l'hypothèse que chaque fois que l'on fait une partition sur un tableau  $T$ , le médian se trouve dans un sous-tableau contenant  $\frac{2}{3}$  des éléments de  $T$ .

5. Donner un équivalent asymptotique du nombre de comparaisons faites (on pourra s'aider de l'exercice 14).

 .5 pt  
4 min

6. En supposant que ce résultat représente la complexité moyenne, l'algorithme est-il asymptotiquement optimal en moyenne?

 .5 pt  
4 min

Dans cette partie on s'intéresse au problème suivant : étant donné un tableau  $T$  de  $N$  éléments deux à deux comparables, déterminer quel est l'élément de rang  $k$  (le  $k$ -ième plus petit élément), c'est à dire l'élément  $x$  de  $T$  tel que exactement  $k - 1$  éléments de  $T$  sont plus petits que  $x$ . Bien entendu, on peut supposer que  $k$  est choisi tel que  $1 \leq k \leq N$ . On pourra considérer que les éléments de  $T$  sont distincts (la comparaison ne les déclare jamais égaux). Si on a par exemple les éléments 23, 62, 67, 56, 34, 90, 17 ( $N$  vaut 7) alors l'élément de rang 3 est 34.

**Exercice 30.**

Le moyen le plus simple d'écrire une fonction  $\text{Rang}(T, k)$  résolvant ce problème est de trier le tableau et de renvoyer l'élément d'indice  $k - 1$  du tableau obtenu (les tableaux commencent à l'indice 0). En choisissant au mieux l'algorithme de tri, quel sera en pire cas le temps d'exécution de cette fonction ?

On souhaite évaluer la complexité d'une autre solution à ce problème, dérivée du fonctionnement du tri rapide. Pour calculer  $\text{Rang}(T, k)$  on utilise l'algorithme suivant :

---

**Fonction**  $\text{Rang}(T, k)$

---

$p = \text{Partitionner}(T);$

**si**  $p + 1 = k$  **alors**

  └ retourner  $T[p];$

**si**  $k < p + 1$  **alors**

  └ retourner  $\text{Rang}(T[0 \dots p - 1], k);$

**si**  $k > p + 1$  **alors**

  └ retourner  $\text{Rang}(T[p + 1 \dots \text{Taille}(T) - 1], k - p - 1);$

---

Il s'agit de partitionner le tableau  $T$  autour de la valeur  $x$  contenue à l'indice 0, appelée *pivot*, après quoi le tableau contient : les éléments plus petits que  $x$  (dans le désordre), puis  $x$  à un certain indice  $p$ , puis les éléments plus grands que  $x$  (dans le désordre). La fonction de partitionnement travaille en place et renvoie le nouvel indice  $p$  de l'élément qui a servi de pivot ( $x$ ).

Si l'indice  $p$  coïncide avec le rang recherché (c'est à dire si  $p + 1 = k$ ) alors c'est terminé et on renvoie le pivot de ce partitionnement  $x = T[p]$  qui est bien l'élément de rang  $k$ . Sinon, il y a deux cas selon s'il faut chercher à gauche ou à droite de  $x$  : si  $k$  est plus petit que  $p + 1$  on cherche l'élément de rang  $k$  dans le sous-tableau des éléments plus petits que  $x$ , et, si  $k$  est plus grand que  $p + 1$  on cherche dans le sous tableau des éléments plus grands que  $x$  l'élément de rang  $k - p - 1$  (puisque  $p + 1$  éléments sont déjà plus petits).

**Exercice 31.**

On suppose que le partitionnement d'un tableau  $T$  replace les éléments plus petits que

0.5 pt  
4 min

le pivot dans le même ordre qu'ils étaient avant partitionnement et de même pour les éléments plus grands. En représentant les tableaux et sous-tableaux obtenus successivement, exécuter à la main  $\text{Rang}(T, 4)$  où  $T$  contient dans cet ordre les éléments : 3, 2, 8, 6, 9, 1, 5.

 1 pt  
9 min

### Exercice 32.

En supposant que partitionner un tableau de  $n$  éléments prend un temps  $n$ , on étudie le temps d'exécution global (appels récursifs compris) de  $\text{Rang}(T, k)$  en fonction de la taille  $N$  de  $T$ , en notation asymptotique.

 3 pt  
27 min

1. Quel est le meilleur cas et quel temps obtient-on ? (Donner un exemple générique de tableau et de valeur de  $k$  réalisant ce meilleur cas et un équivalent asymptotique du temps d'exécution).
2. Quel est le pire cas et quel temps obtient-on ? (Donner un exemple générique de tableau et de valeur de  $k$  réalisant ce pire cas et un équivalent asymptotique du temps d'exécution).
3. Supposons que  $T$  est de taille  $N = 2^m$  et qu'au cours de la recherche de l'élément de rang 1, chaque partitionnement d'un tableau de taille  $n$  donne  $\frac{n}{2}$  éléments plus petits que le pivot (c'est à dire que la fonction de partitionnement renvoie l'indice  $p = \frac{n}{2}$ ) jusqu'à arriver à la taille 1 dans la récursion. Quel est alors le temps d'exécution en notation asymptotique ? (Ne pas chercher d'exemple de tableau réalisant ce cas).
4. Que penser du temps d'exécution dans le cas plus général où, après chaque partition d'un tableau de taille  $n$ , l'appel récursif a lieu sur un tableau de taille  $r \times n$  où  $r < 1$  est une proportion fixée globalement (à la question précédente  $r$  valait  $\frac{1}{2}$ ) ?

### Exercice 33 (Tris en temps linéaire 1).

On se donne un tableau de taille  $n$  en entrée et on suppose que ses éléments sont des entiers compris entre 0 et  $n - 1$  (les répétitions sont autorisées).

1. Trouver une méthode pour trier le tableau en temps linéaire,  $\Theta(n)$ .
2. Même question si le tableau en entrée contient des éléments numérotés de 0 à  $n - 1$ . Autrement dit, chaque élément possède une clé qui est un entier entre 0 et  $n - 1$  mais il contient aussi une autre information (la clé est une étiquette sur un produit, par exemple).
3. lorsque les clés sont des entiers entre  $-n$  et  $n$ , cet algorithme peut-il être adaptée en un tri en temps linéaire ? Et lorsque on ne fait plus de supposition sur la nature des clés ?

### Exercice 34 (Tri par base (partiel mi-semester 2006)).

Soit la suite d'entiers décimaux 141, 232, 045, 112, 143. On utilise un tri stable pour trier ces entiers selon leur chiffre le moins significatif (chiffre des unités), puis pour trier la liste obtenue selon le chiffre des dizaines et enfin selon le chiffre le plus significatif (chiffre des centaines).

tot: 10 pt

Rappel. Un tri est stable lorsque, à chaque fois que deux éléments ont la même clé, l'ordre entre eux n'est pas changé par le tri. Par exemple, en triant  $(2, a), (3, b), (1, c), (2, d)$  par chiffres croissants, un tri stable place  $(2, d)$  après  $(2, a)$ .

1. Écrire les trois listes obtenues. Comment s'appelle cette méthode de tri ?

On se donne un tableau  $t$  contenant  $N$  entiers entre 0 et  $10^k - 1$ , où  $k$  est une constante entière. Sur le principe de la question précédente (où  $k = 3$  et  $N = 5$ ), on veut appliquer un tri par base, en base 10 à ces entiers.

On se donne la fonction auxiliaire :

```
int cle(int x, int i){
    int j;
    for (j = 0; j < i; j++)
        x = x / 10; // <- arrondi par partie entière inférieure.
    return x % 10;
}
```

2. Que valent  $\text{cle}(123, 0)$ ,  $\text{cle}(123, 1)$ ,  $\text{cle}(123, 2)$  (inutile de justifier votre réponse) ? Plus généralement, que renvoie cette fonction ?

On suppose que l'on dispose d'une fonction auxiliaire de tri void  $\text{triaux}(\text{tableau\_t } t, \text{ int } i)$  qui réordonne les éléments de  $t$  de manière à ce que

$$\text{cle}(t[0], i) \leq \text{cle}(t[1], i) \leq \dots \leq \text{cle}(t[N - 1], i).$$

On suppose de plus que ce tri est stable.

- Écrire l'algorithme de tri par base du tableau  $t$  (utiliser la fonction  $\text{triaux}$ ). On pourra considérer que  $k$  est un paramètre entier passé à la fonction de tri.
- Si le temps d'exécution en pire cas de  $\text{triaux}$  est majoré asymptotiquement par une fonction  $f(N)$  de paramètre la taille de  $t$ , quelle majoration asymptotique pouvez donner au temps d'exécution en pire cas de votre algorithme de tri par base ?
- Démontrer par récurrence que ce tri par base trie bien le tableau  $t$ . Sur quelle variable faites vous la récurrence ? Où utilisez vous le fait que  $\text{triaux}$  effectue un tri stable ?
- La fonction  $\text{triaux}$  utilise intensivement la fonction à deux paramètres  $\text{cle}$ . Si on cherche un majorant  $f(N)$  au temps d'exécution de  $\text{triaux}$ , peut on considérer qu'un appel à  $\text{cle}$  prend un temps borné par une constante ?
- Décrire en quelques phrases une méthode pour réaliser la fonction  $\text{triaux}$  de manière à ce qu'elle s'exécute en un temps linéaire en fonction de la taille du tableau (on pourra utiliser une structure de donnée).

**Exercice 35** (Plus grande sous-suite équilibrée).

On considère une suite finie  $s = (s_i)_{0 \leq i \leq n-1}$  contenant deux types d'éléments  $a$  et  $b$ . Une sous-suite équilibrée de  $s$  est une suite d'éléments consécutif de  $s$  où l'élément  $a$  et l'élément  $b$  apparaissent exactement le même nombre de fois. L'objectif de cet

exercice est de donner un algorithme rapide qui prend en entrée une suite finie  $s$  ayant deux types d'éléments et qui rend la longueur maximale des sous-suites équilibrées de  $s$ .

Par exemple, si  $s$  est la suite  $aababba$  alors la longueur maximale des sous-suites équilibrées de  $s$  est 6. Les suites  $aababb$  et  $ababba$  sont deux sous-suites équilibrées de  $s$  de cette longueur.

Pour faciliter l'écriture de l'algorithme, on considérera que :

- la suite en entrée est donnée dans un tableau de taille  $n$ , avec un élément par case;
- chaque cellule de ce tableau est soit l'entier 1 soit l'entier  $-1$  (et non pas  $a$  et  $b$ ).

1. Écrire une fonction qui prend deux indices  $i$  et  $j$  du tableau, tels que  $0 \leq i < j < n$ , et rend 1 si la sous-suite  $(s_k)_{i \leq k \leq j}$  est équilibrée, 0 sinon.
2. Écrire une fonction qui prend en entrée un indice  $i$  et cherche la longueur de la plus grande sous-suite équilibrée commençant à l'indice  $i$ .
3. En déduire une fonction qui rend la longueur maximale des sous-suites équilibrées de  $s$ .
4. Quel est la complexité asymptotique de cette fonction, en temps et en pire cas ?
5. Écrire une fonction qui prend en entrée le tableau  $t$  des éléments de la suite  $s$  et crée un tableau d'entiers  $aux$ , de même taille que  $t$  et tel que  $aux[k] = \sum_{j=0}^k s_j$ .
6. Pour que  $(s_k)_{i \leq k \leq j}$  soit équilibrée que faut-il que  $aux[i]$  et  $aux[j]$  vérifient ?

Supposons maintenant que chaque élément de  $aux$  est en fait une paire d'entiers, (clé, donnée), que la clé stockée dans  $aux[k]$  est  $\sum_{j=0}^k s_j$  et que la donnée est simplement  $k$ .

7. Quelles sont les valeurs que peuvent prendre les clés dans  $aux$  ?
8. À votre avis, est-il possible de trier  $aux$  par clés croissantes en temps linéaire ? Si oui, expliquer comment et si non, pourquoi.
9. Une fois que le tableau  $aux$  est trié par clés croissantes, comment l'exploiter pour résoudre le problème de la recherche de la plus grande sous-suite équilibrée ?
10. Décrire de bout en bout ce nouvel algorithme. Quelle est sa complexité ?
11. Écrire complètement l'algorithme.

**Exercice 36** (Plus grand sous-tableau à somme nulle).

Soit un tableau  $T$  de  $N$  entiers. On cherche un sous-tableau (contigu) de  $T$ , tel que la somme des éléments de ce sous-tableau soit nulle. De plus on souhaite que ce sous-tableau soit le plus grand possible. Un sous-tableau  $T'$  non-vide de  $T$  est donné par un couple d'indices  $(i, j)$  avec  $0 \leq i \leq j \leq N - 1$ , les éléments de  $T'$  sont alors  $T[i], T[i + 1], \dots, T[j]$ .

On peut pour cela calculer toutes les sommes pour tous les sous-tableaux (non-vides) possibles de  $T$  et parmi ceux pour lesquels cette somme est nulle en trouver un de longueur maximum.

tot: 3 pt
-----------

1. Quel serait le temps d'exécution d'un algorithme fondé sur cette méthode, en notation asymptotique et en fonction de  $N$ ? (Il faut expliquer comment vous écririez l'algorithme mais sans nécessairement le détailler).

Supposons maintenant que l'on fabrique un autre tableau  $S$  de même taille que  $T$  de la manière suivante. Chaque élément  $S[i]$  est un couple d'entiers. Au départ, le premier de ces deux entiers,  $S[i].\text{somme}$  en  $C$ , est égal à la somme  $\sum_{j=0}^{i-1} T[j]$ . Le second,  $S[i].\text{indice}$ , a simplement pour valeur l'indice  $i$ . On tri ensuite  $S$  par sommes croissantes à l'aide d'un tri stable.

2. Comment peut on résoudre le problème initial à l'aide de  $S$ ? (Décrire l'algorithme sans nécessairement le détailler, en supposant que  $S$  trié est fourni).
3. Quel temps d'exécution global peut on obtenir par cette méthode (donner un temps d'exécution pour chaque étape).

### Exercice 37.

Classer les fonctions de complexité  $n \log n$ ,  $2^n$ ,  $\log n$ ,  $n^2$ ,  $n$  par ordre croissant et pour chacune d'elle donner l'exemple d'un algorithme (du cours ou des TD) qui a asymptotiquement cette complexité en pire cas, en temps. Répondre dans un tableau en donnant le nom de l'algorithme ou le nom du problème qu'il résout.

### Exercice 38.

Classer les fonctions de complexité  $n \log n$ ,  $\log n$ ,  $n^2$ ,  $n$  par ordre croissant. Pour les complexités  $\log n$  et  $n$  donner l'exemple d'un algorithme (du cours ou des TD) qui a asymptotiquement cette complexité en pire cas, en temps.

### Exercice 39.

Classer les fonctions de complexité  $n^2$ ,  $n$ ,  $2^n$ ,  $n \log n$  par ordre croissant. Pour les complexités  $n \log n$  et  $n^2$  donner l'exemple d'un algorithme (du cours ou des TD) qui a asymptotiquement cette complexité en pire cas, en temps.

### Exercice 40.

Soit la fonction *MaFonction* donnée ci-dessous en pseudo-code et en langage  $C$  (au choix).

1. En supposant que le tableau passé en entrée est trié par ordre croissant, que renvoie exactement cette fonction (sous quel nom connaissez-vous cet algorithme)?
2. Donner une majoration asymptotique, en pire cas, du temps d'exécution de *MaFonction* en fonction de la taille  $n$  du tableau en entrée. Démontrer ce résultat dans les grandes lignes (on pourra se contenter de raisonner sur les cas  $n = 2^k - 1$  pour  $k \geq 0$  entier).

---

**Fonction**  $\text{MaFonction}(T, c)$ 

---

**Entrées :** Un tableau croissant d'entiers  $T[0..n - 1]$  de taille  $\text{Taille}(T) = n$  et un entier  $c$ .

**Sorties :** Une case du tableau  $T$  ou bien une case vide.

```
si  $\text{Taille}(T) > 0$  alors  
   $m = \lfloor \text{Taille}(T)/2 \rfloor$  (partie entière inférieure);  
  si  $c = T[m]$  alors  
    retourner  $T[m]$ ;  
  sinon  
    si  $c < T[m]$  alors  
       $T' =$  le sous tableau  $T[0..m - 1]$ ;  
      retourner  $\text{MaFonction}(T', c)$ ;  
    sinon  
       $T'' =$  le sous tableau  $T[m + 1..\text{Taille}(T) - 1]$ ;  
      retourner  $\text{MaFonction}(T'', c)$ ;  
sinon  
  retourner case vide;
```

---

```
int * mafonction(int T[], int taille, int c) {  
  int milieu;  
  if (taille > 0) {  
    milieu = taille / 2;  
    if (c == T[milieu]) {  
      return &(T[milieu]);  
    }  
    else {  
      if (c < T[milieu]) {  
        return mafonction(T, milieu, c);  
      }  
      else { /* On a c > T[milieu] */  
        return mafonction(T + milieu + 1, taille - milieu - 1, c);  
      }  
    }  
  }  
  return NULL;  
}
```

Figure 6.1 – mafonction en langage C

**Exercice 41.**

Montrer que dans un arbre binaire, si la profondeur maximale des feuilles est  $k$  alors

*le nombre de feuilles est au plus  $2^k - 1$ .*

## 6.3 Exercices du troisième chapitre

### Exercice 42 (Piles).

On forme une nouvelle pile en empilant successivement 1, 2, 3 puis on dépile deux éléments. Quel élément reste-t-il dans la pile ? (Facile)

 0,5 pt  
3 min

### Exercice 43 (Déplacement de pile).

On se donne trois piles  $P_1$ ,  $P_2$  et  $P_3$ . La pile  $P_1$  contient des nombres entiers positifs. Les piles  $P_2$  et  $P_3$  sont initialement vides. En n'utilisant que ces trois piles :

1. Écrire un algorithme pour déplacer les entiers de  $P_1$  dans  $P_2$  de façon à avoir dans  $P_2$  tous les nombres pairs au dessus des nombres impairs.
2. Écrire un algorithme pour copier dans  $P_2$  les nombres pairs contenus dans  $P_1$ . Le contenu de  $P_1$  après exécution de l'algorithme doit être identique à celui avant exécution. Les nombres pairs doivent être dans  $P_2$  dans l'ordre où ils apparaissent dans  $P_1$ .

 1 pt  
6 min

 1 pt  
6 min

### Exercice 44 (Test de bon parenthésage).

Soit une expression mathématique dont les éléments appartiennent à l'alphabet suivant :

$$\mathcal{A} = \{0, \dots, 9, +, -, *, /, (, ), [, ]\}.$$

Écrire un algorithme qui, à l'aide d'une unique pile d'entiers, vérifie la validité des parenthèses et des crochets contenus dans l'expression. On supposera que l'expression est donnée sous forme d'une chaîne de caractères terminée par un zéro. L'algorithme retournera 0 si l'expression est correcte ou  $-1$  si l'expression est incorrecte.

### Exercice 45 (Calculatrice postfixe).

On se propose de réaliser une calculatrice évaluant les expressions en notation postfixe. L'alphabet utilisé est le suivant :  $\mathcal{A} = \{0, \dots, 9, +, -, *, /\}$  (l'opérateur  $-$  est ici binaire). Pour un opérateur  $n$ -aire  $P$  et les opérandes  $O_1, \dots, O_n$ , l'expression, en notation postfixe, associée à  $P$  sera :  $O_1, \dots, O_n P$ . Ainsi, la notation postfixe de l'expression  $(2 * 5) + 6 + (4 * 2)$  sera :  $2 5 * 6 + 4 2 * +$ .

On suppose que l'expression est valide et que les nombres utilisés dans l'expression sont des entiers compris entre 0 et 9. De plus, l'expression est donnée sous forme de chaînes de caractères terminée par un zéro. Par exemple  $(2 * 5) + 6 + (4 * 2)$  sera donnée par la chaîne "25 \* 6 + 42 \* +".

Écrire un algorithme qui évalue une expression postfixe à l'aide d'une pile d'entiers. (On pourra utiliser la fonction `int atoi(char c){return (int)(c-'0');` pour convertir un caractère en entier).

### Exercice 46 (Tri avec files).

On se donne une file d'entiers que l'on voudrait trier avec le plus grand élément en fin de file, selon un tri fusion.

1. On suppose que l'on a trois files  $f_1$ ,  $f_2$  dont les éléments sont déjà triés et une file  $f_3$  initialement vide. Écrire une fonction `Interclasser(f1, f2, f3)` qui range dans l'ordre les éléments de  $f_1$  et  $f_2$  dans  $f_3$ .

2. Proposer une manière d'effectuer le partage des éléments d'une file  $f_1$  en deux files  $f_2$  et  $f_3$  ayant le même nombre d'éléments. L'écrire sous la forme d'une fonction  $Scinder(f_1, f_2, f_3)$  ( $f_2$  et  $f_3$  sont initialement vides).
3. Écrire le tri.
4. Ce tri nécessite-t-il, comme dans les tableaux, une mémoire auxiliaire de l'ordre du nombre d'éléments à trier ?

**Exercice 47** (Tri à l'aide d'une liste de priorité).

Considérons des éléments avec une clé de tri (chaque élément contient une clé, par exemple un prix, et les clés sont deux à deux comparables).

1. En supposant que ces éléments sont initialement dans un tableau et que l'on dispose de la structure de donnée file de priorité, proposer un algorithme de tri utilisant cette structure.
2. Quel serait le temps d'exécution d'un tel tri, en pire cas ?

Rappel : l'insertion et la suppression dans les files de priorité est en  $\Theta(\log N)$  en pire cas (où  $N$  est le nombre d'éléments dans la file).

**Exercice 48** (Files et piles avec des tableaux).

Nous avons évoqué en cours d'amphi une réalisation des piles avec des tableaux.

1. Pourriez-vous rappeler cette réalisation des piles (type des piles, empiler, dépiler, estVide, sommet), sans vous référer à vos notes de cours ? Pour simplifier, considérer qu'il y a au plus NMAX (une constante) éléments dans la pile.

Il est également possible de réaliser les files à l'aide de tableaux.

2. Donner une réalisation des files (estVide, tete, ajouter, retirer) de taille bornée par NMAX.

**Exercice 49** (Listes Chaînées).

Soit la structure **liste** définie en C par :

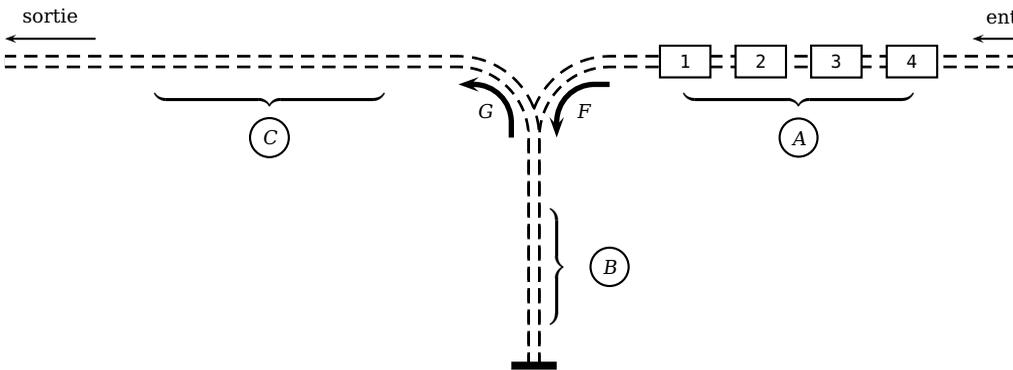
```
typedef struct cellule_s{
    element_t element;
    struct cellule_s *suivant;
} cellule_t;
typedef cellule_t * liste_t;
```

1. Écrire un algorithme récursif (et itératif) qui permet de fusionner deux listes triées dans l'ordre croissant et retourne la liste finale. On pourra utiliser la fonction `cmpListe(liste_t l1, liste_t l2);` qui retourne 1 si le premier élément de  $l1$  est inférieur au premier élément de  $l2$ , 0 s'ils sont égaux et -1 sinon.
2. Écrire un algorithme qui permet d'éliminer toutes les répétitions dans une liste chaînée.

**Exercice 50** (Juin 2007).

On suppose que 4 wagons numérotés de 1 à 4 sont placés en entrée sur le réseau ferroviaire suivant.

4 pt



Les actions possibles sont : *Ajouter(wagon)* qui fait entrer un nouveau wagon sur le réseau (le wagon arrive par l'entrée dans la zone A), *Retirer()* qui fait sortir un wagon (le wagon sort de la zone C par la sortie) ainsi que *F()* qui fait passer un wagon de la zone A à la zone B et *G()* qui fait passer un wagon de la zone B à la zone C. Par exemple, la séquence d'actions : *Ajouter(1)*, *Ajouter(2)*, *Ajouter(3)*, *Ajouter(4)*, *F()*, *F()*, *F()*, *F()*, *G()*, *G()*, *G()*, *G()*, *Retirer()*, *Retirer()*, *Retirer()*, *Retirer()* donnera, par ordre de sorties : 4, 3, 2, 1.

1. Si la séquence de wagons ajoutés est 1, 2, 3, 4 dans cet ordre, peut-on obtenir en sortie les wagons dans l'ordre 2, 4, 3, 1 (expliquer) ?
2. Même question avec six wagons en entrée 1, 2, 3, 4, 5, 6 et la sortie 3, 2, 5, 6, 4, 1 puis la sortie 1, 5, 4, 6, 2, 3.
3. On peut modéliser le réseau comme l'assemblage de trois structures de données élémentaires, une par zone (zone A, zone B, zone C). Quelles sont ces structures de données ? Comment coder les quatre actions possibles à partir des opérations élémentaires des structures de données choisies ?
4. On suppose que l'on a une structure de données réalisant le réseau et ses quatre actions. Comment l'utiliser pour réaliser une pile ? Une file ? (Décrire les opérations d'ajout et de retrait de ces deux structures de données élémentaires à partir des quatre actions).
5. Donner une séquence de wagons en entrée, la plus courte possible, et un ordre de sortie que l'on ne peut pas réaliser (expliquer).

0.5 pt  
4 min

1 pt  
9 min

1 pt  
9 min

1 pt  
9 min

0.5 pt  
4 min

**Exercice 51** (Tours de Hanoi : jouer sans ordinateur).

Le jeu des tours de Hanoi se résout très simplement et élégamment de manière récursive au sens où on peut obtenir la liste des actions à effectuer, pour un  $n$  quelconque. Cependant cela ne permet pas à un joueur humain de résoudre le problème s'il ne dispose pas d'ordinateur : la solution suppose de mémoriser et surtout de reproduire un

tot: 6 pt

grand nombre d'états du jeu ce qui est hors de portée de notre esprit. Nous proposons ici de trouver la solution optimale sous la forme d'une méthode à appliquer pas à pas pour résoudre le jeu.

Un état du jeu est une distribution des disques sur les trois piquets, représentés par des piles,  $a$  (départ),  $b$  (intermédiaire),  $c$  (arrivée). Dans l'état initial la pile  $a$  contient les  $n$  disques, représentés par les entiers de 1 à  $n$ , empilés du plus grand  $n$  (à la base), au plus petit 1 (au sommet). Les deux autres piles sont vides. Il faut déplacer un par un les disques d'une pile vers une autre sans que jamais un disque ne soit posé sur un disque plus petit. L'état final que l'on veut atteindre est celui où la pile d'arrivée contient les  $n$  disques.

L'opération de base est le déplacement d'un disque d'une pile  $p$  vers une autre pile  $q$ .

1. Définir  $\text{Deplacer}(p, q)$  à l'aide des primitives sur les piles.

On sait qu'il y a une unique solution optimale en nombre de déplacements, pour chaque  $n$ . Elle effectue exactement  $2^n - 1$  déplacements.

Ainsi, pour  $n = 2$ , il suffit de 3 déplacements :  $\text{Deplacer}(a, b)$ ,  $\text{Deplacer}(a, c)$ ,  $\text{Deplacer}(b, c)$ .

2. Donner la suite des déplacements pour  $n = 3$ .

Dans un état quelconque, le disque 1 (le plus petit) est toujours au sommet d'une pile  $p$ .

3. Si aucune des autres piles n'est vide, combien exactement de déplacements différents sont possibles ? Combien ont pour origine  $p$  ? Combien ont une autre origine ? Et si l'une des deux autres piles que  $p$  est vide ?
4. Si on déplace un disque est-il intéressant de le déplacer à nouveau le coup suivant ?
5. Quel disque est déplacé exactement un coup sur deux ? Combien de fois est-il déplacé en tout, en fonction de  $n$  ?

On admet la propriété suivante. Lors d'un déplacement du disque 1 celui-ci ne revient jamais sur la pile qu'il occupait avant son déplacement précédent. On en déduit qu'il y a deux possibilités pour les déplacements du disque 1 :

6. soit il occupe tour à tour les piles  $a, b, \dots$  (compléter, sur votre copie) soit il occupe tour à tour les piles  $a, c, \dots$  (compléter).
7. En raisonnant en arithmétique modulo trois, trouver une condition sur  $n$  permettant de déterminer exactement quelle sera la séquence des déplacements du disque 1.
8. Dédurre de ce qui précède les trois déplacements qui suivent celui-ci ( $n$  vaut 6) :

$$\left| \begin{array}{c|c|c} 5 & 2 & 1 \\ \hline 6 & 3 & 4 \\ \hline a & b & c \end{array} \right| \xrightarrow{\text{Deplacer}(b, a)} \left| \begin{array}{c|c|c} 2 & & \\ \hline 5 & & 1 \\ \hline 6 & 3 & 4 \\ \hline a & b & c \end{array} \right|$$

**Exercice 52** (Réussite patience sort (partiel 2007)).

On se donne un paquet  $\sigma$  de  $N$  cartes sur lesquelles sont inscrites des valeurs deux à deux comparables. Pour simplifier, on considérera que ces valeurs sont les entiers de 0 à  $N - 1$ , dans le désordre mais sans répétition.

On fait une réussite de la manière suivante :

- prendre la première carte du paquet et la poser devant soi, à sa gauche, inscription au dessus (de manière à pouvoir voir sa valeur).
- Poser tour à tour les cartes suivantes sur la table, inscription au dessus, en formant des piles de cartes. Pour chaque nouvelle carte  $x$  on peut :
  - soit la poser sur une carte déjà posée sur la pile  $y$ , à la condition que la valeur de  $x$  soit inférieure à la valeur de  $y$  ;
  - soit commencer une nouvelle pile de carte en la posant à droite de toutes les piles existantes.
- On s'arrête lorsqu'il n'y a plus de cartes dans le paquet.

Le but du jeu est de finir avec un nombre minimum de piles de cartes devant soi.

On emploie la stratégie qui consiste à placer une nouvelle carte toujours le plus à gauche possible. Par exemple, si le paquet de cartes contient au départ la suite de cartes :

$$\sigma = 6, 1, 5, 0, 7, 2, 9, 4, 3, 8$$

alors les piles de cartes seront successivement comme ceci (en gras le dernier élément empilé) :

<b>6</b>	<b>1</b> 6	1 6 <b>5</b>	<b>0</b> 1 6 <b>5</b>	0 1 6 <b>5 7</b>	0 <b>1 2</b> 6 <b>5 7</b>	0 1 2 6 <b>5 7 9</b>	0 1 2 <b>4</b> 6 <b>5 7 9</b>	0 <b>3</b> 1 2 4 6 <b>5 7 9</b>	0 3 1 2 4 <b>8</b> 6 <b>5 7 9</b>
----------	---------------	-----------------	-----------------------------	------------------------	---------------------------------	----------------------------	-------------------------------------	---------------------------------------	---

Le nombre de piles est alors 4.

On va montrer que la stratégie du plus à gauche minimise le nombre de piles.

On suppose que l'on a écrit l'algorithme Réussite( $\sigma$ ) qui prend  $\sigma$  en entrée (comme un tableau d'éléments) et rend les  $p$  piles obtenues par la stratégie du plus à gauche. Cet algorithme rend son résultat sous la forme d'un tableau  $T$  de  $p$  piles non vides (la première pile est  $T[0]$ , la dernière  $T[p - 1]$ ).

1. Le nombre de piles dépend de la suite  $\sigma$  des cartes du paquet de départ. Donner une suite de  $N$  cartes réalisant le meilleur cas, c'est à dire donnant le nombre minimum de piles et une suite de  $N$  cartes réalisant le pire cas, c'est à dire donnant le nombre maximum de piles (pour l'algorithme Réussite( $\sigma$ )).

1 pt  
9 min

On s'intéresse maintenant à l'écriture de l'algorithme Réussite( $\sigma$ ). On dispose des fonctions standards sur les piles :

- EstVide( $P$ ) qui rend 1 si la pile  $P$  est vide et 0 sinon.
- Tête( $P$ ) qui rend la valeur de l'élément du dessus de la pile  $P$  sans dépiler cet élément.
- Empiler( $e, P$ ) qui empile l'élément  $e$  sur la pile  $P$ .
- Dépiler( $P$ ) qui dépile l'élément du dessus de la pile  $P$  et rend sa valeur.

Au départ de l'algorithme Réussite( $\sigma$ ) on dispose d'un tableau  $T$  de piles suffisamment grand, et qui ne contient que des piles vides ( $p$  vaut 0). Au cours de l'exécution de l'algorithme Réussite( $\sigma$ ), on empile des éléments de  $\sigma$  dans les piles de  $T$ . Pour poser une nouvelle carte  $x$  il faut comparer la valeur de  $x$  avec les valeurs des têtes de piles.

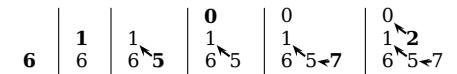
- Quelle méthode peut-on utiliser pour chercher la place de  $x$  en limitant le nombre de comparaisons ? S'il y a  $p$  piles combien fera-t-on au plus de comparaisons pour insérer une nouvelle carte  $x$  (donner un majorant asymptotique) ? En déduire que Réussite( $\sigma$ ) peut être écrite de manière à faire  $O(N \log N)$  comparaisons.

Une suite extraite de  $\sigma$  est une suite formée d'éléments de  $\sigma$  pris dans le même ordre qu'ils apparaissent dans  $\sigma$ , sans forcément choisir des éléments successifs. On considère les suites extraites croissantes de  $\sigma$ . Par exemple, 1, 5, 7, 8 et 0, 2, 3 sont des suites extraites croissantes de  $\sigma = 6, 1, 5, 0, 7, 2, 9, 4, 3, 8$ . On note  $l(\sigma)$  la longueur maximum des suites extraites croissantes de  $\sigma$ . Dans notre exemple  $l(\sigma) = 4$ .

- Montrer que  $l(\sigma)$  minore le nombre de piles obtenues quel que soit la stratégie. (Indication : il faut montrer que si  $a_1 < \dots < a_k$  est une suite extraite de  $\sigma$  alors il faut au moins  $k$  piles).

On veut montrer que si Réussite( $\sigma$ ) forme  $p$  piles, alors il existe une suite extraite croissante de  $\sigma$  de longueur  $p$ . À chaque fois qu'on pose une nouvelle carte  $x$  sur une pile, sauf si on la pose sur la première pile, on dessine une flèche de  $x$  vers la carte du dessus de la pile à sa gauche.

Voici ce que ça donne sur l'exemple, pour les six premières insertions.



- Compléter l'exemple précédent (sur votre copie).
- En prenant un élément d'une pile et en suivant les flèches on obtient une suite d'éléments de  $\sigma$  (en ordre inverse). Par exemple,  $1 \leftarrow 5 \leftarrow 7$ . À quoi correspond une telle suite par rapport à  $\sigma$  ? (Justifier.)
- Montrer que si Réussite( $\sigma$ ) a formé  $p$  piles, alors il existe une suite extraite croissante de  $\sigma$  de longueur  $p$ .
- Comment peut-on déduire de la question 3 et de la question précédente que  $p = l(\sigma)$  ? Que la stratégie du plus à gauche est optimale ?

Ainsi la stratégie du plus à gauche, qui peut s'écrire comme un algorithme Réussite( $\sigma$ ), permet de calculer la taille de la plus grande suite extraite croissante en  $O(N \log N)$  comparaisons.

On veut maintenant écrire un algorithme RassemblerPiles( $T$ ) qui prend en entrée le tableau  $T$  rendu par Réussite( $T$ ) et rend le tableau trié des éléments de  $\sigma$ . Dans chacune des  $p$  piles les éléments sont triés (le plus petit en haut de la pile), il suffit donc d'interclasser les  $p$  piles d'éléments. On peut rendre l'interclassement plus efficace, en observant qu'au départ les éléments du dessus des piles sont également bien

ordonnés :

$$\text{Tête}(T[0]) < \text{Tête}(T[1]) < \dots < \text{Tête}(T[p - 1]). \quad (6.6)$$

Mais si on retire un élément du dessus d'une des piles, la propriété (6.6) n'est plus forcément vraie.

8. Pouvez-vous donner une suite  $\sigma$  la plus courte possible, telle que depiler un élément de la première pile rend la suite des têtes de piles non croissante ?  0,5 pt  
4 min
9. Après un `Dépiler(T[0])` tel que  $T[0]$  contient encore un élément, que faut-il faire pour réordonner les piles de manière à avoir de nouveau la propriété (6.6) ?  0,5 pt  
4 min
10. Écrire l'algorithme `RassemblerPiles(T)`, en pseudo-code ou en C.  2 pt  
18 min

Au besoin on pourra faire appel à une fonction `Echanger(T, i, j)` qui échange dans le tableau  $T$ , la pile  $T[i]$  avec la pile  $T[j]$ . En C, on pourra considérer tous les arguments auxiliaires éventuellement nécessaires. Par exemple on pourra considérer que le premier argument est le tableau de piles, que  $p$  est donné comme second argument, que le tableau dans lequel ranger le résultat est donné comme troisième argument et que  $N$  est donné comme quatrième argument : `B(pile_t T[], int nb_piles, elements_t res[], int nb_elts)`

## 6.4 Exercices du quatrième chapitre

**Exercice 53** (Tas, septembre 2007).

Dans un tas max où se trouve l'élément maximum ? Où peut-on trouver l'élément minimum ? (Facile)

**Exercice 54** (Insertion / suppression tas, septembre 2007).

Former un tas max en insérant successivement et dans cet ordre les éléments : 10, 7, 3, 9, 11, 5, 6, 4 (répondre en représentant le tas obtenu). Supprimer l'élément maximum (répondre en représentant le nouveau tas).

**Exercice 55** (Insertion / suppression).

On se donne le tas max (ou maximier) de la figure 6.2. En utilisant les algorithmes vus en cours :

1. Insérer un élément de clé 17 dans ce tas. Répondre en représentant le nouveau tas.
2. En repartant du tas initial, retirer l'élément de clé maximale. Répondre en représentant le nouveau tas.

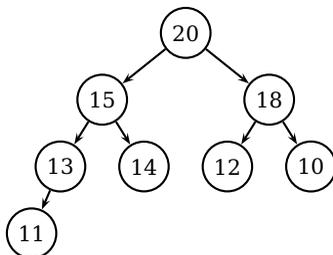


Figure 6.2 – Tas

**Exercice 56** (Indexation).

Dans le cours nous avons vu qu'un tas max est un arbre quasi-parfait ayant de plus la propriété de dominance des tas : le parent est toujours plus grand que ses fils. Nous avons également vu qu'un arbre quasi-parfait est efficacement représenté par un tableau en mettant la racine à la première case.

1. Représenter sous forme d'arbre quasi-parfait le tableau suivant :

20	19	10	7	15	12	3	6	5	1
----	----	----	---	----	----	---	---	---	---

Est-ce un tas ?

pour mémoriser un tas d'entiers, on se donne un tableau d'entiers `tab` suffisamment grand (on suppose que l'allocation mémoire initiale du tableau est suffisante pour l'usage qui en sera fait) et un entier `taille` qui correspondra au nombre d'éléments du tas actuellement stockés dans le tableau.

Un nœud du tas sera représenté par un indice du tableau. Étant donné un nœud  $i$  nous voulons pouvoir obtenir son parent à l'indice  $\text{parent}(i)$ , son fils gauche à l'indice  $\text{gauche}(i)$  et son fils droit à l'indice  $\text{droite}(i)$ .

2. En vous aidant de l'exemple, proposer des fonctions pour le calcul de ces indices.

Pour l'insertion d'un élément et la suppression de l'élément maximum nous avons vu en cours qu'il faut faire appel respectivement à deux procédures `maintien_haut()` et `maintien_bas()`. Elles prennent en argument le tas (le tableau et la taille) et l'indice à partir duquel effectuer le maintien.

3. Écrire ces procédures `maintien_haut()` et `maintien_bas()` de manière récursive.
4. Écrire une fonction `estvide()` qui prend le tas (tableau et taille) en argument et renvoie vrai (1) si le tas est vide et faux (0) sinon.
5. Écrire la fonction `maximum()` qui renvoie l'élément (ici un entier) à la racine du tas.
6. Écrire la procédure `retirer_maximum()` qui retire l'élément maximum du tas (et reforme un tas) ainsi que la procédure `insérer()` qui insère un élément (un entier  $e$  passé en argument) dans le tas.
7. Proposer un algorithme de tri utilisant les tas (on triera un tableau  $t$  de taille  $n$  passé en argument en utilisant un tas stocké dans un tableau de  $N > n$  cases).

### Exercice 57 (Tri par tas).

Simuler étape par étape l'exécution d'un tri par tas sur le tableau 12, 15, 26, 30, 10, 80, 29, 31, 23, 45 en représentant les tas successifs obtenus :

1. en supposant que le tri consiste en (i) la formation du tas par ajouts successifs des éléments du tableau dans le tas (ii) puis à leurs retraits.
2. Puis en supposant que pour former le tas, on utilise plutôt l'algorithme qui consiste en rétablir la propriété de domination, à partir de l'arbre quasi-parfait correspondant au tableau, par application de `maintienBas` sur chacun des nœuds internes, successivement, par indices décroissants (du nœud interne d'indice  $\text{parent}(N-1)$  à la racine, d'indice 0).

### Exercice 58 (Septembre 2007).

On se donne un tableau  $T$  de  $p$  piles d'entiers. Autrement dit les éléments de  $T$  sont des piles et chaque pile contient des entiers. On suppose que  $T$  est tel que :

- Aucune pile n'est vide et il y a en tout  $N$  entiers répartis dans les piles (donc  $p \leq N$ )
- les sommets des piles vont décroissants :

$$\text{Sommet}(T[0]) \geq \text{Sommet}(T[1]) \geq \dots \geq \text{Sommet}(T[p-1])$$

- Dans chaque pile les entiers sont ordonnés du plus grand (au sommet) au plus petit.

 4 pt  
24 min

On souhaite réaliser l'interclassement des éléments des  $p$  piles de manière à obtenir un nouveau tableau contenant les  $N$  éléments dans l'ordre croissant.

Pouvez vous donner un algorithme en  $O(N \log N)$  comparaisons pour ce problème ? (Justifier) Vous pouvez utiliser des algorithmes vus en cours et les résultats de complexité sur ces algorithmes.

Indication : le tableau  $T$  est un tas max dont les éléments sont des piles et où les clés sont les entiers au sommet des piles.

### Exercice 59 (Indexation).

Dans le cours nous avons vu qu'un tas max est un arbre quasi-parfait ayant de plus la propriété de dominance des tas : le parent est toujours plus grand que ses fils. Nous avons également vu qu'un arbre quasi-parfait est efficacement représenté par un tableau contenant les éléments donnés dans l'ordre du parcours en largeur de l'arbre. Ainsi la racine est à l'indice  $\text{racine}() = 0$  et le dernier élément à l'indice  $\text{dernier}() = N - 1$  (où  $N$  est le nombre total d'éléments). Lorsqu'il existe, le parent de l'élément à l'indice  $i$  est à l'indice  $\text{parent}(i) = \lfloor \frac{i-1}{2} \rfloor$ . Lorsqu'ils existent, le fils gauche de l'élément d'indice  $i$  est à l'indice  $\text{gauche}(i) = 2i+1$  et le fils droit à l'indice  $\text{droite}(i) = 2i+2$ . On introduit également les fonctions  $\text{suivant}(i) = i+1$  et  $\text{précédant}(i) = i-1$  qui permettent de se déplacer respectivement à l'élément suivant et à l'élément précédent dans l'ordre du parcours en largeur de l'arbre qui est ici le même que l'ordre des indices du tableau.

1. On décide de placer la racine à un indice  $r$  du tableau, tout en conservant l'ordre direct de parcours de l'arbre (le dernier élément sera à l'indice  $N+r-1$ ). Comment faut-il modifier les fonctions précédentes ?
2. Même question mais cette fois-ci, on change également l'ordre des éléments : les éléments sont maintenant rangés dans le tableau dans l'ordre inverse du parcours en profondeur (le dernier élément est à l'indice  $r-N+1$ ).

### Exercice 60 (Biprocasseur).

Nous utilisons ici les files de priorités pour gérer l'ordonnancement de processus sur une machine biprocasseur. Les processus arrivent alternativement aux fils d'attente  $f_0$  et  $f_1$  des deux processeurs  $P_0$  et  $P_1$ . Chaque processus possède une charge (qui représente un temps de travail) et une valeur d'attente. Plus l'attente est faible plus le processus est prioritaire.

Chaque processeur traite en un tour le processus de plus haute priorité (attente la plus faible) puis diminue sa charge d'une constante  $c$  et augmente son attente de  $c$ . Si la charge d'un processus est nulle il est retiré de sa file d'attente.

Le traitement s'effectue en parallèle sur les deux processeurs tour par tour. À la fin d'un tour de traitement, si les deux files d'attente ont une différence en nombre de processus supérieure ou égale à deux, il faut rééquilibrer en faisant passer un processus d'une file à l'autre. On choisit pour cela le processus ayant l'attente la plus faible de la plus longue file et on le fait passer dans la file la plus courte.

Lorsque deux processus d'une même file ont même temps d'attente c'est le premier arrivé qui a la priorité (quelle que soit la file d'arrivée).

1. Pour  $c = 1$ , simuler l'exécution du biprocesseur sur la liste de processus (*pid*, *charge*, *attente*) suivante :  $(p_1, 2, 0), (p_2, 3, 1), (p_3, 1, 2), (p_4, 1, 2), (p_5, 5, 2), (p_6, 1, 4), (p_7, 1, 5)$   
Représenter les deux files sous forme d'arbres (tas min) après chaque tour.

On suppose qu'il y a au plus  $N$  processus en tout dans les deux files d'attente, ce qui nous permet de représenter ces deux files de priorité comme deux tas (min) dans un même tableau. Les définitions de types seront les suivantes :

```
typedef struct {
    pid_t    id;
    int     charge;
    int     attente;
} processus_t;

typedef struct {
    processus_t t[N];
    int     card0;
    int     card1;
} * bifile_t;
```

La racine du tas  $f_0$  est à l'indice 0 et la racine du tas  $f_1$  est à l'indice  $N - 1$ . Les éléments du tableau sont les processus.

2. Déterminer toutes les fonctions nécessaires à l'écriture du programme de simulation qui prendra en entrée un tableau de  $N$  processus à exécuter.
3. Écrire ces fonctions.

### Exercice 61.

Combien y a-t'il de formes d'arbres binaires différents à (respectivement) 1, 2, 3, 4 nœuds ? Les dessiner (sans donner de contenu aux nœuds).

Pour  $n$  fixé quelconque donner un exemple d'arbre binaire de hauteur majorée par  $\log n + 1$ , et un exemple d'arbre binaire de hauteur  $n$ .

### Exercice 62.

Écrire une fonction *Taille*( $x$ ) prenant un arbre binaire et rendant le nombre de ses éléments.

### Exercice 63.

Écrire une fonction *Hauteur*( $x$ ) prenant un arbre binaire et rendant sa hauteur, c'est à dire le nombre d'éléments contenus dans la plus longue branche.

### Exercice 64 (Parcours infixe itératif).

Le but de cet exercice est de donner un algorithme non récursif qui effectue un parcours infixe.

1. Rappeler l'algorithme récursif du parcours infixe d'un arbre binaire de recherche. Donner un majorant serré de l'espace mémoire utilisé par le parcours infixe récursif.
2. Il existe une solution simple qui fait appel à une pile comme structure de donnée auxiliaire (et n'utilise jamais de référence à un parent) et une solution plus compliquée, mais plus élégante et en espace constant, qui n'utilise aucune pile et teste des égalités de pointeurs. Donner les deux solutions.

**Exercice 65** (Insertion / suppression ABR).

Former un arbre binaire de recherche en insérant successivement et dans cet ordre les éléments : 10, 7, 3, 9, 11, 5, 6, 4, 8 (répondre en représentant l'arbre obtenu). Supprimer l'élément 7. (répondre en représentant le nouvel arbre. Il y a deux réponses correctes possibles, selon la variante choisie pour l'algorithme de suppression, n'en donner qu'une seule).

**Exercice 66** (Parcours infixe itératif).

Le but de cet exercice est de donner un algorithme non récursif qui effectue un parcours infixe.

1. Rappeler l'algorithme récursif du parcours infixe d'un arbre binaire de recherche. Donner un majorant serré de l'espace mémoire utilisé par le parcours infixe récursif.
2. Il existe une solution simple qui fait appel à une pile comme structure de donnée auxiliaire (et n'utilise jamais de référence à un parent) et une solution plus compliquée, mais plus élégante et en espace constant, qui n'utilise aucune pile et teste des égalités de pointeurs. Donner les deux solutions.

**Exercice 67.**

Écrire une fonction *Hauteur*( $x$ ) prenant un arbre binaire et rendant sa hauteur, c'est à dire le nombre d'éléments contenus dans la plus longue branche. Vous pouvez faire appel à des fonctions *Parent*( $x$ ), *Gauche*( $x$ ), *Droite*( $x$ ).

Rappel. un nœud d'un arbre binaire contient : un élément, une référence vers le nœud parent, une référence vers le nœud fils gauche et une référence vers le nœud fils droit. S'il n'y a pas de nœud parent, fils gauche ou fils droit, les références prennent la valeur spéciale « nulle ». L'arbre est donné par une référence vers son nœud racine. Type en C :

```
typedef struct noeud_s {
    struct noeud_s * parent;
    struct noeud_s * gauche;
    struct noeud_s * droite;
    element_t e;
} * ab_t;
```

**Exercice 68.**

Dans les deux exemples d'arbres binaires de recherche de la figure 6.3 :

1. où peut-on insérer un élément de clé 13 ?
2. comment peut-on supprimer l'élément de clé 14 ?

Les rotations des arbres binaires sont données dans la figure 6.4. Elles se lisent comme ceci. Une rotation à droite de centre  $x$  consiste en : prendre l'élément  $a$  de  $x$ , le sous-arbre droite  $E$  de  $x$ , la racine  $y$  du sous-arbre gauche de  $x$ , l'élément  $b$  contenu dans  $y$ , et  $C$  et  $D$  les deux sous-arbres gauche et droite de  $y$  ; remplacer  $a$  par  $b$  dans  $x$ , remplacer le sous-arbre gauche de  $x$  par  $C$ , et le sous-arbre droite de  $x$  par un arbre

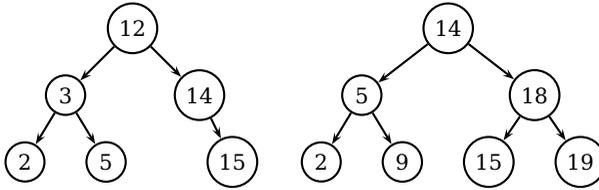


Figure 6.3 – Deux arbres binaires de recherche

de racine contenant  $a$  (on utilise  $y$  pour des raisons d’implantation) et dont les sous-arbres gauche et droite sont respectivement  $D$  et  $E$ . La rotation à gauche de centre  $x$  est l’opération réciproque.

Avec cette manière d’écrire les rotations la référence de  $x$  à son parent n’a pas besoin d’être mise à jour. Il est assez standard de trouver une version qui échange la place de  $x$  et de  $y$  plutôt que d’échanger leurs éléments mais nous ne l’utilisons pas ici.

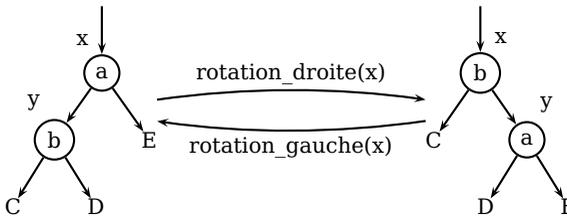


Figure 6.4 – Rotations gauche et droite. Dans cette version les éléments  $a$  et  $b$  sont échangés entre les nœuds  $x$  et  $y$  mais  $x$  garde sa place dans l’arbre qui le contient.

3. Sur le premier exemple de la figure 6.3, faire : une rotation à droite de centre le nœud de clé 3 ; puis une rotation à gauche de centre le nœud de clé 14.
4. Sur le second exemple de la figure 6.3, à l’aide de rotations dont vous préciserez le sens et le centre, amener le nœud de clé 9 à la racine.
5. Démontrer que toute rotation préserve la propriété d’être un arbre de recherche.
6. Écrire l’algorithme de rotation à droite en C.
7. Écrire un algorithme permettant de remonter à la racine n’importe quel nœud d’un arbre binaire de recherche, à l’aide de rotations.

**Exercice 69** (Insertion / suppression ABR).

Former un arbre binaire de recherche en insérant successivement et dans cet ordre les éléments : 10, 7, 3, 9, 11, 5, 6, 4, 8 (répondre en représentant l’arbre obtenu). Supprimer l’élément 7. (répondre en représentant le nouvel arbre. Il y a deux réponses



1,5 pt  
9 min

correctes possibles, selon la variante choisie pour l'algorithme de suppression, n'en donner qu'une seule).

**Exercice 70** (À la fois ABR et tas?).

Un tas est nécessairement un arbre binaire quasi-parfait. Est-il toujours possible d'organiser un ensemble de  $n$  clés ( $n$  quelconque) en tas max de manière à ce que cet arbre binaire soit aussi un arbre binaire de recherche? (Justifier par un raisonnement ou un contre-exemple).

**Exercice 71.**

On peut afficher les éléments d'un ABR de taille  $n$  en ordre trié en un temps  $O(n)$ .

1. Expliquer comment et argumenter sur le temps d'exécution.
2. Est-ce que la propriété de tas permet d'afficher en ordre trié et en temps  $O(n)$  les éléments d'un tas de taille  $n$ ? Argumenter. Indication : on peut planter (former) un tas de  $n$  éléments en un temps  $\Theta(n)$ .



1,5 pt  
9 min

tot: 9 pt

**Problème du rang juin 2007 (quatre exercices)**

Dans cette partie on s'intéresse au problème suivant : étant donné un ensemble  $A$  de  $n$  éléments deux à deux comparables, déterminer quel est l'élément de rang  $k$  (le  $k$ -ième plus petit élément), c'est à dire l'élément  $x \in A$  tel que exactement  $k - 1$  éléments de  $A$  sont plus petits que  $x$ . Bien entendu, on peut supposer que  $k$  est choisi tel que  $1 \leq k \leq n$ . On pourra considérer que les éléments de  $A$  sont distincts (la comparaison ne les déclare jamais égaux). Si on a par exemple les éléments 23, 62, 67, 56, 34, 90, 17 ( $n$  vaut 7) alors l'élément de rang 3 est 34.

Les trois exercices suivants portent sur l'écriture d'un algorithme réalisant la fonction de recherche de l'élément de rang  $k$  dans un ensemble  $A$ , Sélection( $S, k$ ), où  $S$  est la structure de donnée contenant les éléments de  $A$ .

**Rang dans un tableau**

On suppose que les éléments de  $A$  sont donnés en entrée dans un tableau  $T$  non trié de  $n$  éléments.

**Exercice 72.**

Quel est le moyen le plus simple (3-4 lignes de pseudo-code ou de C) de réaliser la fonction Sélection( $T, k$ ) à partir d'algorithmes du cours? Quelle borne asymptotique minimale obtiendra-t-on pour le temps d'exécution de cette fonction en fonction de  $n$ ? (Préciser le ou les algorithmes du cours que vous utilisez).

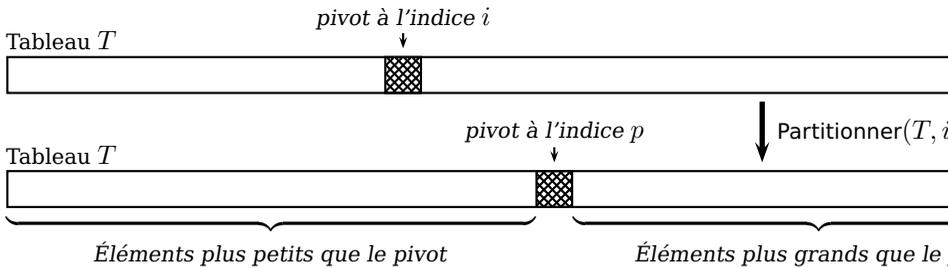


1 pt  
9 min

**Rappel sur le tri rapide.** On rappelle le fonctionnement du tri rapide (*quick-sort*), algorithme permettant de trier un tableau d'éléments deux à deux comparables (le résultat, c'est à dire le tableau contenant les éléments dans l'ordre

est rendu en place). Le principe de fonctionnement est le suivant. Si le tableau contient un ou zéro élément, il n'y a rien à faire.

**Partition** Si le tableau contient plus d'un élément, on choisit (au hasard ou bien en prenant le premier élément du tableau) un élément du tableau, d'indice  $i$ , appelé *pivot* et on partitionne le tableau autour de cet élément. Plus précisément, la partition fait en sorte que, après partition, le pivot est à l'indice  $p$ , les éléments plus petits que le pivot sont (dans le désordre) aux indices inférieurs à  $p$  et les éléments plus grands que le pivot sont (dans le désordre) aux indices supérieurs à  $p$ . Le pivot est donc à sa place définitive.



**Appels récursifs** On relance le tri sur chacun des deux sous-tableaux obtenus par partition : le tableau des éléments d'indices inférieurs à  $p$  et le tableau des éléments d'indices supérieurs à  $p$ .

**Exercice 73.**

Dans cet exercice, on va mettre en œuvre un algorithme inspiré du tri rapide pour réaliser la fonction  $Sélection(T, k)$ . L'idée est qu'il n'est pas besoin de trier tout le tableau pour trouver l'élément de rang  $k$ .

On suppose que les indices du tableau commencent à 1. La remarque importante est que le pivot est l'élément de rang  $p$  ( $p + 1$  si les indices commencent à zéro). Si après partition on trouve un  $p$  égal à  $k$ , alors l'élément de rang  $k$  est le pivot.

On suppose que l'on a une fonction  $Partitionner(T, i)$  qui effectue la partition autour de l'élément d'indice  $i$  (pivot) et renvoie l'indice  $p$  du pivot après partition.

1. Après partition autour d'un pivot où chercher l'élément de rang  $k$  lorsque  $k < p$ ? Et lorsque  $k > p$ ? Décrire simplement le principe d'un algorithme récursif, basé sur la partition, réalisant  $Sélection(T, k)$ . (A t-on besoin de faire deux appels récursifs comme dans le tri rapide?) 1.5 pt  
13 min
2. Écrire en pseudo-code ou en C, l'algorithme  $Sélection(T, k)$  sous forme itérative. 2 pt  
18 min
3. Écrire l'algorithme de partitionnement  $Partitionner(T, i)$ . 1.5 pt  
13 min

## Rang dans un arbre binaire de recherche avec comptage des descendants

On enrichit la structure de données *arbre binaire de recherche* (ABR) en ajoutant à chaque nœud  $x$  un entier  $\text{Total}(x)$  donnant le nombre d'éléments stockés dans le sous-arbre dont le nœud  $x$  est racine, l'élément stocké dans  $x$  inclus. Exemple figure 6.5.

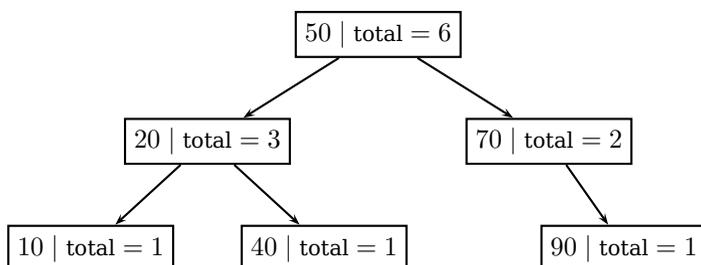


Figure 6.5 – Un exemple d'arbre binaire de recherche avec comptage des descendants

Dans la suite, on suppose que les éléments de  $A$  sont stockés dans un ABR enrichie par ce comptage, de racine  $r$ .

### Exercice 74.

Écrire un algorithme (C ou pseudo-code) réalisant la fonction  $\text{Sélection}(r, k)$ . Donner en fonction de la hauteur  $h$  de l'arbre, un majorant asymptotique de son temps d'exécution.

1.5 pt  
13 min

En guise d'aide, on rappelle, en pseudo-code, l'algorithme de recherche dans un ABR.

---

**Fonction Rechercher( $r, c$ )**

---

**Entrées :** Le nœud racine  $r$  d'un arbre binaire de recherche et une clé  $c$ .

**Sorties :** Le nœud de l'arbre contenant l'élément de clé  $c$  s'il en existe un, ou le nœud vide  $N$  sinon.

**si** EstVide( $r$ ) **alors**

| retourner  $N$  ;

**sinon**

| **si**  $c = Clé(r)$  **alors**

| | retourner  $r$  ;

| **sinon**

| | **si**  $c < Clé(r)$  **alors**

| | | retourner Rechercher(Gauche( $r$ ),  $c$ ) ;

| | **sinon**

| | | retourner Rechercher(Droite( $r$ ),  $c$ ) ;

---

**Exercice 75.**

Pour réaliser les ABRs avec comptage des descendants, il faut adapter les fonctions d'ajout et de suppression d'un élément. Expliquer les modifications à apporter. Et pour les rotations ?

 1.5 pt  
13 min

**Exercice 76.**

Soit un tableau  $t$  de  $n$  entiers tous différents. Quel est le rapport entre :

- l'arbre des appels récursifs de la fonction de tri du quicksort lorsque le pivot est toujours choisi dans la première case et que partitionner conserve dans chaque sous-tableau l'ordre des éléments du tableau d'origine ;
- et les arbres binaires de recherche ?

**Exercice 77.**

Notre but est de montrer que la hauteur d'un arbre rouge noire est logarithmique en son nombre de nœuds.

Rappel. Soit  $x$  un nœud d'un arbre rouge noir. On appelle hauteur noire de  $x$ , notée  $H_n(x)$ , le nombre de nœuds noirs présents dans un chemin descendant de  $x$  (sans l'inclure) vers une feuille de l'arbre.

1. Dans l'arbre rouge noir donné en figure 6.6, que valent  $H_n(30)$ ,  $H_n(20)$ ,  $H_n(35)$ ,  $H_n(50)$  ?

Montrer que, pour un nœud  $x$  quelconque dans un arbre rouge noir, le sous-arbre enraciné à  $x$  contient au moins  $2^{H_n(x)} - 1$  nœuds internes.

En déduire qu'un arbre rouge noir comportant  $n$  nœuds internes a une hauteur au plus égale à  $2 \log(n + 1)$ .

**Exercice 78.**

Est-il possible de colorier tous les nœuds de l'arbre binaire de recherche de la figure 78 pour en faire un arbre rouge noir ?

 1 pt  
9 min

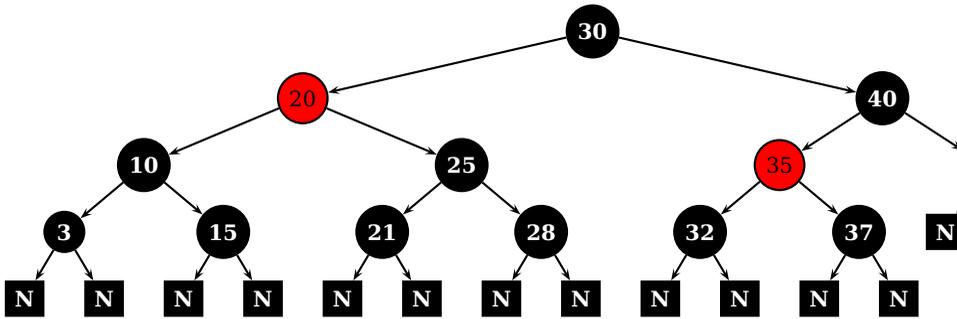
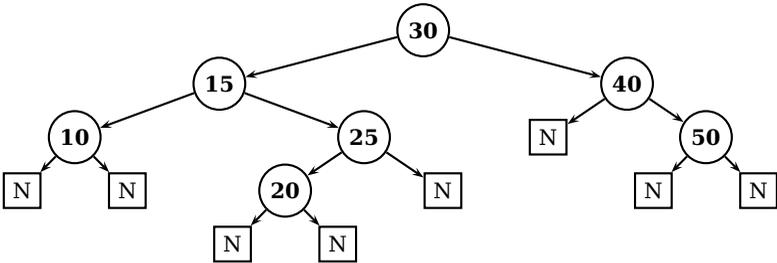


Figure 6.6 – Un exemple d'arbre rouge noir



**Exercice 79.**

Pour chaque arbre de la figure 6.7, dire s'il s'agit d'un arbre rouge noir. Si non, pourquoi ?

1 pt  
6 min

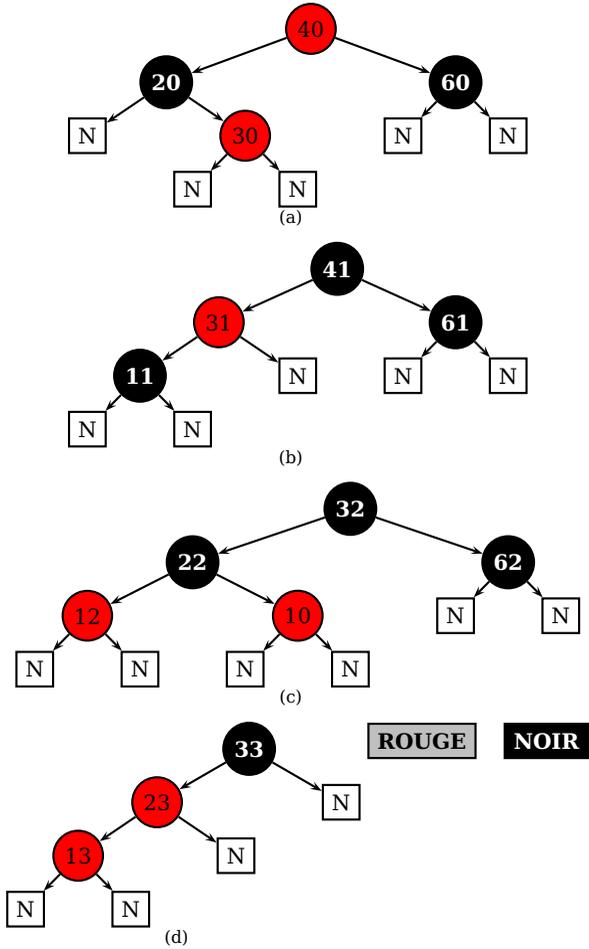


Figure 6.7 – Rouge noir ?

## **Chapitre 7**

# **Correction des exercices**

# 7.1 Premier chapitre

## Correction de l'exercice 1.

1. Factorielle de son argument.

2. `int Fibonacci(n)`

```
{
    if (n < 3) /* cas de base */
    {
        return 1;
    }
    return Fibonacci(n - 1) + Fibonacci(n - 2); /* /\ double appel récursif */
}
```

Vous pouvez en profiter pour dire rapidement que ce genre de doubles appels récursifs prennent du temps en montrant par exemple que pour calculer `Fibonacci(n)` il faut revenir `Fibonacci(n)` fois au cas de base (puisque le résultat est calculé comme une somme  $1 + 1 + \dots + 1$  où les 1 proviennent du cas de base, n'hésitez pas à dessiner un arbre). Et la suite croît très vite, pour  $n = 31$  on est déjà au delà du million.

3. On utilise  $\text{pgcd}(a, b) = a$  si  $b = 0$  et  $\text{pgcd}(a, b) = \text{pgcd}(b, a \bmod b)$ .

```
unsigned int pgcd(unsigned int a, unsigned int b){
    if (a < b) return pgcd(b, a);
    if (b == 0) return a;
    return pgcd(b, a % b);
}
```

4. Pour  $n > 0$ , la fonction `Tata(n)` calcule  $\lfloor \log n \rfloor$  en base 2.

5. L'exécution de cette fonction (appel) provoque une `Segmentation fault` En effet `Morris(1, 0)` lance le calcul de l'expression `Morris(0, Morris(1, 0))` qui induit le calcul préalable de la valeur de `Morris(1, 0)` et ainsi indéfiniment, jusqu'à ce que le programme ne puisse plus lancer de nouveaux sous calculs faute de mémoire disponible à cet effet. (Sur mon système ça s'arrête au bout de  $255 \times 1024$  appels, ça doit être quelque chose comme 8 Mo alloués à la pile, avec 8 mots de 32 bits alloués à chaque appel mais il ne faut pas s'étendre sur la structure de la mémoire d'un processus, les étudiants de MIEF et de maths n'ont pas vu ce genre de choses).

## Correction de l'exercice 2.

Note : c'est le John McCarthy du Lisp, pas celui de la *terreur rouge*.

Prenons 91 comme exemple (ce n'est pas le meilleur choix mais un des plus naturels au regard de l'énoncé) et notons  $f$  la fonction. Comme  $91 < 100$  le résultat de  $f(91)$  sera  $f(f(91 + 11)) = f(f(102))$ . Mais  $f(102) = 92$  donc  $f(91) = f(f(102)) = f(92)$ . Maintenant  $f(92) = f(f(92 + 11)) = f(f(103)) = f(93)$  et

par le même raisonnement  $f(93) = f(94) = \dots = f(99) = f(f(110)) = f(100) = f(f(111)) = f(101) = 91$ . Donc  $f(91) = \dots = f(100) = 91$ . Finalement, pour 90 on a  $f(90) = f(f(101)) = f(91) = 91$ . Voyons ce que ça donne pour 89. On a  $f(89) = f(f(100)) = f(91) = 91$ .

On conjecture que  $f(n)$  vaut 91 pour tout  $n \leq 100$  (y compris les négatifs).

Commençons par prouver plus proprement que  $f(n) = 91$  pour les onze entiers  $n$  compris entre 90 et 100.

*Démonstration.* On a  $f(101) = 91$ . On montre que  $f(n) = f(n + 1)$  pour  $90 \leq n \leq 100$ . Si  $90 \leq n \leq 100$  alors

$$\begin{aligned} f(n) &= f(f(n + 1)) \text{ et } n + 1 \geq 101 \\ &= f(n + 11 - 10) \\ &= f(n + 1) \\ &\dots \\ &= f(101) \end{aligned}$$

On montre alors par récurrence sur  $k$  que lorsque  $90 - k \times 11 \leq n \leq 100 - k \times 11$ ,  $f(n) = 91$ . Ceci démontrera que  $f(n) = 91$  pour ( $n \leq 100$ ).

Le cas de base  $k = 0$  est démontré. Supposons l'assertion vraie pour  $k$  on montre qu'elle l'est encore pour  $k + 1$ . Soit  $n$  un entier dans l'intervalle  $[90 - (k + 1) \times 11, 100 - (k + 1) \times 11]$ . Alors  $n \leq 100$  donc  $f(n) = f(f(n + 1))$ . Mais  $f(n + 1)$  est dans l'intervalle  $[90 - k \times 11, 100 - k \times 11]$  donc par hypothèse de récurrence  $f(n + 1) = 91$  et ainsi  $f(n) = f(91)$  qui est égal à 91 (cas de base). Finalement le résultat est prouvé pour tous les intervalles  $[90 - k \times 11, 100 - k \times 11]$  et leur union  $([90, 100] \cup [79, 89] \cup [68, 78] \cup \dots)$  correspond à l'ensemble des entiers inférieurs à 100.  $\square$

### Correction de l'exercice 3.

1. Facile :

```
deplacerdisque(p1, p2);
deplacerdisque(p1, p3);
deplacerdisque(p2, p3);
```

2. En trois coups de cuillères à pot : on déplace la tour des  $n - 1$  disques du dessus du premier piquet vers le deuxième piquet (en utilisant le troisième comme piquet de travail), puis on déplace le dernier disque du premier au troisième piquet et enfin on déplace à nouveau la tour de  $n - 1$  disques, cette fois ci du deuxième piquet vers le troisième piquet, en utilisant le premier piquet comme piquet de travail.

3. Ce qui précède nous donne immédiatement la structure d'une solution récursive :

```

void deplacertour(unsigned n, piquet_t p1, piquet_t p2, piquet_t p3){
  if (n > 0){
    /* tour(n) = un gros disque D et une tour(n - 1)          */
    deplacertour(n - 1, p1, p3, p2); /* tour(n - 1): p1 -> p2 */
    deplacerdisque(p1, p3);          /* D: 1 -> 3             */
    deplacertour(n - 1, p2, p1, p3); /* tour(n - 1): p2 -> p3 */
  }
}

```

- On fait  $u_n = 2u_{n-1} + 1$  appels avec  $u_0 = 0$ . On pose  $v_n = u_n + 1$  Ce qui donne  $v_n = 2v_{n-1}$  avec  $v_0 = 1$ . On obtient  $v_n = 2^n$  d'où la forme close  $u_n = 2^n - 1$ .
- Par récurrence. Ça l'est pour  $n = 0$ . On suppose que ça l'est pour une tour de  $n - 1$  éléments. Supposons qu'on ait une série de déplacements quelconque qui marche pour une tour de  $n$  éléments. Il faut qu'à un moment  $m$  on puisse déplacer le disque du dessous. On doit donc avoir un piquet vide  $p$  pour y poser ce disque et rien d'autre d'empilé sur le premier piquet (où se trouve le "gros disque"). Le cas où  $p$  est le troisième piquet nous ramène immédiatement à notre algorithme. Dans ce cas, entre le début des déplacements et le moment  $m$  ainsi qu'entre juste après le moment  $m$  et la fin des déplacements on déplace deux fois en entier une tour de taille  $n - 1$ . Notre hypothèse de récurrence établie que pour un seul de ces déplacements complet d'une tour on effectue au moins  $2^{n-1} - 1$  déplacements de disques. En ajoutant le déplacement du gros disque on obtient alors que le nombre total de déplacements de disques est minoré par  $2 \times (2^{n-1} - 1) + 1$  c'est à dire par  $2^n - 1$ . Reste le cas où  $p$  est le second piquet. Dans ce cas, il doit y avoir un moment ultérieur  $m'$  où on effectue le déplacement vers le troisième piquet (le second ne contient alors que le gros disque). On conclue alors comme dans le cas précédent, en remarquant de plus que les étapes entre  $m$  à  $m'$  sont une perte de temps.

## Correction de l'exercice 4.

- Le gain maximum en  $D$  est la valeur en  $D$  plus le maximum entre : le gain maximum en  $A$ ; le gain maximum en  $B$  et le gain maximum en  $A$ . On peut oublier de considérer le gain en  $A$ , puisqu'avec des valeurs non négatives sur chaque case, c'est toujours au moins aussi intéressant, partant de  $A$ , de passer par  $B$  ou  $C$  pour aller en  $D$  plutôt que d'y aller directement.
- On considère que les coordonnées sont données à partir de zéro.

```

int robot_cupide(int x, int y){

  /* Cas de base */
  if ( (x == 0) && (y == 0) ) then return Damier[x][y];

  /* Autres cas particuliers */
  if (x == 0) then return Damier[x][y] + robot_cupide(x, y - 1);
  if (y == 0) then return Damier[x][y] + robot_cupide(x - 1, y);
}

```

```

/* Cas général x, y > 0 */
return Damier[x][y] + max(robot_cupide(x - 1, y),
                          robot_cupide(x, y - 1));
}

```

3. Un appel à `robot_cupide(3, 3)` entraînera six appels à `robot_cupide(1, 1)`. Partant de la dernière case (Sud-Est) du damier, on peut inscrire, en remontant, sur chaque case du damier le nombre d'appels au calcul du gain sur cette case.

...	...	...	1
...	6	3	1
4	3	2	1
1	1	1	1

On retrouve alors le triangle de Pascal à une rotation près (il faut prendre comme sommet du triangle le coin inférieur droit du tableau). Rien d'étonnant du fait de l'identité binomiale :

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n}{p-1}. \quad (7.1)$$

Le nombre d'appels à `Robot-cupide(i, j)` fait lors du calcul du gain maximum sur la case  $(x, y)$  est

$$\binom{x+y-i-j}{x-i}.$$

Il y a donc un nombre important de répétitions des mêmes appels récursifs.

Une version itérative stockant les gains max dans un nouveau tableau (`gain[n][m]` variable globale) permet d'éviter ces répétitions inutiles.

```

int robot_cupide(int x, int y){
    int i, j;

    gain[0][0] = Damier[0][0];

    /* Bord Nord */
    for (i = 1; i <= x; i++) {
        gain[i][0] = Damier[i][0] + gain[i - 1][0];
    }

    /* Bord Ouest */
    for (j = 1; j <= y; j++) {
        gain[0][j] = Damier[0][j] + gain[0][j - 1];
    }

    /* Autres cases */
}

```

```

for (j = 1; j <= y; j++) {
    for (i = 1; i <= x; i++) {
        gain[i][j] = Damier[i][j]
                    + max(gain[i - 1][j], gain[i][j - 1]);
    }
}
// affiche(x, y); <--- pour afficher...
return gain[x][y];
}

```

Ce n'était demandé mais on peut chercher à afficher la suite des déplacements effectués par le robot. On peut remarquer que le tableau des gains maximaux, c'est à dire le tableau gain après exécution de la fonction précédente (robot itératif), permet de retrouver la case d'où l'on venait quelle que soit la case où l'on se trouve (parmi les provenances possibles, c'est celle ayant la valeur maximum). Il est donc facile de reconstruire le trajet dans l'ordre inverse. Pour l'avoir dans le bon ordre, on peut utiliser une fonction récursive d'affichage qui montre chaque coup à *la remontée* de la récursion c'est à dire après s'être appelée.

Le tableau gain[n][m] contient les gains maximaux.

```

void affiche(int i, int j){
    if (i == 0 && j == 0) {
        printf("depart");
        return;
    }
    if (i == 0) {
        affiche(i, j - 1);          // <--| noter l'ordre de ces deux
        printf(", aller à droite"); // <--| instructions. (idem suite)
        return;
    }
    if (j == 0) {
        affiche(i - 1, j);
        printf(", descendre");
        return;
    }
    if (gain[i - 1][j] > gain[i][j - 1]) {
        affiche(i - 1, j);
        printf(", descendre");
    }
    else {
        affiche(i, j - 1);
        printf(", aller à droite");
    }
}
}

```

## Correction de l'exercice 5.

1. Une solution :

```
double explent(double x, unsigned n){
    double acc = 1;
    int j;
    for (j = 0; j < n; j++){
        acc = acc * x;
    }
    return acc;
}
```

2. L'appel effectuera quatre multiplications (une de plus que naïvement – multiplication par 1, pour des questions d'homogénéité).
3. On calcule ainsi :  $3^4 = (3 \times 3)^2 = 9^2 = 9 \times 9 = 81$ . On effectue donc deux multiplications. Quand on travaille « à la main » on ne fait pas deux fois  $3 \times 3$ . Pour  $3^8 = ((3 \times 3)^2)^2$  on effectue trois multiplications, pour  $3^{16}$  quatre. Pour  $3^{10}$  on peut faire  $3^8 \times 3^2$  c'est à dire  $3 + 1 + 1 = 5$  multiplications. Mais si on remarque qu'il est inutile de calculer deux fois  $3^2$  (une fois pour faire  $3^8$  et une fois pour  $3^2$ ), on obtient que quatre multiplications suffisent.
4. On calcule :

$$x^{256} = \left( \left( \left( \left( \left( (x^2)^2 \right)^2 \right)^2 \right)^2 \right)^2 \right)^2 \quad \text{et}$$

$$x^{32+256} = \left( \left( \left( (x^2)^2 \right)^2 \right)^2 \right)^2 \times \left( (x^{32})^2 \right)^2$$

Ce qui donne respectivement huit et  $5 + 1 + 3 = 9$  multiplications.

5. On se ramène à des calculs où les exposants sont des puissances de deux :

$$x^{\overline{10011}} = x^{\overline{1}} \times x^{\overline{10}} \times x^{\overline{10000}}$$

$$= x \times x^2 \times \left( \left( (x^2)^2 \right)^2 \right)^2.$$

On ne calcule qu'une fois  $x^2$ . On obtient donc  $0 + 1 + 3 + 2 = 6$ , six multiplications sont suffisantes.

6. On décompose, au pire en  $k$  facteurs (reliés par  $k-1$  multiplications) :  $\prod_{j=0}^{k-1} x^{(2^j)}$ . Et il faut  $k-1$  multiplications pour obtenir la valeur de tous les  $k$  facteurs :  $j$  multiplications pour le  $j$ ème facteur  $f_{j-1} = x^{(2^{(j-1)})}$  et une de plus (mise au carré) pour passer au  $j+1$ ème. Ce qui fait  $2k-2$  multiplications dans le pire cas.
7. Une solution :

```

double exprapide(double x, unsigned n){
    double acc = 1;
    while ( n != 0 ){
        if ( n % 2 ) acc = acc * x;
        n = n / 2; // <-- Arrondi par partie entière inférieure
        x = x * x;
    }
    return acc;
}

```

8. Il faut 1023 multiplications pour la version lente, et exactement  $2 \times 10 - 1 = 19$  multiplications pour la version rapide. L'algo est donc de l'ordre de 50 fois plus efficace sur cette entrée si seules les multiplications comptent.
9. Le nombre d'opérations sur les réelles (en fait des multiplications) est  $n$  dans le cas lent. Dans le cas rapide, si  $k$  est la partie entière supérieure de  $\log_2 n$  alors le nombre de multiplications est entre  $k$  et  $2 \times k - 1$ . La borne basse est atteinte lorsque  $n$  est une puissance de 2 (dans ce cas  $n = 2^k$ ). La borne haute est atteinte lorsque le développement en binaire de  $n$  ne contient que des 1 (dans ce cas  $n = 2^k - 1$ ).

Ainsi les complexités en temps de l'algorithme lent et de l'algorithme rapide sont respectivement en  $\Theta(n)$  et en  $\Theta(\log n)$ , où  $n$  est l'exposant.

10.  $x^2 = x \times x$  puis  $x^5 = x^2 \times x^2 \times x$  et enfin  $x^{15} = x^5 \times x^5 \times x^5$ . Cinq multiplications. Alors que, avec l'exponentiation rapide on fait (au mieux) six multiplications.

## Correction de l'exercice 6.

1. Une solution :

```

void drapeau2(tableau_t t){
    int j, k;
    j = 0;          /* les éléments 0, ..., j - 1 de t sont rouges */
    k = taille(t) - 1; /* les éléments k + 1, ..., n - 1 sont verts */
    while ( j < k ){
        if ( couleur(t, j) == 0){
            /* Si t[j] est rouge, il est à la bonne place */
            j++;
        }
        else {
            /* Si t[j] est vert on le met avec les verts */
            echange(t, j, k);
            /* Cet échange fait du nouveau t[k] un vert. On regardera la
               couleur de l'ancien t[k] à l'étape suivante */
            k--;
        }
    }
}

```

```

} /* fin du while :
    On sort avec j = k sans avoir regardé la couleur de l'élément à
    cette place. Aucune importance. */
}

```

2. Une solution :

```

drapeau3(tableau_t t){
    int i, j, k;
    i = 0;          /* Les éléments 0, ..., i - 1 sont rouges */
    j = 0;          /* Les éléments i, ..., j - 1 sont verts */
    k = taille(t) - 1; /* Les éléments k + 1, ..., n - 1 sont bleus */
    while ( j <= k ){
        switch ( couleur(t, j) ){
            case 0: /* ----- rouge ----- */
                /* t[j] doit être mis avec les rouges. */
                if ( i < j ){ /* Si il y a des verts */
                    echange(t, i, j); /* on doit le déplacer, */
                } /* sinon il est à la bonne place. */
                i++; /* Il y a un rouge de plus. */
                j++; /* Le nombre de verts reste le même. */
                break;
            case 1: /* ----- vert ----- */
                /* t[j] est à la bonne place. */
                break;
            case 2: /* ----- bleu ----- */
                echange(t, j, k); /* t[j] est mis avec les bleus. */
                k--; /* Le nombre de bleus augmente. */
        }
    }
}

```

## Correction de l'exercice 7.

Au minimum, il faut bien se rendre chez notre ami, donc parcourir une distance de  $n$  immeubles, ce qui donne bien  $\Omega(n)$ . Ceci suppose que l'on sache dans quel direction partir. Autrement dit, même si on suppose un algorithme capable d'interroger un oracle qui lui dit où aller, la complexité minimale est  $\Omega(n)$ .

Nous n'avons ni oracle ni carnet d'adresses. Pour être sûr d'arriver, il faut passer devant l'immeuble numéro  $n$  et devant l'immeuble numéro  $-n$ . On essaye différents algorithmes.

**Idée simplificatrice pour faire raisonner les étudiants.** Nous considérons uniquement des parcours construits en répétant l'étape suivante :

- choisir un certain numéro  $k$
- parcourir les numéros de 0 à  $k$ , puis de  $k$  à  $-k$  puis de  $-k$  à 0.

Le choix d'un parcours particulier est donc donné par la suite des valeurs de  $k$ . Le temps de parcours d'une  $k$ -étape est de  $4k$ .

Avec cette idée les étudiants doivent arriver à émettre une proposition de parcours et à estimer la distance totale. Voici les deux parcours qu'il faudra nécessairement traiter.

Si les valeurs successives de  $k$  sont tous les entiers jusqu'à  $n$ , la sommation donne  $2n(n+1)$ . On retranche le dernier retour en zéro qui est inutile, ce qui donne :  $2n(n+1) - n = 2n^2 + n = \Theta(n^2)$ .

On fait trop souvent demi-tour.

Si on ne fait demi-tour qu'à chaque fois qu'une puissance de 2 est atteinte, les valeurs successives de  $k$  sont les  $2^i$  pour  $i$  allant de 0 à  $\lceil \log n \rceil$ , alors le parcours total sera de :

$$4 \left( \sum_0^{\lceil \log n \rceil} 2^i \right) - n = 4(2^{\lceil \log n \rceil} - 1) - n = \Theta(n).$$

Bien entendu il faut développer un peu pour convaincre les étudiants de cette dernière égalité.

**Ancienne correction.** Pour le premier, on se déplace aux numéros suivants : 1, -1, 2, -2, 3, -3, ...,  $k$ ,  $-k$ , etc. Les distances parcourues à chaque étape sont 1, 2, 3, 4, 5, 6, ...,  $2k-1$ ,  $2k$ , etc. Ainsi, si notre ami habite au numéro  $n$ , respectivement au numéro  $-n$ , on va parcourir une distance de :

$$\sum_{i=1}^{2n-1} i = \frac{(2n-1)(2n)}{2} \quad \text{respectivement} \quad \sum_{i=1}^{2n} i = \frac{(2n+1)(2n)}{2}$$

en nombre d'immeubles. Cette distance est quadratique en  $n$ .

Effectuer le parcours 1, -2, 3, -4, 5 etc. donnera encore un algorithme quadratique en distance. La raison est simple : entre un ami qui habite en  $n$  ou  $-n$  et un qui habite en  $n+1$  ou  $-(n+1)$  notre algorithme va devoir systématiquement parcourir une distance supplémentaire de l'ordre de  $n$ .

Essayons le parcours : 1, -1, 2, -2, 4, -4, 8, -8, ...,  $2^k$ ,  $-2^k$ , etc.

Si  $u_k$  compte le temps mis pour aller en  $2^k$  et  $-2^k$  (une unité par immeuble) alors :

$$u_k = 2^{k+1} + 2^k + 2^{k-1} + u_{k-1}$$

$$u_0 = 3$$

En dépliant, on a :

$$\begin{aligned}
 u_k &= 2^{k+1} + 2^k + 2^{k-1} \\
 &\quad + 2^k + 2^{k-1} + 2^{k-2} \\
 &\quad \quad + 2^{k-1} + 2^{k-2} + 2^{k-3} \\
 &\quad \quad \quad \vdots \\
 &\quad \quad \quad 2^4 \quad + 2^3 \quad + 2^2 \\
 &\quad \quad \quad \quad 2^3 \quad + 2^2 + 2^1 \\
 &\quad \quad \quad \quad \quad 2^2 + 2^1 + 2^0 + u_0
 \end{aligned}$$

D'où l'on déduit que :

$$\begin{aligned}
 u_k &= 2^{k+1} + 2 \times 2^k + 3 \times \left( \sum_{i=0}^{k-1} 2^i \right) - 2^1 - 2 \times 2^0 + u_0 \\
 u_k &= 7 \times 2^k - 4 = 7n - 4
 \end{aligned}$$

Ceci pour  $n = 2^k$ . Pour  $2^k \leq n \leq 2^{k+1}$ , le temps mis sera majoré par  $7 \times 2^{k+1} - 4$  expression elle-même majorée par  $7 \times (2 \times n) - 4$  puisque  $2^k \leq n$ . Ce qui donne un temps mis pour aller en  $(n, -n)$  majoré par  $14n - 4$ .

Donc la distance parcourue est strictement majorée par  $14 \times n$  ce qui montre que la distance parcourue est cette fois en  $O(n)$ . Conclusion, ce dernier algorithme est en  $\Theta(n)$  et, en complexité asymptotique c'est un algorithme optimal. Si notre ami habite très loin on a économisé quelques années de marche.

Une alternative peut être de doubler la distance parcourue à chaque demi-tour : 1, -1, 3, -5, 11, ce parcours forme une suite  $(u_k)_{k \in \mathbb{N}}$  de terme général :

$$u_k = \sum_{i=0}^{i=k} (-2)^i = \frac{(-2)^{k+1} - 1}{-2 - 1}.$$

On s'arrête lorsque  $|u_k| \geq n$ . On en déduit, après calcul, que  $k = \Theta(\log n)$  et que, là aussi, le temps est en  $\Theta(n)$ .

## Correction de l'exercice 8.

Pas de correction.

## Correction de l'exercice 9.

- On a  $f(n) = O(n)$ ,  $f(n) = O(n \log n)$ ,  $f(n) \neq \Theta(n^2)$  et  $f(n) = \Omega(\log n)$ .  
deux égalités justes = 0, 25, trois 0, 25<sup>+</sup>, une 0<sup>+</sup>

 0.5 pt  
4 min

2. En pire cas, comme en moyenne les tris par comparaison font (au moins)  $\Omega(n \log n)$  comparaisons.  0.5 pt  
4 min
- 0, 25 si une réponse juste parmi moyenne et pire pas

## Correction de l'exercice 10.

Pas de correction.

## Correction de l'exercice 11.

Rappel des définitions (c'était demandé) :

-  $f = O(g)$  ssi

$$\exists c_0 > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq c_0 g(n).$$

-  $f = \Omega(g)$  ssi

$$\exists c_0 > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_0 g(n) \leq f(n).$$

-  $f = \Theta(g)$  ( $f$  ssi  $f = O(g)$  et  $f = \Omega(g)$ )

- Oui  $n^2 - 2n + 1 = O(n^2)$ . En effet,  $n^2 - 2n + 1 = (n-1)^2 \leq 1 \times n^2 (\forall n \geq 1)$  ainsi prendre  $c = 1$  et  $n_0 = 1$  convient.
- Oui. Pour  $i \geq 2$ , on a  $\log i \geq 1$ . Donc  $\sum_{i=1}^n \log i \geq \log 1 + \sum_{i=2}^n 1 = n - 1$ . Lorsque  $n \geq 2$  on a  $\frac{n}{2} - 1 \geq 0$  donc (en ajoutant  $\frac{n}{2}$  de part et d'autre) on a encore  $n - 1 \geq \frac{1}{2}n$ . Ainsi, prendre  $c = \frac{1}{2}$  et  $n_0 = 2$  convient.
- Oui. Puisque  $f = O(g)$ , il existe  $n_0 \in \mathbb{N}$  et  $c_0 > 0$  tels que  $f(n) \leq c_0 g(n), \forall n \geq n_0$ . Et puisque  $f = \Omega(h)$ , il existe  $n_1 \in \mathbb{N}$  et  $c_1 > 0$  tels que  $f(n) \geq c_1 h(n), \forall n \geq n_1$ . Donc, à partir du rang  $n_2 = \max(n_0, n_1) \in \mathbb{N}$ ,  $c_1 h(n) \leq c_0 g(n)$ , ce qui donne, en posant  $c_2 = \frac{c_1}{c_0}$ ,  $g(n) \geq c_2 h(n)$  avec  $c_2 > 0$ . Ce qui montre bien que  $g = \Omega(h)$ .

## Correction de l'exercice 12.

Rappel des définitions (c'était demandé) :

-  $f = O(g)$  ssi

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq c g(n).$$

-  $f = \Omega(g)$  ssi

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c g(n) \leq f(n).$$

-  $f = \Theta(g)$  ssi  $f = O(g)$  et  $f = \Omega(g)$

- Oui  $n \log n = O(n^2)$ . En effet, pour tout  $n \geq 1$ , on a  $\log n \leq n$  d'où encore  $n \log n \leq n^2$ . Ainsi, prendre  $n_0 = 1$  et  $c = 1$  convient.

- Oui. On a  $\log(n!) = \sum_{i=1}^n \log i \leq \sum_{i=1}^n \log n = n \log n$  pour tout  $n \in \mathbb{N}$ . Ainsi, prendre  $c = 1$  et  $n_0 = 0$  convient.
- Non. Il faudrait que  $\log(n!) = O(n^2)$  (ce qui est vrai) et que  $\log(n!) = \Omega(n^2)$ . Mais on n'a pas  $\log(n!) = \Omega(n^2)$ . En effet, supposons que ce soit le cas pour un certain  $n_0$  et un certain  $c$  et montrons qu'il y a contradiction. À partir du rang  $n_0$  on a  $cn^2 \leq \log(n!) = \sum_{i=1}^n \log i \leq n \log n$ . D'où  $cn \leq \log n$  pour tout  $n \leq n_0$  avec  $c > 0$ . Ce qui est impossible puisque  $\lim_{n \rightarrow +\infty} \frac{\log n}{n} = 0$ .

### Correction de l'exercice 13.

- On montre qu'il existe une constante positive  $c$  et un rang  $n_0$  à partir duquel  $(n+3) \log n - 2n \geq cn$ .

$$(n+3) \log n - 2n \geq n(\log n - 2) \quad (\forall n > 0)$$

$$\text{pour } n \geq 8, \quad \log n - 2 \geq \log 8 - 2 = 1$$

$$\text{Donc } \forall n \geq n_0 = 8, \quad (n+3) \log n - 2n \geq 1 \times n.$$

- On montre que c'est faux, par l'absurde. Supposons que ce soit vrai, alors :

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 2^{2n} \leq c \times 2^n.$$

Donc par croissance du log, pour  $n \geq n_0$  et  $n > 0$  :

$$2 \times n \leq \log c + n$$

$$\text{ce qui donne } n \leq \log c$$

L'ensemble des entiers naturels n'étant pas borné et  $c$  étant une constante, ceci est une contradiction.

### Correction de l'exercice 14.

(Rappeler les définitions)

- Réponse Oui. Par définition, on a  $\sum_{i=1}^n i = \Theta(n^2)$  lorsque  $\sum_{i=1}^n i = O(n^2)$  et  $\sum_{i=1}^n i = \Omega(n^2)$ .

$$\text{On a } \sum_{i=1}^n i = \frac{n(n+1)}{2}.$$

Montrons que  $\frac{n(n+1)}{2} = O(n^2)$  c'est à dire que :

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, \frac{n(n+1)}{2} \leq cn^2.$$

Pour  $n \geq 1$ ,  $\frac{n^2}{2} \geq \frac{n}{2}$  donc  $n^2 \geq \frac{n^2+n}{2} = \frac{n(n+1)}{2}$ . Ainsi  $c = 1$  et  $n_0 = 1$  conviennent.

Montrons que  $\frac{n(n+1)}{2} = \Omega(n^2)$  c'est à dire que :

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, \frac{n(n+1)}{2} \geq cn^2.$$

Pour  $n \geq 0$ ,  $\frac{n}{2} \geq 0$  donc  $\frac{1}{2}n^2 \leq \frac{n^2+n}{2} = \frac{n(n+1)}{2}$ . Ainsi  $c = \frac{1}{2}$  et  $n_0 = 0$  conviennent.

2. Non. Supposons que  $n^2 = \Omega(2^n)$  alors, par définition, on aurait un réel positif  $c$  et un entier  $n_0$  tels que

$$\forall n \geq n_0, n^2 \geq c2^n.$$

Ou encore :

$$\forall n > n_0, \frac{2^n}{n^2} \leq \frac{1}{c}.$$

Mais  $\lim_{n \rightarrow \infty} \frac{2^n}{n^2} = +\infty$ , on ne peut donc pas borner par une valeur à partir d'un certain rang. Contradiction.

3. Oui. Montrons que :

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, \sum_{i=0}^n \left(\frac{2}{3}\right)^i \leq c.$$

Il s'agit de la somme de termes positifs donc les sommes partielles seront inférieures à la somme infinie. Il ne reste qu'à montrer que la somme infinie est bornée et à prendre cette borne comme valeur pour  $c$  (et 0 pour  $n_0$ ).

On a

$$\sum_{i=0}^n \left(\frac{2}{3}\right)^i = \frac{q^{n+1} - 1}{q - 1} \text{ où } q = \frac{2}{3}$$

Et  $\lim_{n \rightarrow \infty} \left(\frac{2}{3}\right)^{n+1} = 0$  donc

$$\sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^i = \frac{0 - 1}{\frac{2}{3} - 1} = 3.$$

Ainsi  $c = 3$  et  $n_0 = 0$  conviennent.

## Correction de l'exercice 15.

1. Réponse Oui. Par définition, on a  $\log(n/2) = \Omega(\log n)$  si et seulement si :

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 < c \log n \leq \log(n/2).$$

On a  $\log(n/2) = \log n - 1 = \frac{1}{2} \log n + (\frac{1}{2} \log n - 1)$ . Posons  $c = \frac{1}{2}$  et  $n_0 = 4$ . Lorsque  $n \geq n_0$ ,  $\frac{1}{2} \log n - 1 \geq \frac{1}{2} \log n_0 - 1 = 0$  et donc  $\log(n/2) \geq \frac{1}{2} \log n$ . Ce qui prouve bien que  $\log(n/2) = \Omega(\log n)$ .

2. Réponse non. Par définition, on a  $n = \Omega(n \log n)$  si et seulement si :

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 < cn \log n \leq n.$$

Supposons l'existence d'un tel  $c$  et d'un tel  $n_0$ . Alors pour n'importe quel  $n \geq n_0$  on doit avoir  $\log n \geq \frac{1}{c}$ . Ceci est en contradiction avec le fait que  $\lim_{+\infty} \log n = +\infty$ .

3. Réponse oui. Par définition, on a  $\log(n!) = O(n \log n)$  si et seulement si :

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 < \log(n!) \leq cn \log n.$$

Soient  $c = 1$  et  $n_0 = 0$ . Supposons  $n \geq n_0 = 0$ . On a  $n! \leq n^n$  donc par croissance du log,  $\log(n!) \leq \log(n^n) = n \log n$ . Ce qui montre bien que  $\log(n!) = O(n \log n)$ .

4. Réponse Oui. Le meilleur cas minore la moyenne donc un minorant asymptotique du meilleur cas est également un minorant asymptotique de la moyenne. De même le pire cas majore la moyenne donc un majorant asymptotique de la moyenne est également un majorant asymptotique de la moyenne. Ainsi la moyenne est en  $\Omega(f(n))$  et en  $\Gamma(f(n))$  c'est à dire bien en  $\Theta(f(n))$ .

### Correction de l'exercice 16.

Pas de correction.

### Correction de l'exercice 17.

### Correction de l'exercice 18.

Pas de correction.

### Correction de l'exercice 19.

**Solution 1** Soit  $n > 0$ . Si  $n$  est pair alors  $n = 2k$  avec  $k > 0$ . Dans ce cas :

$$(2k)! \geq (2k) \times \dots \times k \geq \underbrace{k \times \dots \times k}_{k+1 \text{ termes}} \geq k^k$$

Donc  $\log((2k)!) \geq k \log k$ , c'est à dire  $\log(n!) \geq \frac{1}{2}n \log(n/2)$ .

Si  $n$  est impair, alors  $n = 2k + 1$  avec  $k \geq 0$ . Dans ce cas :

$$(2k+1)! \geq (2k+1) \times \dots \times (k+1) \geq \underbrace{(k+1) \times \dots \times (k+1)}_{k+1 \text{ termes}} = (k+1)^{k+1}$$

Donc  $\log((2k+1)!) \geq (k+1) \log(k+1)$  et donc  $\log(n!) \geq \frac{1}{2}n \log(n/2)$ .

Ainsi pour  $n > 0$  on a

$$\log(n!) \geq \frac{1}{2}n \log(n/2).$$

Pour  $n \geq 4$ ,  $n/2 \geq \sqrt{n}$ , et ainsi  $\log(n/2) \geq \log(\sqrt{n}) = \frac{1}{2} \log n$ .

Finalement, pour  $n \geq 4$ ,

$$\log(n!) \geq \frac{1}{4}n \log n.$$

Ce qui montre bien que  $\log(n!) = \Omega(n \log n)$ .

**Solution 2** On a (faire un dessin)

$$\log(n!) = \sum_{k=2}^n \log k \geq \int_1^n \log t dt.$$

Donc  $\log(n!) \geq [t \log t]_1^n = n \log n - n + 1$ . Pour  $n \geq 4$ ,  $\log n - 1 \geq \frac{1}{2} \log n$ . Ainsi pour  $n \geq 4$ ,  $\log(n!) \geq \frac{1}{2}n \log n$ . Ce qui conclue.

**Solution 3** On suppose que  $n > 0$ . On écrit

$$\log(n!) = \sum_{k=1}^n \log k$$

Et comme pour la sommation  $\sum_{k=1}^n k$  on somme deux fois :

$$\begin{aligned} 2 \log(n!) &= \sum_{k=1}^n \log k + \sum_{k=1}^n \log(n+1-k) \\ &= \sum_{k=1}^n \log(k(n+1-k)) \end{aligned}$$

Mais lorsque  $1 \leq k \leq n$ ,  $k(n+1-k)$  est maximal pour  $k = \frac{n+1}{2}$  et minimal pour  $k = 1$  et  $k = n$  (on raisonne sur  $(a+b)(a-b)$  avec  $a = \frac{n+1}{2}$  et  $b = k - a$ ).

Ainsi pour tout  $1 \leq k \leq n$ ,  $\log(k(n+1-k)) \geq \log(n)$ .

On en déduit qu'à partir du rang  $n = 1$  :

$$\log(n!) \geq \frac{n \log n}{2}.$$

Ce qui montre bien que  $\log(n!) = \Omega(n \log n)$ .

## 7.2 Deuxième chapitre

### Correction de l'exercice 20.

1. On écrit une fonction itérative, qui parcourt le tableau de gauche à droite et maintient l'indice du minimum parmi les éléments parcourus.

```
int iminimum(tableau_t t){ /* t ne doit pas être vide */
    int j, imin;
    imin = 0;
    for (j = 1; j < taille(t); j++){
        /* Si T[imin] > T[j] alors le nouveau minimum est T[j] */
        if ( 0 > comparer(t, imin, j) ) imin = j;
    }
    return imin;
}
```

2. On fait exactement  $\text{taille}(\text{tab}) - 1 = N - 1$  appels.
3. On cherche le minimum du tableau et si il n'est pas déjà à la première case, on échange sa place avec le premier élément. On recommence avec le sous-tableau commençant au deuxième élément et de longueur la taille du tableau de départ moins un. ça s'écrit en itératif comme ceci :

```

1 void triselection(tableau_t tab){
2     int n, j;
3     for (n = 0; n < taille(tab) - 1; n++) {
4         j = iminimum(sous_tableau(tab, n, taille(tab) - n));
5         if (j > 0) echanger(tab, n + j, n);
6     }
7 }
```

et en récursif comme ceci :

```

1 void triselectionrec(tableau_t tab){
2     int j;
3     if ( taille(tab) > 1 ){
4         j = iminimum(tab);
5         if (j > 0) echanger(tab, j, 0);
6         triselectionrec(soustableau(tab, 1, taille(tab) - 1));
7     }
8 }
```

4. Traitons le cas itératif.

**On pose l'invariant :** le tableau a toujours le même ensemble d'éléments mais ceux indexés de 0 à  $n - 1$  sont les  $n$  plus petits éléments dans le bon ordre et les autres sont indexés de  $n$  à  $N - 1$  (où on note  $N$  pour  $\text{taille}(\text{tab})$ ).

**Initialisation.** Avant la première étape de boucle  $n = 0$  et la propriété est trivialement vraie (il n'y a pas d'élément entre 0 et  $n - 1 = -1$ ).

**Conservation.** Supposons que l'invariant est vrai au début d'une étape quelconque. Il reste à trier les éléments de  $n$  à la fin. On considère le sous-tableau de ces éléments. À la ligne 4 on trouve le plus petit d'entre eux et  $j$  prend la valeur de son plus petit indice dans le sous-tableau (il peut apparaître à plusieurs indices). L'indice de cet élément  $e$  dans le tableau de départ est  $n + j$ . Sa place dans le tableau trié final sera à l'indice  $n$  puisque les autres éléments du sous-tableau sont plus grands et que dans le tableau général ceux avant l'indice  $n$  sont plus petits. À

la ligne 5 on place l'élément  $e$  d'indice  $n + j$  à l'indice  $n$  (si  $j$  vaut zéro il y est déjà on ne fait donc pas d'échange). L'élément  $e'$  qui était à cette place est mis à la place désormais vide de  $e$ . Ainsi, puisqu'on procède par échange, les éléments du tableau restent inchangés globalement. Seul leur ordre change. À la fin de l'étape  $n$  est incrémenté. Comme l'élément que l'on vient de placer à l'indice  $n$  est plus grand que les éléments précédents et plus petits que les suivants, l'invariant de boucle est bien vérifié à l'étape suivante.

**Terminaison.** La boucle termine lorsque on vient d'incrémenter  $n$  à  $N - 1$ . Dans ce cas l'invariant nous dit que : (i) les éléments indexés de 0 à  $N - 2$  sont à leur place, (ii) que l'élément indexé  $N - 1$  est plus grand que tout ceux là, (iii) que nous avons là tous les éléments du tableau de départ. C'est donc que notre algorithme résout bien le problème du tri.

5. Pour le cas récursif on raisonne par récurrence (facile). On travaille dans l'autre sens que pour l'invariant : on suppose que le tri fonctionne sur les tableaux de taille  $n - 1$  et on montre qu'il marche sur les tableaux de taille  $n$ .

6. Pour le tri itératif : on appelle la fonction `iminimum` autant de fois qu'est exécutée la boucle (3-6), c'est à dire  $N - 1$  fois. Le premier appel à `iminimum` se fait sur un tableau de taille  $N$ , puis les appels suivant se font en décrémentant de 1 à chaque fois la taille du tableau, le dernier se faisant donc sur un tableau de taille 2. Sur un tableau de taille  $K$  `iminimum` effectue  $K - 1$  appels à des comparaisons `cmptab`. On ne fait pas de comparaison ailleurs que dans `iminimum`. Il y a donc au total  $\sum_{k=2}^N k - 1 = \sum_{k=1}^{N-1} k = \frac{N(N-1)}{2}$  appels à `cmptab`.

Chaque exécution de la boucle (3-6) peut donner lieu à un appel à `echangetab`. Ceci fait a priori entre 0 et  $N - 1$  échanges. Le pire cas se réalise, par exemple, lorsque l'entrée est un tableau dont l'ordre a été inversé. Dans ce cas on a toujours  $j > 0$  puisque, à la ligne 4, le minimum n'est jamais le premier élément du sous tableau passé en paramètre. Le meilleur cas ne survient que si le tableau passé en paramètre est déjà trié (dans ce cas  $j$  vaut toujours 0).

La version récursive fournit une formule de récurrence immédiate donnant le nombre  $u_n$  d'appels à la fonction `minimum` pour une entrée de taille  $n$ , qui se réduit immédiatement :

$$\begin{cases} u_1 = 0 \\ u_n = u_{n-1} + 1 \end{cases} \iff u_n = n - 1.$$

Donc  $N - 1$  appels pour un tableau de taille  $N$ . On fait au plus un échange à chaque appel et le pire et le meilleur cas sont réalisés comme précédemment. Cela donne entre 0 et  $N - 1$  échanges.

7. Réponse oui (il faut relire les démonstrations de correction). De plus on remarque qu'avec la manière dont a été écrite notre recherche du minimum, le tri est *stable*. Un tri est dit stable lorsque deux éléments ayant la même clé de tri (ie égaux par comparaison) se retrouvent dans le même ordre dans le tableau trié que dans le tableau de départ. Au besoin on peut toujours rendre un tri stable en augmentant la clé de tri avec l'indice de départ. Par exemple en modifiant la fonction de comparaison de manière à ce que dans les cas où les deux clés sont égales on rende le signe de  $k - j$ .

## Correction de l'exercice 21.

1. On écrit en C. On ne s'occupe pas de l'allocation mémoire, on suppose que le tableau dans lequel écrire le résultat a été alloué et qu'il est passé en paramètre.

```

/* ----- */
/* Interclassement de deux tableaux avec écriture dans un troisième tableau */
/* ----- */
void interclassement(tableau_t t1, tableau_t t2, tableau_t t){
    int i, j, k;
    i = 0;
    j = 0;
    k = 0;
    for (k = 0; k < taille(t1) + taille(t2); k++){
        if ( j == taille(t2) ){/* on a fini de parcourir t2 */
            for (; k < taille(t1) + taille(t2); k++){
                t[k] = t1[i];
            i++;
            }
            break; /* <----- sortie de boucle */
        }
        if ( i == taille(t1) ){/* on a fini de parcourir t1 */
            for (; k < taille(t1) + taille(t2); k++){
                t[k] = t2[j];
            j++;
            }
            break; /* <----- sortie de boucle */
        }
        if ( t1[i] <= t2[j] ){/* choix de l'élément suivant de t : */

```

```

    t[k] = t1[i];                /* - dans t1;          */
    i++;
}
else {
    t[k] = t2[i];                /* - dans t2.          */
    j++;
}
}
}
}

```

2. En pire cas, notre algorithme effectue  $n + m - 1$  comparaisons.

Pour trouver un minorant du nombre de comparaisons pour n'importe quel algorithme, on raisonne sur le tableau trié  $t$  obtenu en sortie. Dans le cas où  $n = m$ , il contient  $2n$  éléments. On considère l'origine respective de chacun de ses éléments relativement aux deux tableaux donnés en entrée. Pour visualiser ça, on peut imaginer que les éléments ont une couleur, noir s'ils proviennent du tableau  $t_1$ , blanc s'ils proviennent du tableau  $t_2$ . On se restreint aux entrées telles que dans  $t$  l'ordre entre deux éléments est toujours strict (pas de répétitions des clés de tri).

**Lemme 7.1.** *Quel que soit l'algorithme, si deux éléments consécutifs  $t[i]$ ,  $t[i+1]$  de  $t$  ont des provenances différentes alors ils ont nécessairement été comparés.*

**Preuve.** Supposons que ce soit faux pour un certain algorithme  $A$ . Alors, sans perte de généralités (quitte à échanger  $t_1$  et  $t_2$ ), il existe un indice  $i$  tel que :  $t[i]$  est un élément du tableau  $t_1$ , disons d'indice  $j$  dans  $t_1$ ;  $t[i+1]$  est un élément du tableau  $t_2$ , disons d'indice  $k$  dans  $t_2$ ; ces deux éléments ne sont pas comparés au cours de l'interclassement. (Remarque :  $i$  est égal à  $j + k$ ). On modifie les tableaux  $t_1$  et  $t_2$  en échangeant  $t[i]$  et  $t[i+1]$  entre ces deux tableaux. Ainsi  $t_1[j]$  est maintenant égal à  $t[i+1]$  et  $t_2[k]$  est égal à  $t[i]$ . Que fait  $A$  sur cette nouvelle entrée? Toute comparaison autre qu'une comparaison entre  $t_1[j]$  et  $t_2[k]$  donnera le même résultat que pour l'entrée précédente (raisonnement par cas), idem pour les comparaisons à l'intérieur du tableau  $t$ . Ainsi l'exécution de  $A$  sur cette nouvelle entrée sera identique à l'exécution sur l'entrée précédente. Et  $t_1[j]$  sera placé en  $t[i]$  tandis que  $t_2[k]$  sera placé en  $t[i + 1]$ . Puisque maintenant  $t_1[j]$  est plus grand que  $t_2[k]$ ,  $A$  est incorrect. Contradiction.

Ce lemme donne un minorant pour le nombre de comparaisons égal au nombre d'alternance entre les deux tableaux dans le résultat. En prenant un tableau  $t$  trié de taille  $2n$  on construit des tableaux en entrée

comme suit. Dans  $t_1$  on met tous les éléments de  $t$  d'indices pairs et dans  $t_2$  on met tous les éléments d'indices impairs. Cette entrée maximise le nombre d'alternance, qui est alors égal à  $2n - 1$ . Par le lemme, n'importe quel algorithme fera alors au minimum  $2n - 1$  comparaisons sur cette entrée (et produira  $t$ ). Notre algorithme aussi. Donc du point de vue du pire cas et pour  $n = m$  notre algorithme est optimal. Des résultats en moyenne ou pour les autres cas que  $n = m$  sont plus difficiles à obtenir pour le nombre de comparaisons. On peut remarquer que pour  $n = 1$  et  $m$  quelconque notre algorithme n'est pas optimal en nombre de comparaisons (une recherche dichotomique de la place de l'élément de  $t_1$  serait par exemple plus efficace).

Par contre, il est clair que le nombre minimal d'affectations sera toujours  $n + m$ , ce qui correspond à notre algorithme.

## Correction de l'exercice 22.

Pas de correction voir celle du partiel 2006 (tri gnome similaire).

## Correction de l'exercice 23.

1. Code C :

```

1 void trignome(tableau_t *t){
2     int i = 0; //                On part du début du tableau t
3     while ( i < taille(t) - 1 ){// Tant qu'on a pas atteint la fin de t
4         if ( cmptab(t, i, i + 1) < 0 ){// Si (i, i + 1) inversion alors :
5             echangetab(t, i, i + 1); // 1) on reordonne par échange;
6             if ( i > 0 ) i--;//        2) on recule, sauf si on était
7             else i++; //                au début, auquel cas on avance.
8         }
9         else i++; //                Sinon, on avance.
10    }
11 }
```

2. La seule hypothèse est, qu'au cours de l'exécution du tri gnome (sur une entrée non spécifiée), à une certaine étape, un échange a eu lieu. Soit  $t'$  le tableau juste avant cet échange,  $t''$  le tableau juste après cet échange, et  $i_0$  la valeur de la variable  $i$  au moment de l'échange. Un échange ne se produit que lorsque  $t[i] > t[i + 1]$  et il s'agit d'un échange entre  $t[i]$  et  $t[i + 1]$ . Ainsi, dans  $t'$ , on avait  $t'[i_0] < t'[i_0 + 1]$  et  $t''$  est égal à  $t'$

dans lequel on a procédé à l'échange entre les éléments d'indices  $i_0$  et  $i_0 + 1$ . Cet échange élimine exactement une inversion. En effet :

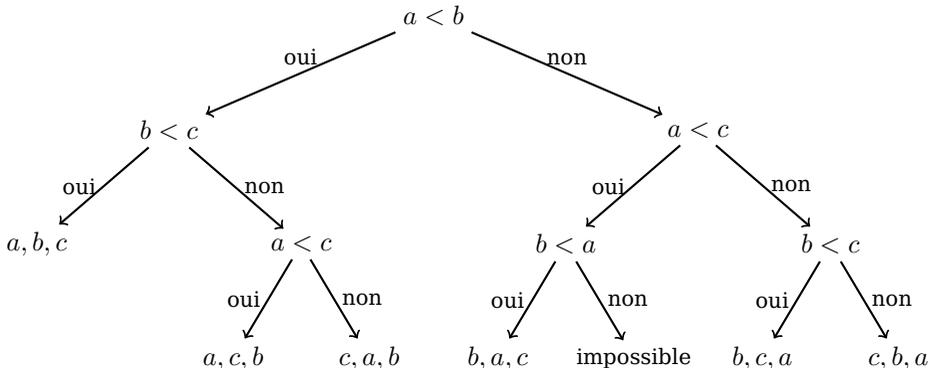
- dans  $t'$ ,  $(i_0, i_0 + 1)$  est une version, pas dans  $t''$
- les autres inversions sont préservées :
  - lorsque  $i < j$  et  $i, j$  tous deux différents de  $i_0$  et  $i_0 + 1$ ,  $(i, j)$  est une inversion dans  $t'$  ssi c'est une inversion dans  $t''$  ;
  - lorsque  $i < i_0$  alors :  $(i, i_0)$  est une inversion dans  $t'$  ssi  $(i, i_0 + 1)$  est une inversion dans  $t''$  et  $(i, i_0 + 1)$  est une inversion dans  $t'$  ssi  $(i, i_0)$  est une inversion dans  $t''$  ;
  - lorsque  $i_0 + 1 < j$  alors :  $(i_0, j)$  est une inversion dans  $t'$  ssi  $(i_0 + 1, j)$  est une inversion dans  $t''$  et  $(i_0 + 1, j)$  est une inversion dans  $t'$  ssi  $(i_0, j)$  est une inversion dans  $t''$  ;
- et ceci prend en considération toutes les inversions possibles dans  $t'$  et  $t''$ .

3. Nous venons de voir que tout échange au cours du tri gnome élimine exactement une inversion. Le nombre d'échanges effectués au cours du tri gnome de  $t$  est donc la différence entre le nombre d'inversions au départ,  $f(n)$ , et le nombre d'inversions à fin du tri. Mais le tableau final est trié et un tableau trié ne contient aucune inversion. Donc le nombre d'échange est simplement  $f(n)$ .

Le nombre d'échanges est égal au nombre d'inversions dans le tableau initial. Donc le nombre moyen d'échanges sur des tableaux de taille  $n$  est  $\frac{n(n-1)}{4}$ .

## Correction de l'exercice 24.

1. L'arbre :



2. Le meilleur cas du tri bulle se réalise lorsque la première passe est sans échange, c'est à dire sur un tableau déjà ordonné, par exemple :  $0, 1, \dots, N - 1$ . Ce tri fait alors  $C(N) = N - 1$  comparaisons (entre 0 et 1 puis 1 et 2, ...,  $N - 2$  et  $N - 1$ ).
3. Il n'est pas possible qu'un algorithme fasse moins de  $N - 1$  comparaisons en meilleur cas. En effet, on sait (cours) que pour trouver le maximum dans un tableau de  $N$  éléments, quel que soient les éléments du tableau, il faut faire au moins  $N - 1$  comparaisons. Or une fois que le tableau est trié on peut trouver ce maximum sans faire de comparaison supplémentaire. Il n'est donc pas possible qu'on ait fait moins de  $N - 1$  comparaisons au cours du tri.
4. Les tris par comparaison font en moyenne  $\Omega(N \log N)$  comparaisons.
5. Supposons qu'un algorithme de tri  $A$  fasse au plus  $N - 1$  comparaisons sur un tableau de taille  $N$ , quel que soit l'ordre initial des éléments. On considère son arbre de décision. Celui est par hypothèse de hauteur  $N - 1$ . Il a donc  $2^{N-1}$  feuilles. Mais toutes les permutations doivent apparaître comme des feuilles différentes de cet arbre et il y en a  $N!$ . Pour  $N > 2$ , on a

$$N! = \underbrace{N \times \dots \times 3 \times 2 \times 1}_{N-1 \text{ termes}} > \underbrace{2 \times \dots \times 2 \times 2 \times 1}_{N-1 \text{ termes}} = 2^{N-1}$$

Ce qui est une contradiction.

## Correction de l'exercice 25.

1. L'algo en C :

```
int Minimum(int T[]){
    int i, x;
    x = T[0];
    for (i = 1; i < taille(T); i++){
        if (T[i] < x) {
            x = T[i];
        }
    }
    return x;
}
```

2. La fonction Minimum fera  $N - 1$  comparaisons, de même pour la fonction Maximum. Donc la fonction MinEtMax en fera  $2 \times (N - 1) = 2N - 2$ .

3. Pour réaliser l'algorithme il faut plusieurs fois trouver le minimum et le maximum entre deux entiers  $a$  et  $b$ , et ceci se fait en une seule comparaison (si  $a < b$  alors le minimum est  $a$  et le maximum est  $b$  sinon c'est l'inverse). On le fait une fois entre  $T[0]$  et  $T[1]$ . Puis pour chaque paire suivante, on fait une comparaison pour trouver le minimum (MinLocal) et le maximum (MaxLocal) dans la paire plus une comparaison pour trouver le minimum entre Minimum et MinLocal et encore une autre comparaison pour trouver le maximum entre Maximum et MaxLocal. Au final cela fait une comparaison pour la première paire et trois comparaisons pour les chacune des  $N/2 - 1$  paires suivantes, c'est à dire  $3N/2 - 2$  comparaisons.

## Correction de l'exercice 26.

1. On cherche  $i$  et  $j$  différents tels que  $t[i] + t[j] = x$ . Par symétrie on peut ne s'intéresser qu'à trouver de tels  $i$  et  $j$  avec  $i < j$ . Pour chaque indice  $i$  allant de 0 à  $N - 2$ , pour chaque  $j$  entre  $i + 1$  et  $N - 1$  on teste si  $t[i] + t[j] = x$ , si c'est vrai on s'arrête et on renvoie vrai. Si à la fin de cette double boucle on n'a rien renvoyé on renvoie faux. Le pire cas advient lorsque on ne trouve pas  $i$  et  $j$ . Ceci va nécessiter  $N - 1 - i$  comparaisons pour chaque  $i$ , c'est à dire  $\sum_{i=0}^{N-2} N - 1 - i = \sum_{i=1}^{N-1} N - i = \sum_{k=1}^{N-1} k = \frac{N(N-1)}{2} = \Theta(N^2)$
2. On peut commencer par trier le tableau avec un tri de complexité  $N \log N$  en pire cas, par exemple un tri par tas, puis pour chaque indice  $i$  (donc  $N$  fois), calculer  $y = x - t[i]$  et chercher par dichotomie si  $y$  est dans le tableau (ailleurs qu'à l'indice  $i$ ), ce qui coûtera  $\log(N)$  en pire cas. Si on trouve  $y$  dans le tableau pour un certain  $i$  alors on s'arrête car on a deux éléments dont la somme est  $x$  sinon il n'existe pas deux éléments différents dont la somme est  $x$ . Le temps total d'exécution en pire cas est celui du tri plus  $N$  fois celui de la recherche dichotomique ( $\log N$ ) donc en  $O(N \log N)$  en pire cas. Remarque : si pour un indice  $i$ , la recherche dichotomique de  $y = x - t[i]$  renvoie le même indice  $i$  alors il faut regarder si  $t[i - 1]$  ou  $t[i + 1]$  ne contiennent pas la même valeur  $y$  (mais sans précision particulière on pouvait supposer que  $t$  contient des valeurs toutes distinctes).

En voici une version en C, qui tient compte de la dernière remarque :

```
int testsomme(int t[], int taille, int x) {
    int i, j, k;
    int y;
    for (i = 0; i < taille; i++) {
```

```

y = x - t[i];
j = recherche_dichotomique(t, taille, y); /* indice de y dans t, -1 sinon */
if (0 <= j) /* y a été trouvé à l'indice j */
    if ((i != j) /* i, j sont déjà distincts */
        || ((0 < i) && (t[i - 1] == y)) /* j = i mais j = i - 1 conviendrait */
        || ((i + 1 < taille) && (t[i + 1] == y))) /* ou j = i + 1 conviendrait */
    {
        return TRUE;
    }
}
return FALSE;
}

```

## Correction de l'exercice 27.

L'algorithme est donné dans le poly du cours.

## Correction de l'exercice 28.

On pose l'invariant de boucle :  $\forall 0 \leq j \leq i - 1, T[j] \leq T[m]$ .

À l'initialisation de la boucle  $i = 1$  et l'invariant est bien vérifié puisque  $m$  vaut 0 et le seul  $j$  pour lequel on doit vérifier  $T[j] \leq T[m]$  est 0.

Conservation. On suppose que l'invariant est vérifié jusqu'au début de la  $i$ -ème étape de boucle et on montre qu'il est encore vérifié au début de l'étape suivante. À l'étape  $i$ ,  $T[m]$  est comparé à  $T[i]$ , il y a alors deux possibilités :

- si  $T[i]$  est supérieur ou égal à  $T[m]$  donc  $T[i]$  est supérieur ou égal à  $T[j]$  pour tout  $0 \leq j \leq i - 1$  (par hypothèse) et bien entendu supérieur ou égal à lui même. Et dans ce cas  $m$  prend la valeur  $i$  donc l'invariant est vérifié au début de l'étape suivante (pour  $i$  incrémenté de 1).
- sinon  $T[i]$  est plus petit que  $T[m]$ ,  $m$  ne change pas et l'invariant est vérifié au début de l'étape suivante (pour  $i$  incrémenté de 1) : pour la valeur de  $i$  correspondant à l'étape actuel on a, par hypothèse, que  $T[j] \leq T[m]$  pour tout  $0 \leq j \leq i - 1$  et on l'a encore pour  $j = i$ .

Terminaison. L'invariant est vérifié au début de l'étape  $i = \text{Taille}(T)$  pour laquelle la condition d'exécution de la boucle devient fausse. Ainsi on a  $T[j] \leq T[m]$  pour tout  $0 \leq j \leq \text{Taille}(T) - 1$ , c'est à dire que  $m$  est bien l'indice du plus grand élément du tableau. Par ailleurs le tableau n'est pas modifié par cet algorithme.

## Correction de l'exercice 29.

## Correction de l'exercice 30.

Les meilleurs algorithmes de tri, tel le tri fusion et le tri par tas, font en pire  $N \log N$  comparaisons. Le fait de renvoyer l'élément d'indice  $k - 1$  a un coût constant quelle que soit la taille du tableau. Le temps total d'exécution sera donc asymptotiquement  $N \log N$  en pire cas.

## Correction de l'exercice 31.

Le premier partitionnement se fait autour de 3 qui est l'élément d'indice 0. Le tableau devient : 2, 1, 3, 8, 6, 9, 5 et  $p$  vaut 2.

Comme  $k = 4 > p + 1 = 3$ , il y a un appel récursif sur le sous-tableau 8, 6, 9, 5 avec pour  $k$  la valeur  $4 - 3 = 1$ .

Le partitionnement se fait autour de 8, le tableau devient 6, 5, 8, 9 et  $p$  vaut 2.

Comme  $k = 1 < p + 1 = 3$ , il y a un appel récursif sur le sous-tableau 6, 5 avec  $k = 1$ .

Le partitionnement se fait autour de 6, le tableau devient 5, 6 et  $p$  vaut 1.

Comme  $k = 1 < p + 1 = 2$ , il y a un appel récursif sur le sous-tableau 5 avec  $k = 1$ .

Le partitionnement se fait autour de 5, laisse le tableau inchangé et renvoie  $p = 0$ .

Finalement, comme  $k = 1 = p + 1$  la valeur 5 (à l'indice  $p$ ) est renvoyée par l'algorithme.

## Correction de l'exercice 32.

1. Dans tous les cas il faudra procéder au moins à un partitionnement. Le meilleur cas se produit lorsque le pivot est l'élément recherché. Par exemple, si le tableau est déjà trié et que c'est l'élément de rang 1 qui est recherché, après le partitionnement on aura  $p = 0$  et le premier élément du tableau sera renvoyé. Ceci prend le temps du partitionnement plus un temps constant (ne dépendant pas de  $N$ ). Ainsi le meilleur cas est en  $\Theta(N)$  où  $N$  est la taille du tableau.
2. Chaque partitionnement réduit d'au moins 1 la taille du tableau. Au pire, l'élément n'est pas trouvé avant d'atteindre dans la récursion une taille de tableau égale à 1 et chaque appel se fait sur un tableau dont la taille diminue de 1 par rapport à l'appel précédent. Ce cas peut tout à fait se produire : par exemple si le tableau est trié et qu'on cherche l'élément

de rang  $N$  (où  $N$  est la taille du tableau initial), alors chaque partitionnement laisse inchangé le tableau et élimine de la zone de recherche un élément à chaque étape (le pivot). Le temps d'exécution est alors la somme des temps des partitionnements :  $\sum_{i=1}^N i = \frac{N(N+1)}{2}$  plus  $N$  fois un temps ne dépendant pas de  $N$ , c'est à dire finalement un temps en  $\Theta(N^2)$ .

3. Si  $T$  est de taille  $N = 2^m$ , qu'il faut attendre la taille 1 pour terminer et que pour chaque appel récursif la taille du tableau est divisée par deux alors la somme des temps des partitionnement est :  $\sum_{i=0}^m 2^i = 2^{m+1} - 1 = 2N - 1$  à laquelle il faut ajouter un temps constant autant de fois qu'il y a de termes dans cette somme ( $m+1$  fois, c'est à dire  $\log N$ ). Ainsi le temps d'exécution dans ce cas est  $2N - 1 + \log N = \Theta(N)$ .
4. En généralisant le raisonnement précédent, sur un tableau de taille  $N$  le temps d'exécution des partitionnement sera :  $P(N) = N + rN + r^2N + \dots + r^mN$  plus ou moins 1 où  $m$  est le nombre de fois où il faut multiplier  $N$  par  $r$  pour trouver 1 :  $m = \frac{\ln N}{-\ln r}$  (qui est en  $\Theta(\log N)$ ). Pour simplifier on peut supposer que  $N = \left(\frac{1}{r}\right)^m$ . Ainsi on obtient un temps d'exécution égal à

$$E(N) = N \left( \sum_{i=0}^m r^i \right) + c \log N$$

où  $c$  est une constante positive, ce qui se simplifie en

$$E(N) = N \left( \frac{1 - r^{m+1}}{1 - r} \right) + c \log N$$

Comme  $0 < r < 1$ ,  $\frac{1 - r^{m+1}}{1 - r}$  tend vers une constante et  $E(N)$  est en  $\Theta(N)$ .

### Correction de l'exercice 33.

1. Première solution, dans un tableau annexe on compte le nombre de fois que chaque entier apparaît et on se sert directement de ce tableau pour produire en sortie autant de 0 que nécessaire, puis autant de 1, puis autant de 2, etc.

```
void trilin1(int t[], int n){
    int j, k;
    int *aux;
    aux = calloc(n * sizeof(int)); //<---- allocation et mise à zéro
```

```

if (aux == NULL) perror("calloc aux trilin1");
for (j = 0; j < n; j++) aux[t[j]]++; // décompte
k = 0;
for (j = 0; j < n; j++) {
    while ( aux[j] > 0 ) {
        t[k] = j;
        k++;
        aux[j]--;
    } // fin du while
} // fin du for
free(aux);
}

```

La boucle de décompte est exécutée  $n$  fois. Si on se contente de remarquer que  $\text{aux}[j]$  est majoré par  $n$ , on va se retrouver avec une majoration quadratique. Pour montrer que l'algorithme fonctionne en temps linéaire, il faut être plus précis sur le contenu du tableau  $\text{aux}$ . Pour cela il suffit de montrer que la somme des éléments du tableau  $\text{aux}$  est égale à  $n$  (à cause de la boucle de décompte). Ainsi le nombre de fois où le corps du `while` est exécuté est exactement  $n$ .

2. Si les éléments de  $t$  ont des données satellites, on procède différemment : on compte dans un tableau auxiliaire; on passe à des sommes partielles pour avoir en  $\text{aux}[j]$  un plus l'indice du dernier élément de clé  $j$ ; puis on déplace les éléments de  $t$  vers un nouveau tableau en commençant par la fin (pour la stabilité) et en trouvant leur nouvel indice à l'aide de  $\text{aux}$ .

```

tableau_t *trilin2(tableau_t *t){
    int j, k;
    int *aux;
    tableau_t out;
    aux = calloc(n * sizeof(int)); //<---- allocation et mise à zéro
    if (aux == NULL) perror("calloc aux trilin2");
    out = nouveautab(taille(t)); // <----- allocation
    for (j = 0; j < n; j++) aux[cle(t, j)]++;
    for (j = 1; j < n; j++) aux[j] = aux[j - 1] + aux[j];
    for (j = n - 1; j >= 0; j--) {
        aux[cle(t, j)]--;
        *element(out, aux[cle(t, j)]) = element(t, j);
    }
    free(aux);
    return out;
}

```

}

3. Tant que l'espace des clés est linéaire en  $n$  l'algorithme sera linéaire. Dans le cas général, l'espace des clés est non borné on ne peut donc pas appliquer cette méthode.

### Correction de l'exercice 34.

1. Liste 1 : 141, 232, 112, 143, 045. Liste 2 : 112, 232, 141, 143, 045. Liste 3 : 045, 112, 141, 143, 232. Ce tri s'appelle un tri par base.
2. On a  $\text{cle}(123, 0) = 3$ ;  $\text{cle}(123, 1) = 2$ ;  $\text{cle}(123, 2) = 1$ . Cette fonction prend un nombre  $n$  et un indice  $i$  en entrée et renvoie le  $i + 1$  ième chiffre le moins significatif de l'expression de  $n$  en base 10. Autrement dit, si  $n$  s'écrit  $\overline{b_{k-1} \dots b_0}$  en base 10, alors  $\text{cle}(n, i)$  renvoie  $b_i$  si  $i$  est inférieur à  $k - 1$  et 0 sinon.
3. Code C :

```
1 void tribase(tableau_t t, int k){
2     int i;
3     for (i = 0; i < k; i++){// k passages
4         triaux(t, i);
5     }
6 }
7
```

4. Sur un tableau de taille  $N$ , le tri base effectue  $k$  appels à `triaux` et chacun de ces appels se fait aussi sur un tableau de taille  $N$ . Les autres opérations (contrôle de boucle) sont négligeables. Chacun de ces appels est majoré en temps par  $f(N)$ . Dans le pire cas, le temps d'exécution du tri gnome est donc majoré par  $kf(N)$ .
5. Étant donné un entier  $n$  d'expression  $\overline{b_{k-1} \dots b_0}$  en base 10, pour  $0 \leq i \leq k - 1$ , on lui associe l'entier  $c_i(n)$  d'expression  $\overline{b_i \dots b_0}$  (les  $i + 1$  premiers digits les moins significatifs de  $n$ ).

On démontre par récurrence sur  $i$  qu'en  $i + 1$  étapes de boucle le tri gnome range les éléments du tableau  $t$  par ordre croissant des valeurs de  $c_i$ . Cas de base. Pour  $i = 0$  (première étape de boucle) le tri gnome a fait un appel à `triaux` et celui-ci a rangé les éléments de  $t$  par ordre croissant du digit le moins significatif. Comme  $c_0$  donne ce digit le moins significatif, l'hypothèse est vérifiée. On suppose l'hypothèse vérifiée après l'étape de boucle  $i$  ( $i + 1$  ème étape). En une étape supplémentaire par l'appel à `triaux`, les éléments de  $t$  sont triés selon leur digit d'indice  $i + 1$  ( $i + 2$  ème digit). Soient  $t[j]$  et  $t[k]$  deux éléments quelconques du tableau après cette étape, avec  $j < k$ . Comme

le tri auxiliaire, `triaux` est stable si  $t[j]$  et  $t[k]$  ont même digits d'indice  $i + 1$  alors ils sont rangés dans le même ordre qu'à l'étape précédente. Mais par hypothèse à l'étape précédente ces deux éléments étaient rangés selon des  $c_i$  croissants, il sont donc rangés par  $c_{i+1}$  croissants. Si les digits d'indices  $i + 1$  de  $t[j]$  et  $t[k]$  diffèrent alors celui de  $t[k]$  est le plus grand et ainsi  $c_{i+1}(t[j]) < c_{i+1}(t[k])$ . Dans les deux cas  $t[j]$  et  $t[k]$  sont rangés par  $c_{i+1}$  croissants. Ainsi l'hypothèse est vérifiée après chaque étape de boucle.

Lorsque  $i = k - 1$ , pour tout indice  $j$  du tableau  $c_i(t[j]) = t[j]$  ainsi le tableau est bien trié à la fin du tri gnome.

La récurrence est faite sur  $i$  qui va de  $0$  à  $k - 1$  (on peut aussi considérer qu'elle porte sur  $k$ ). La stabilité de `triaux` a servi à démontrer le passage de l'étape  $i$  à l'étape  $i + 1$  de la récurrence.

6. L'exécution de la fonction `c1e` demande  $i + 1$  étapes de boucle. Comme, lors d'un appel à `triaux`,  $i$  est borné par  $k$ , le temps d'exécution de `c1e` est linéaire en  $k$ . Mais  $k$  est considéré comme une constante. Le temps d'exécution de `c1e` peut donc être considéré comme borné par une constante.
7. Pour réaliser `triaux` en temps linéaire, il suffit de faire un tri par dénombrement, avec données satellites. Il est alors pratique d'utiliser une structure de pile : on crée dix piles vides numérotées de  $0$  à  $9$ , on parcourt  $t$  du début à la fin en empilant chaque élément sur la pile de numéro la valeur de la clé (son  $i + 1$ ème digit). Ceci prend  $N$  étapes. Lorsque c'est fini on dépile en commençant par la pile numéroté  $9$  et en stockant les éléments dépilés dans  $t$  en commençant par la fin. Ceci prend encore  $N$  étapes.

### **Correction de l'exercice 35.**

Pas de correction.

### **Correction de l'exercice 36.**

Pas de correction.

### **Correction de l'exercice 37.**

### **Correction de l'exercice 38.**

Par ordre croissant :  $\log n, n, n \log n, n^2$ .

Exemples de réponses correctes :

La recherche dichotomique d'un élément dans un tableau ordonné de  $n$  éléments est asymptotiquement en  $\log n$  en pire cas.

La recherche du maximum dans un tableau de  $n$  éléments est asymptotiquement en  $n$  dans tous les cas, en particulier en pire cas.

### Correction de l'exercice 39.

Par ordre croissant :  $\log n, n, n \log n, n^2$ .

Exemples de réponses correctes :

Le tri fusion a une complexité en pire cas de  $n \log n$ .

Le tri rapide a une complexité en pire cas de  $n^2$ .

### Correction de l'exercice 40.

Pas de correction.

### Correction de l'exercice 41.

À écrire (récurrence facile).

## 7.3 Troisième chapitre

### Correction de l'exercice 42.

Il reste *pince-moi je rêve ce sujet est trop facile*.

### Correction de l'exercice 43.

```
1. void pilePaireImpaire(pile_t P1,
    pile_t P2, pile_t P3){
    int aux;

    viderPile(P2);
    viderPile(P3);
    while(pileVide(P1)==0){
        aux=depiler(P1);
        if( (aux&1)==1)
            empiler(P2, aux);
        else
            empiler(P3, aux);
    }
}

while(pileVide(P3)==0){
    aux=depiler(P3);
    empiler(P2, aux);
}

2. void pilePaire(pile_t P1,
    pile_t P2, pile_t P3) {
    int aux;

    viderPile(P2);
    viderPile(P3);

    while(pileVide(P1)==0){
        aux=depiler(P1);
```

```

empiler(P3, aux);
}
while(pileVide(P3)==0){
    aux=depiler(P3);
    empiler(P1, aux);
}

```

```

if( (aux&1)==0)
    empiler(P2, aux);
}
}

```

## Correction de l'exercice 44.

```

int verifieParenthese(char *expr){
    int i, aux;
    pile_t p=creerPile();

    for(i=0; expr[i]!=0; i++){
        switch(expr[i]){
            case '(' :
                empiler(p,1); break;
            case '[' :
                empiler(p,2); break;
            case ')' :
                if(pileVide(p)==1)
                    return -1;
                else{
                    aux=depiler(p);
                    if(aux!=1)
                        return -1;
                }
            break;
        }
    }
}

```

```

case ']' :
    if(pileVide(p)==1)
        return -1;
    else{
        aux=depiler(p);
        if(aux!=2)
            return -1;
    }
    break;
default:
}
}

if(pileVide(p)==0)
    return -1;

return 0;
}

```

## Correction de l'exercice 45.

```

int calculePostfixe(char *expr){
    int i, op1, op2;
    pile_t p=creerPile();

    for(i=0; expr[i]!=0; i++){
        switch(expr[i]){
            case '+' :
                op2=depiler(p);
                op1=depiler(p);
                empiler(p, op1+op2);
                break;
            case '-' :
                op2=depiler(p);

```

```

                op1=depiler(p);
                empiler(p, op1-op2);
                break;
            case '*' :
                op2=depiler(p);
                op1=depiler(p);
                empiler(p, op1*op2);
                break;
            case '/' :
                op2=depiler(p);
                op1=depiler(p);
                empiler(p, op1/op2);
                break;

```



```

void tri_fusion (file_t f1) {
    file_t f2, file_t f3;
    if (!EstVide(f1)) {
        f2 = nouvelleFile();
        f3 = nouvelleFile();
        scinder(f1, f2, f3);
        if (!EstVide(f2)) tri_fusion(f2);
        if (!EstVide(f3)) tri_fusion(f3);
        interclasser(f2, f3, f1);
        detruireFile(f2);
        detruireFile(f3);
    }
}

```

4. La consommation mémoire réside dans les appels récursifs. On crée notamment deux emplacements mémoires à chaque appel à `tri_fusion` à quoi il faut ajouter la mémoire occupée par l'appel de fonction proprement dit (valeur du paramètre, adresse de retour, etc.). Les appels forment un arbre binaire à  $N$  noeuds. Le nombre maximal d'appel imbriqués est la hauteur de l'arbre, qui est en  $\log N$ . L'empreinte mémoire est donc de l'ordre de  $\log N$  ce qui est donc mieux que d'être de l'ordre de  $N$ .

## Correction de l'exercice 47.

1. Il suffit d'insérer un à un les éléments du tableau dans une file de priorité, jusqu'au dernier puis de les retirer un à un en les stockant dans le tableau en commençant à le remplir par la fin.
2. En pire cas les insertions successives sont en  $\Theta(1)$  puis  $\Theta(2)$ ,  $\dots$ ,  $\Theta(\log N)$  (puisque la file est successivement de taille  $1, 2, \dots, N$ ). Chacune de ces insertions est donc en  $O(\log N)$  et il y en a  $N$ . Le temps pris pour mettre tous les éléments dans la file complète est donc en  $O(N \log N)$  (majoration). De même le temps pris pour remplir le tableau dans l'ordre en retirant un à un les éléments de la file va être en  $O(N \log N)$ . Ceci donne une majoration en  $O(N \log N)$  pour l'ensemble du tri, en pire cas. Comme il s'agit d'un tri généraliste, on ne peut pas faire mieux que  $\Omega(N \log N)$  (minoration), en moyenne et en pire cas, notre tri est donc optimal avec une complexité, en pire cas comme en moyenne, de  $\Theta(N \log N)$ . Ce tri est optimal.

## Correction de l'exercice 48.

1. Une pile sera un tableau de NMAX éléments, un entier taille servira à donner la taille de la pile, le sommet de la pile sera la dernière case occupée (celle d'indice taille - 1).

```
#define NMAX 10
typedef int element_t;

typedef struct {
    element_t t[NMAX];
    int taille;
} pile_t;

int estVide(pile_t p) {
    return 0 == p.taille;
}

element_t sommet(pile_t p) {
    if (estVide(p)) {
        perror("Exception: sommet sur pile vide");
    }
    return p.t[p.taille - 1];
}

element_t depiler(pile_t p) {
    if (estVide(p)) {
        perror("Exception: depiler sur pile vide");
    }
    p.taille--;
    return p.t[p.taille];
}

empiler(element_t e, pile_t p) {
    if (p.taille >= NMAX) {
        perror("Exception: depassement de capacite du type pile_t");
    }
    p.t[p.taille] = e;
    p.taille++;
}
```

Rappel et remarque. En C, comme un tableau est simplement l'adresse de sa première case, si `tab1` et `tab2` sont deux tableaux et que l'on effectue l'affectation `tab2 = tab1`, il n'y a pas copie des cases de `tab1`. Ainsi `tab1[i]` a même adresse que `tab2[i]`, ces deux variables sont identiques.

```
int tab1[3] = {100, 200, 300};
int tab2[3];
tab2 = tab1;
tab1[0] = 0;
/* tab2[0] contient 0 */
```

Par contre si `x` et `y` sont deux struct de même type, l'affectation `y = x` déclenche une copie de tout le contenu de l'espace mémoire alloué à `x` dans l'espace mémoire alloué à `y`. Comme un struct contenant un tableau alloué statiquement réserve un espace mémoire contenant ce tableau l'affectation entre struct de ce type copiera le tableau.

```
typedef struct {
    int t[3];
} stab_t;
stab_t x = {{100, 200, 300}};
stab_t y;
y = x;
x.t[0] = 0;
/* y.t[0] vaut 100 */
```

2. Pour implémenter une file dans un tableau nous allons retirer les éléments d'un côté (le début) et les ajouter de l'autre (la fin) le problème est qu'il faudra revenir au début du tableau lorsque l'indice maximum sera atteint.

```
typedef struct {
    element_t t[NMAX];
    int debut; /* indice de la tete de file */
    int taille;
} file_t;
```

```
int estVide(file_t f) {
    return 0 == taille;
}
```

```
element_t tete(file_t f) {
```

```

    if (estVide(f)) {
        perror("Exception: tete sur file vide");
    }
    return f.t[f.debut];
}

ajouter(element_t e, file_t f) {
    int fin;
    if (f.taille >= NMAX) {
        perror("Exception: depassement de capacite du type file");
    }
    fin = (f.debut + f.taille) %NMAX;
    f.t[fin] = e;
    f.taille++;
}

element_t retirer(file_t f) {
    element_t e;
    if (estVide(f)) {
        perror("Exception: retirer sur file vide");
    }
    e = tete(f);
    f.debut = (f.debut + 1) %NMAX;
    f.taille--;
    return e;
}

```

## Correction de l'exercice 49.

```

1. liste_t entrelacement(liste_t liste1,         return liste2;
    liste_t liste2){                             }
    if (liste1 == NULL) return liste2;          }
    if (liste2 == NULL) return liste1;
    if ( cmpListe(liste1, liste2)>=0 ){
        liste1->suivant = entrelacement(
            liste1->suivant, liste2);
        return liste1;
    }
    else {
        liste2->suivant = entrelacement(
            liste1, liste2->suivant);
    }
}

```

```

liste_t entrelacIter(liste_t liste1,
    liste_t liste2) {
    liste_t out, finale;
    if(liste1==NULL)
        return liste2;
}

```

```

if(liste2==NULL)
    return liste1;

if(cmpListe(liste1, liste2)>=0){
    out=liste1;
    liste1=liste1->suivant;
}
else{
    out=liste2;
    liste2=liste2->suivant;
}

finale=out;
while((liste1!=NULL)&&
      (liste2!=NULL) ){
    if(cmpListe(liste1, liste2)>=0){
        out->suivant=liste1;
        liste1=liste1->suivant;
    }
    else{
        out->suivant=liste2;
        liste2=liste2->suivant;
    }
    out=out->suivant;
}

if(liste1==NULL)
    out->suivant=liste2;
else
    out->suivant=liste1;

return finale;
}

2. void elimineRepetition(liste_t liste){
    liste_t courant, aux, suiv;
    courant=liste;
    while(courant!= NULL) {
        aux=courant;
        do {
            suiv=aux->suivant;
            if((suiv!=NULL) &&
              (suiv->element==courant->element)){
                aux->suivant=suiv->suivant;
                free(suiv);
            }
            else
                aux=suiv;
        } while(aux!=NULL);

        courant=courant->suivant;
    }
}

```

## Correction de l'exercice 50.

Pas de correction.

## Correction de l'exercice 51.

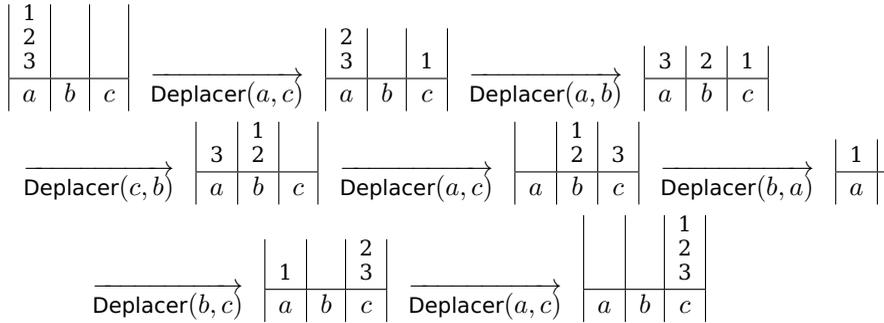
1. Déplacer :

```

void deplacer(pile_t p, pile_t q) {
    empiler(q,depiler(p));
    /* ou encore: if (!EstVide(p)) empiler(q,depiler(p)); */
}

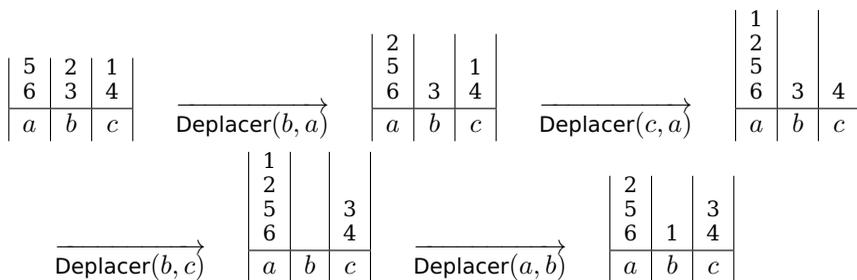
```

2. Pour  $n = 3$  :



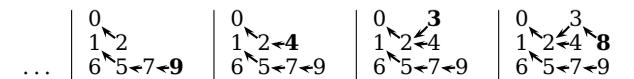
3. Déplacements possibles. Deux à partir de  $p$  puisque 1 peut être posé n'importe où (deux destinations possibles. Et un seul déplacement à partir d'un des autres piquet : si  $d$  et  $d'$  sont les deux disques aux sommets des deux autres piquets avec  $d < d'$  alors  $d'$  ne peut pas être posé sur 1 ou sur  $d$ , et  $d$  peut seulement être posé sur  $d'$ . Deux même si l'une des autres piles est vide (c'est comme si  $d'$  était le fond de la pile vide).
4. Si on déplace un disque deux fois de suite ce n'est pas optimal, autant l'avoir posé à la bonne place dès la première fois.
5. Donc 1 est déplacé un coup sur deux. Comme il est déplacé en premier et en dernier et qu'il y a  $2^n - 1$  déplacements, 1 est déplacé exactement  $2^{n-1}$  fois.
6. Soit 1 occupe tour à tour  $a, b, c, a, b, c, a, b, c, \dots$ . Soit 1 occupe tour à tour  $a, c, b, a, c, b, a, c, b, \dots$ .
7. La séquence des positions de 1 doit se terminer sur  $c$ . Dans les deux séquences précédentes, la position  $a$  correspond à un nombre de déplacements divisible par 3 (0 déplacement pour la première position). S'il y a  $k$  déplacements de 1, dans la séquence  $a, b, c, a, b, c, \dots$  (respectivement  $a, c, b, a, c, b, \dots$ ) on doit avoir  $k \bmod 3 = 2$  (respectivement  $k \bmod 3 = 1$ ), pour atterrir sur le piquet  $c$ . Or il y a  $2^{n-1}$  déplacements du disque 1. Reste donc à déterminer le reste modulo 3 de  $2^{n-1}$  en fonction de  $n$  pour savoir quelle séquence de déplacement choisir. Ce reste n'est jamais nul (c'est cohérent). Il peut être 1 (pour  $n = 1$ ) ou 2 (pour  $n = 2$ ), et en remultipliant par deux pour passer à chaque fois au  $n$  suivant, on obtient  $4 = 1 \bmod 3$  ( $n$  vaut 3), et on tombe sur un cycle ( $1 \times 2 = 2, 2 \times 2 = 4 = 1 \bmod 3$  etc). Pour  $n$  pair il faut donc choisir la séquence  $a, b, c, \dots$  et pour  $n$  impair la séquence  $a, c, b, \dots$ .
8. On est dans le cas  $n$  pair donc 1 se déplace selon la séquence de positions  $a, b, c, \dots$ . Le déplacement qui vient d'être fait ne portait pas sur 1, on

doit donc déplacer 1.



### Correction de l'exercice 52.

1. La suite décroissante  $\sigma = N - 1, N - 2, \dots, 0$  donne le meilleur cas. Dans ce cas il n'y a qu'une seule pile. La suite croissante  $\sigma = 0, \dots, N - 1$  donne le pire cas. Dans ce cas il y a autant de piles que de cartes c'est à dire  $N$ .
2. La suite des cartes en haut des piles est croissante. Pour trouver l'emplacement d'une nouvelle carte, il suffit donc de chercher la dernière pile dont la carte du dessus  $y$  a une valeur inférieure à  $x$  et de poser la carte  $x$  sur la pile suivante. Pour chercher cette pile, on cherche par dichotomie  $x$  dans le tableau des têtes de piles. Ceci prend un nombre de comparaisons en  $\log$  du nombre de piles. Comme il y a au plus  $N$  piles, insérer une carte prend  $O(\log N)$  comparaisons. Donc insérer  $N$  cartes prend  $O(N \log N)$  comparaisons.
3. On remarque que si  $x$  vient d'être placé sur une pile  $T[j]$  alors toute carte  $y$  qui suit  $x$  dans  $\sigma$  et qui est plus grande que  $x$  doit être placée après la pile  $T[j]$ . En effet au moment de placer  $x$ , toutes les piles avant  $T[j]$  ont des cartes du dessus de valeur inférieure à  $x$ . Ces valeurs du dessus, jusqu'à  $T[j]$  incluse ne peuvent que diminuer par ajout de nouvelles cartes. Donc au moment de placer  $y$ , il n'est pas possible de le poser sur une des piles avant  $T[j + 1]$ .  
Supposons que  $a_1 < \dots < a_k$  est une sous-suite de  $\sigma$ . Une fois que  $a_i$  est placé sur une pile,  $a_{i+1}$  est nécessairement placé sur une des piles suivantes. Il faut donc au moins  $k$  piles.
- 4.



5. Si il y a une flèche d'un élément  $x$  vers élément  $y$  alors l'élément  $x$  est plus grand que l'élément  $y$  et  $x$  a été posé après  $y$ . Ainsi une suite obtenue en suivant les flèches donne, en ordre inverse, une suite croissante d'éléments de  $\sigma$  dans l'ordre de leur apparition dans  $\sigma$ , c'est à dire une sous-suite croissante de  $\sigma$ .
  6. Tout élément qui n'est pas dans la première pile possède une flèche vers un élément dans la pile juste avant. Prenons n'importe quel élément de la dernière pile  $T[p - 1]$  et suivons les flèches. Nous aurons alors une sous-suite croissante de  $\sigma$  de longueur  $p$ .
  7. La question précédente nous montre que le nombre de piles  $p$  formées par Réussite( $\sigma$ ) est égal à la longueur d'au moins une sous-suite croissante de  $\sigma$ , et donc que  $p$  minore  $l(\sigma)$ . Nous avons aussi montré (question 3) que quel que soit la stratégie le nombre de piles est toujours supérieur ou égal à  $l(\sigma)$ . On en déduit l'égalité  $p = l(\sigma)$ . D'autre part puisque toute stratégie forme au moins  $l(\sigma)$  piles et que Réussite en forme exactement  $l(\sigma)$ , c'est que Réussite est optimal.
  8. En prenant la suite  $\sigma = 3, 1, 2$  on aura les deux piles :  $\frac{1}{3} \ 2$  et après avoir enlevé 1 les têtes de piles ne vont plus croissante.
  9. Une fois que la carte du dessus de  $T[0]$  est enlevée les têtes des piles  $T[1], \dots, T[p]$  sont toujours croissantes mais la tête de  $T[0]$  n'a plus forcément une valeur inférieure à celle de la tête de  $T[1]$ . Il faut alors procéder par insertion de  $T[0]$  dans la suite du tableau de manière à retrouver la propriété.
10. Code :

```

void rassembler(pile_t T[], int nb_piles, elements_t res[], int nb_elts){
    int i;          /* indice de res (le tableau résultant)          */
    int j;          /* indice de T (le tableau de piles)          */
    int k = 0;     /* indice dans T de la première pile non vide */
    for (i = 0; i < nb_elts; i++) {
        res[i] = depiler(T[k]); /* On récupère le plus petit élément. */
        if ( estvide(T[k]) ) { /* Pile vide: passer à la suivante. */
            k++;
        }
    }
    else {          /* Sinon on doit reclasser les piles. */
        j = k;
        while ( ( j + 1 < nb_piles ) &&
                ( tete(T[j]) > tete(T[j + 1]) ) ) {
            Echanger(T, j , j + 1)
            j++;
        }
    }
}

```

```
    }  
  }  
}  
}
```

## 7.4 Quatrième chapitre

### Correction de l'exercice 53.

Du fait de la propriété de domination, la racine contient l'élément maximum. De même, il est impossible que l'élément minimum soit ailleurs que dans l'une des feuilles (sinon ses descendants seraient plus petits) mais cela peut être n'importe quelle feuille.

### Correction de l'exercice 54.

Le tas obtenu par insertion successives est donné figure 7.1, et le tas obtenu en supprimant l'élément maximum (11) est donné figure 7.2.

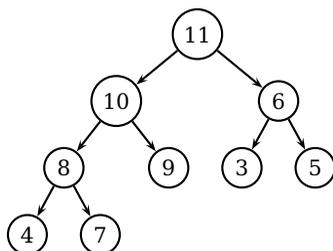


Figure 7.1 – Tas : insertion de tous les éléments

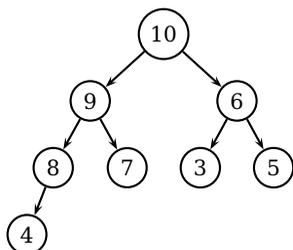


Figure 7.2 – Tas : suppression de la racine (11)

## Correction de l'exercice 55.

Pas de correction.

## Correction de l'exercice 56.

C'est un exercice d'apprentissage du cours. Les solutions sont toutes données au chapitre tas du poly. Pour la première question : l'arbre quasi-parfait obtenu n'est pas un tas, car il ne vérifie pas la propriété de domination ( $12 \not\leq 10$ ).

## Correction de l'exercice 57.

1. On ne représente que le tas intermédiaire obtenu (fig. 7.3).
2. On ne représente que le tas intermédiaire obtenu (fig. 7.4).

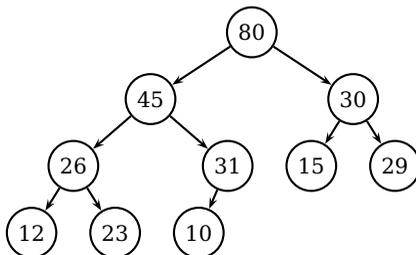


Figure 7.3 – Le tas obtenu après tous les ajouts

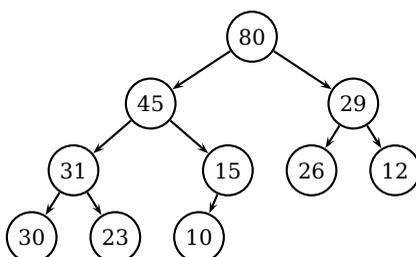


Figure 7.4 – Le tas obtenu par le second algorithme

## Correction de l'exercice 58.

L'idée est de former un tas de piles à partir de ce tableau en considérant que pour comparer deux piles on compare uniquement leurs sommets. Le ta-

bleau vu comme un arbre quasi-parfait est déjà dans le bon ordre pour la propriété de domination. On retire les éléments un à un en les dépilant de la pile en  $T[0]$ . Après chaque retrait :

- Si la pile en  $T[0]$  est vide, on remplace  $T[0]$  par la dernière pile  $T[p-1]$ , et on décrémente  $p$ .
- On rétablit la propriété de domination par maintien bas à partir de  $T[0]$ .

On effectue  $N$  retrait. Chacun de ces retraits prend au maximum un temps  $c + \log p$ , où  $c$  est une constante et où  $\log p$  est la hauteur du tas (coût maximum en temps de la phase de maintien). Comme  $p \leq N$ ,  $\log p$  est majoré par  $\log N$ . Ainsi le temps mis par cet algorithme est majoré par  $cN + N \log N$  où  $c$  est une constante, ce qui est bien en  $O(N \log N)$ .

**Correction de l'exercice 59.**

Pas de correction.

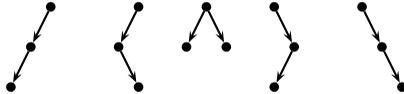
**Correction de l'exercice 60.**

**Correction de l'exercice 61.**

Un seul arbre à un nœud, deux à deux nœuds :



Cinq à trois nœuds :



Quatorze arbres à quatre nœuds (non dessinés). On peut le calculer en trouvant la récurrence :

$$C_{n+1} = \sum_{k=0}^n C_k \times C_{n-k}$$

Qui donne ici  $C_4 = C_0 \times C_3 + C_1 \times C_2 + C_2 \times C_1 + C_3 \times C_0 = 5 + 2 + 2 + 5 = 14$ . (nombres de Catalan).

Pour  $n$  fixé l'arbre quasi-parfait à  $n$  nœuds est tel que si sa hauteur est  $h$  alors :

$$2^{h-1} - 1 < n \leq 2^h - 1$$

D'où  $h - 1 \leq \log n < h$ , donc  $\log n + 1$  majore  $h$ .

L'arbre peigne est quand à lui exactement de hauteur  $n$ .

## Correction de l'exercice 62.

```
int taille(ab_t x) {
    if (estVide(x)) {
        return 0;
    }
    return 1 + taille(gauche(x)), taille(droite(x));
}
```

## Correction de l'exercice 63.

```
void hauteur(ab_t x) {
    if (estVide(x)) {
        return 0;
    }
    return 1 + max(hauteur(gauche(x)), hauteur(droite(x)));
}
```

## Correction de l'exercice 64.

### Le rappel.

```
void parcours_infixe(x){
    if (x) {
        parcours_infixe(x->gauche);
        affiche_element(x->e);
        parcours_infixe(x->droite);
    }
}
```

**Espace mémoire** L'espace mémoire est consommé par la pile d'appel : chaque instance de la fonction occupe un espace constant et le nombre d'instances est plus la hauteur de l'arbre. Le majorant est donc  $O(h)$ . En fait ça peut être moins que la hauteur à cause de l'optimisation de la récursion terminale (dans ce cas le majorant est un plus le max pour toutes les branches de l'arbre du nombre de fois où dans la branche on est descendu à gauche), voir plus bas.

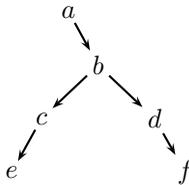
**Avec une pile.** On simule le parcours récursif (la pile rend explicite la pile d'appel qui gère normalement la récursion). On empile les sommets à traiter comme dans le parcours récursif sauf qu'on élimine la récursion terminale (on n'empile jamais un fils droite sur son parent, on dépile le parent d'abord – en mode optimisation, le compilateur va faire ça pour nous).

```

void parcours_infixe_pile(x) {
    ab_t y;
    pile_t p;
    int descendre = 1;
    p = nouvellePile() // p est une pile vide, prête à accueillir des noeuds
    if (x) empiler(p, x);
    while (!estVide(p)) {
        y = sommet(p);
        if (descendre) {                /* Descendre toujours à gauche */
            if (y->gauche) {
                empiler(p, y->gauche);
            } else {                    /* fin de la descente */
                descendre = 0;
            }
        }
        if (!descendre) {              /* On remonte */
            depiler(p);                /* pour ce y, c'est fini */
            afficher_element(y->e);
            if (y->droite) {            /* reprend la descente à droite */
                empiler(y->droite);
                descendre = 1;
            }
        }
    }
}

```

Simulation sur un arbre.



**Départ :** La pile contient *a*, descendre vaut 1 on entre dans la boucle.

**Premier tour :** on met descendre à 0 car *a* n'a pas de fils gauche, puis on dépile *a*, on l'affiche, on empile *b* et on remet descendre à 1.

**Tour 2 :** On empile *c*.

**Tour 3 :** On empile *e*.

**Tour 4 :** On met descendre à 0, on dépile *e* et on l'affiche.

**Tour 5 :** On dépile  $c$  et on l’affiche.

**Tour 6 :** On dépile  $b$ , on l’affiche, on empile  $d$  et on mets descendre à 1.

**Tour 7 :** On mets descendre à 0, on dépile  $d$ , on l’affiche, on empile  $f$  et on remets descendre à 1.

**Tour 8 :** On mets descendre à 0, on dépile  $f$ , on l’affiche.

**Fin** la pile est vide on s’arrête.

**La version sans pile.** On considère le nœud courant et une variable d’état disant si on est actuellement en train de remonter dans l’arbre ou de descendre. Au départ le nœud courant est la racine et on doit descendre. Il y a trois cas de figure.

1. On doit descendre et il y a un fils gauche : on continue de descendre à partir de ce fils.
2. On doit descendre et il n’y a pas de fils gauche, ou bien on doit remonter et le nœud courant est à gauche de son parent. On passe au parent, on l’affiche, et s’il y a un fils droit, on passe au fils droit et on descend, sinon on remonte.
3. Si on doit remonter et que le nœud courant est à droite de son parent ou bien s’il n’y a pas de parent, alors on passe au parent ou bien s’il n’y en a pas on s’arrête.

```
void parcours_infixe_espace_constant(abr_t x){
    abr_t y;
    int descente = 1;
    y = x;
    while (y) {
        if (descente) {
            if (y->gauche) {                /* Cas 1: descente a gauche */
                y = y->gauche;
            } else {                        /* Cas 2: descente a droite */
                afficher_element(y->e);
                if (y->droite) {
                    y = y->droite;
                } else {
                    descente = 0;
                }
            }
        } else {
            if (y->parent && (y == y->parent->gauche) ) { /* Cas 2 */
                /* On remonte pour mieux redescendre à droite */
            }
        }
    }
}
```

```

    y = y->parent;
    afficher_element(y->e);
    if (y->droite) {
        y = y->droite;
        descente = 1;
    }
} else {
    /* Cas 3 */
    /* On remonte vraiment ! */
    y = y->parent;
}
} // fin while
}

```

On peut aussi utiliser les fonctions du cours.

```

void parcours_infixe_espace_constant2(abr_t x) {
    abr_t y;
    if (x) {
        y = minimum(x);
        afficher_element(y->e);
        while ( y = successeur(y) ) {
            afficher_element(y->e);
        }
    }
}

```

## Correction de l'exercice 65.

Pas de correction.

## Correction de l'exercice 66.

### Le rappel.

```

void parcours_infixe(x){
    if (x) {
        parcours_infixe(x->gauche);
        affiche_element(x->e);
        parcours_infixe(x->droite);
    }
}

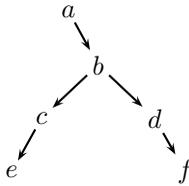
```

**Espace mémoire** L'espace mémoire est consommé par la pile d'appel : chaque instance de la fonction occupe un espace constant et le nombre d'instances est plus la hauteur de l'arbre. Le majorant est donc  $O(h)$ . En fait ça peut être moins que la hauteur à cause de l'optimisation de la récursion terminale (dans ce cas le majorant est un plus le max pour toutes les branches de l'arbre du nombre de fois où dans la branche on est descendu à gauche), voir plus bas.

**Avec une pile.** On simule le parcours récursif (la pile rend explicite la pile d'appel qui gère normalement la récursion). On empile les sommets à traiter comme dans le parcours récursif sauf qu'on élimine la récursion terminale (on n'empile jamais un fils droite sur son parent, on dépile le parent d'abord – en mode optimisation, le compilateur va faire ça pour nous).

```
void parcours_infixe_pile(x) {
    ab_t y;
    pile_t p;
    int descendre = 1;
    p = nouvellePile() // p est une pile vide, prête à accueillir des noeuds
    if (x) empiler(p, x);
    while (!estVide(p)) {
        y = sommet(p);
        if (descendre) {           /* Descendre toujours à gauche */
            if (y->gauche) {
                empiler(p, y->gauche);
            } else {               /* fin de la descente */
                descendre = 0;
            }
        }
        if (!descendre) {         /* On remonte */
            depiler(p);           /* pour ce y, c'est fini */
            afficher_element(y->e);
            if (y->droite) {       /* reprend la descente à droite */
                empiler(y->droite);
                descendre = 1;
            }
        }
    }
}
```

Simulation sur un arbre.



**Départ :** La pile contient  $a$ , descendre vaut 1 on entre dans la boucle.

**Premier tour :** on met descendre à 0 car  $a$  n'a pas de fils gauche, puis on dépile  $a$ , on l'affiche, on empile  $b$  et on remet descendre à 1.

**Tour 2 :** On empile  $c$ .

**Tour 3 :** On empile  $e$ .

**Tour 4 :** On met descendre à 0, on dépile  $e$  et on l'affiche.

**Tour 5 :** On dépile  $c$  et on l'affiche.

**Tour 6 :** On dépile  $b$ , on l'affiche, on empile  $d$  et on mets descendre à 1.

**Tour 7 :** On mets descendre à 0, on dépile  $d$ , on l'affiche, on empile  $f$  et on remets descendre à 1.

**Tour 8 :** On mets descendre à 0, on dépile  $f$ , on l'affiche.

**Fin** la pile est vide on s'arrête.

**La version sans pile.** On considère le nœud courant et une variable d'état disant si on est actuellement en train de remonter dans l'arbre ou de descendre. Au départ le nœud courant est la racine et on doit descendre. Il y a trois cas de figure.

1. On doit descendre et il y a un fils gauche : on continue de descendre à partir de ce fils.
2. On doit descendre et il n'y a pas de fils gauche, ou bien on doit remonter et le nœud courant est à gauche de son parent. On passe au parent, on l'affiche, et s'il y a un fils droit, on passe au fils droit et on descend, sinon on remonte.
3. Si on doit remonter et que le nœud courant est à droite de son parent ou bien s'il n'y a pas de parent, alors on passe au parent ou bien s'il n'y a pas on s'arrête.

```

void parcours_infixe_espace_constant(abr_t x){
  abr_t y;

```

```

int descente = 1;
y = x;
while (y) {
    if (descente) {
        if (y->gauche) {           /* Cas 1: descente a gauche */
            y = y->gauche;
        } else {                   /* Cas 2: descente a droite */
            afficher_element(y->e);
            if (y->droite) {
                y = y->droite;
            } else {
                descente = 0;
            }
        } else {
            if (y->parent && (y == y->parent->gauche) ) { /* Cas 2 */
                /* On remonte pour mieux redescendre à droite */
                y = y->parent;
                afficher_element(y->e);
                if (y->droite) {
                    y = y->droite;
                    descente = 1;
                }
            } else {                /* Cas 3 */
                /* On remonte vraiment ! */
                y = y->parent;
            }
        }
    } // fin while
}

```

On peut aussi utiliser les fonctions du cours.

```

void parcours_infixe_espace_constant2(abr_t x) {
    abr_t y;
    if (x) {
        y = minimum(x);
        afficher_element(y->e);
        while ( y = successeur(y) ) {
            afficher_element(y->e);
        }
    }
}

```

### Correction de l'exercice 67.

Pas de correction.

### Correction de l'exercice 68.

L'insertion est donnée par les figures 7.5 et 7.6.

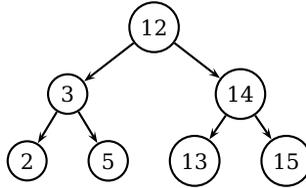


Figure 7.5 – Un arbre binaire de recherche - corrigé

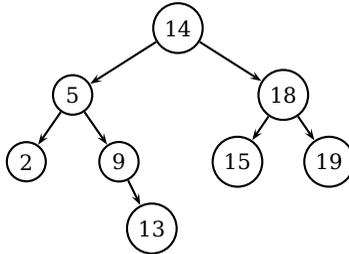


Figure 7.6 – Un autre arbre binaire de recherche - corrigé

La suppression est donnée par les figures 7.7 et 7.8.

1. Figure 7.9.
2. rotation à gauche de centre 5, puis rotation à droite de centre 14.
3. Il faut montrer la préservation de la propriété d'arbre de recherche. On se contente de montrer que si l'arbre à gauche de la figure est un arbre binaire de recherche alors l'arbre à droite en est un, et réciproquement. En effet, pour l'arbre qui contient l'un de ces deux arbres comme sous-arbre, le fait de remplacer l'un par l'autre ne fait aucune différence : les deux sous-arbres ont mêmes ensembles d'éléments. Pour chacun des

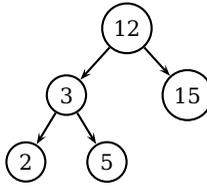


Figure 7.7 – Un arbre binaire de recherche - corrigé B

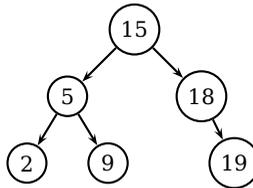


Figure 7.8 – Un autre arbre binaire de recherche - corrigé B

deux arbres de la figure on montre qu'être un arbre de recherche, sous l'hypothèse que  $C$ ,  $D$  et  $E$  en sont, est équivalent à la propriété :  $\forall c \in C, \forall d \in D, \forall e \in E, \text{Clé}(c) \leq \text{Clé}(b) \leq \text{Clé}(d) \leq \text{Clé}(a) \leq \text{Clé}(e)$ , ce qui conclue.

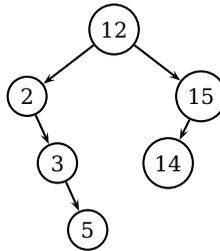


Figure 7.9 – Question 3.1 - corrigé

4. /\* Rotations :

x -> a      ---rot. droite de centre x-->      b <- x

```

      / \
     b  E    <--rot. gauche de centre x--- C  a
    / \
   C  D

```

```

      / \
     C  a
    / \
   D  E

```

```

*/

void rotation_droite(ab_t x){
    element_t tmp;
    ab_t y;

    assert(x && x->gauche);

    y = x->gauche;

    /* échange des éléments */
    tmp = x->e;
    x->e = y->e;
    y->e = tmp;

    /* déplacement des sous-arbres */
    x->gauche = y->gauche;
    y->gauche = y->droite;
    y->droite = x->droite;
    x->droite = y;

    /* mise à jour des parents */
    (x->gauche)->parent = x;
    (y->droite)->parent = y;
}

void rotation_gauche(ab_t x){
    element_t tmp;
    ab_t y;

    assert(x && x->droite);

    y = x->droite;

    /* échange des éléments */

```

```

tmp = x->e;
x->e = y->e;
y->e = tmp;

/* déplacement des sous-arbres */
x->droite = y->droite;
y->droite = y->gauche;
y->gauche = x->gauche;
x->gauche = y;

/* mise à jour des parents */
(x->droite)->parent = x;
(y->gauche)->parent = y;
}

```

5. Voici l'algo en pseudo-code.

---

**Fonction** Remonter( $x$ )

---

```

y = Parent(x);
si y ≠ NULL alors
    /* y existe, x n'est pas la racine */
    si x == Gauche(y) alors
        /* x est fils gauche de y */
        RotationDroite (y);
    sinon
        /* x est fils droit de y */
        RotationGauche (y);
    /* L'élément qui était dans le nœud x est désormais dans le
       nœud y */
    Remonter (y);

```

---

Et en C :

```

void remonter(abr_t x) {
    y = x->parent;
    if (y) { /* x n'est pas encore à la racine */
        if (x == y->gauche) {
            rotation_droite(y);
        }
        else {
            rotation_gauche(y);
        }
    }
    /* l'élément qui était contenu dans x est maintenant dans y */
}

```

```

        remonter(y);
    }
}

```

### Correction de l'exercice 69.

Pas de correction.

### Correction de l'exercice 70.

Dès que  $n$  vaut 3 on est confronté à un problème si  $a$  est l'élément à la racine de l'arbre quasi-parfait et que  $b$  est son fils gauche et  $c$  sont fils droit alors par la propriété des tas on doit avoir  $a \geq c$  et par la propriété des ABR  $a < c$ . Impossible.

### Correction de l'exercice 71.

On peut le faire avec la fonction de parcours infixe. Cette fonction est appelée une fois sur chaque nœud de l'arbre et une fois et chacun de ses appels génère au maximum 2 appels sur des nœuds vides (NULL) ne générant aucun nouvel appel. Ainsi il y a au maximum  $3 * n$  appels à cette fonction au cours d'un parcours (on pourrait être plus précis et trouver  $2n + 1$ ) et chaque appel se faisant en temps constant (pas de boucle), le temps total du parcours est

```

void parcours_infixe(abr_t x) {
    if (x) {
        parcours_infixe(x->gauche);
        affiche_element(x->e);
        parcours_infixe(x->droite);
    }
}

```

Si on pouvait parcourir les éléments d'un tas en ordre trié en temps  $O(n)$ , comme on plante un tas en  $O(n)$ , on aurait un tri par comparaison en  $O(n)$ . Impossible. Le parcours du tas en ordre trié est ainsi nécessairement en  $\Omega(n \log n)$ .

### Correction de l'exercice 72.

Pas de correction.

### Correction de l'exercice 73.

Pas de correction.

### Correction de l'exercice 74.

Pas de correction.

### Correction de l'exercice 75.

Pas de correction.

### Correction de l'exercice 76.

L'énoncé est abrupt pour les forcer à choisir un exemple de tableau et à essayer de construire l'arbre d'appel. La réponse est : l'arbre d'appel dans lequel on ne représente que la valeur du pivot pour chaque appel est exactement le même arbre que celui construit par insertion successives des éléments du tableau dans un abr.

### Correction de l'exercice 77.

1. Réponses :

$$H_n(30) = 3$$

$$H_n(20) = 3$$

$$H_n(35) = 2$$

$$H_n(50) = 1$$

2. On peut prouver cette proposition par induction sur la hauteur de  $x$ .

Si la hauteur de  $x$  est 0,  $x$  est une feuille et le sous-arbre enraciné à  $x$  contient 0 nœud interne, c'est-à-dire  $2^{H_n(x)} - 1 = 2^0 - 1$ .

Soit  $x$  un nœud de hauteur  $h(x) > 0$ . Ce nœud a deux fils,  $g$  et  $d$ . Dans ce cas,  $H_n(g) \geq H_n(x) - 1$  et  $H_n(d) \geq H_n(x) - 1$ . De plus, la hauteur de  $g$  et de  $d$  est inférieure à la hauteur de  $x$ , donc, en appliquant l'hypothèse d'induction à  $g$  et  $d$ , les sous-arbres enracinés à  $g$  et  $d$  possèdent chacun au moins  $2^{H_n(g)} - 1 \geq 2^{H_n(x)-1} - 1$  et  $2^{H_n(d)} - 1 \geq 2^{H_n(x)-1} - 1$  nœuds internes. Il s'ensuit que le sous-arbre enraciné à  $x$  possède au moins  $2(2^{H_n(x)-1} - 1) + 1 = 2^{H_n(x)} - 1$  nœuds internes.

3. Soit  $h$  la hauteur de l'arbre. D'après la propriété 3, au moins la moitié des nœuds d'un chemin simple reliant la racine de l'arbre à une feuille

doivent être noirs (preuve facile par l'absurde). En conséquence, la hauteur noire de la racine vaut au moins  $h/2$ , donc

$$n \geq 2^{h/2} - 1,$$

d'où

$$h \leq 2 \log(n + 1).$$

### Correction de l'exercice 78.

Oui.

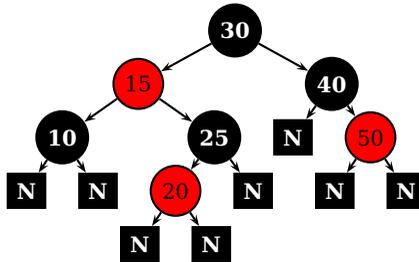


Figure 7.10 – Coloriage corrigé

### Correction de l'exercice 79.

Aucun n'est un rouge noir.

Un *arbre rouge noir* est un arbre binaire de recherche comportant un champ supplémentaire par nœud : sa *couleur*, qui peut valoir soit ROUGE, soit NOIR.

En outre, un arbre rouge noir satisfait les propriétés suivantes :

1. Chaque nœud est soit rouge, soit noir.
  2. Chaque feuille est noire.
  3. Si un nœud est rouge, alors ses deux fils sont noirs.
  4. Pour chaque nœud de l'arbre, tous les chemins descendants vers des feuilles contiennent le même nombre de nœuds noirs.
  5. La racine est noire.
- Le premier arbre (a) n'a pas sa racine noir.
  - Dans le deuxième (b) le chemin vers la troisième feuille contient un seul nœuds noir feuille exclue, tandis qu'un chemin vers la première en contient 2 feuille exclue.

- Le troisième arbre (c) est très joli avec de belles couleurs mais ce n'est pas un ABR donc pas un rouge noir (oui je sais c'est vache).
- Le quatrième (d) a un nœud rouge dont un fils est rouge.

# Table des matières

<b>I</b>	<b>Écriture et comparaison des algorithmes, tris</b>	<b>2</b>
<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	La notion d'algorithme . . . . .	3
1.2	Algorithmique . . . . .	6
<b>2</b>	<b>Les algorithmes élémentaires de recherche et de tri</b>	<b>18</b>
2.1	La recherche en table . . . . .	19
2.2	Le problème du tri . . . . .	21
2.3	Les principaux algorithmes de tri . . . . .	23
2.4	Une borne minimale : $N \log N$ . . . . .	30
2.5	Tris en temps linéaire . . . . .	33
<b>II</b>	<b>Structures de données, arbres</b>	<b>35</b>
<b>3</b>	<b>Structures de données</b>	<b>37</b>
3.1	Structures de données abstraites : pile, file, file de priorité . . . . .	38
3.2	Listes chaînées en C . . . . .	41
<b>4</b>	<b>Arborescences</b>	<b>48</b>
4.1	Arbres binaires parfaits et quasi-parfaits . . . . .	51
4.2	Tas et files de priorité . . . . .	53
4.3	Arbres binaires de recherche . . . . .	64
<b>III</b>	<b>Éléments de calculabilité</b>	<b>69</b>
<b>5</b>	<b>Éléments de calculabilité</b>	<b>70</b>

<b>IV Exercices</b>	<b>77</b>
<b>6 Exercices</b>	<b>78</b>
6.1 Exercices du premier chapitre . . . . .	79
6.2 Exercices du second chapitre . . . . .	86
6.3 Exercices du troisième chapitre . . . . .	99
6.4 Exercices du quatrième chapitre . . . . .	106
<b>7 Correction des exercices</b>	<b>118</b>
7.1 Premier chapitre . . . . .	119
7.2 Deuxième chapitre . . . . .	133
7.3 Troisième chapitre . . . . .	148
7.4 Quatrième chapitre . . . . .	159

# Bibliographie

- [CEBMP<sup>+</sup>94] Jean-Luc Chabert, Michel Guillemot Evelyne Barbin, Anne Michel-Pajus, Jacques Borowczyk, Ahmed Djebbar, and Jean-Claude Martzloff. *Histoire d'algorithmes, du caillou à la puce*. Belin, 1994.
- [CLRS02] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction à l'algorithmique : Cours et exercices (seconde édition)*. Dunod, 2002. 1176 pages, 2,15 Kg.
- [Knu68] D. E. Knuth. *The Art of Computer Programming. Volume 1 : Fundamental Algorithms*. Addison-Wesley, 1968.
- [Knu69] D. E. Knuth. *The Art of Computer Programming. Volume 2 : Seminumerical Algorithms*. Addison-Wesley, 1969.
- [Knu73] D. E. Knuth. *The Art of Computer Programming. Volume 3 : Sorting and Searching*. Addison-Wesley, 1973.

# Table des matières

<b>I</b>	<b>Écriture et comparaison des algorithmes, tris</b>	<b>2</b>
<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	La notion d’algorithme . . . . .	3
1.1.1	Algorithmes et programmes . . . . .	4
1.1.2	Histoire . . . . .	5
1.2	Algorithmique . . . . .	6
1.2.1	La notion d’invariant de boucle . . . . .	6
1.2.2	De l’optimisation des programmes . . . . .	10
1.2.3	Complexité en temps et en espace . . . . .	11
1.2.4	Pire cas, meilleur cas, moyenne . . . . .	11
1.2.5	Notation asymptotique . . . . .	12
1.2.6	Optimalité . . . . .	15
<b>2</b>	<b>Les algorithmes élémentaires de recherche et de tri</b>	<b>18</b>
2.1	La recherche en table . . . . .	19
2.1.1	Recherche par parcours . . . . .	19
2.1.2	Recherche dichotomique . . . . .	19
2.2	Le problème du tri . . . . .	21
2.3	Les principaux algorithmes de tri . . . . .	23
2.3.1	Tri sélection . . . . .	23
2.3.2	Tri bulle . . . . .	24
2.3.3	Tri insertion . . . . .	26
2.3.4	Tri fusion . . . . .	26
2.3.5	Tri rapide . . . . .	28
2.3.6	Tableau récapitulatif (tris par comparaison) . . . . .	29
2.4	Une borne minimale : $N \log N$ . . . . .	30
2.5	Tris en temps linéaire . . . . .	33
2.5.1	Tri du postier . . . . .	34
2.5.2	Tri par dénombrement . . . . .	34

2.5.3 Tri par base . . . . .	34
------------------------------	----

**II Structures de données, arbres 35**

**3 Structures de données 37**

3.1 Structures de données abstraites : pile, file, file de priorité . . . . .	38
3.1.1 Piles . . . . .	39
3.1.2 Files . . . . .	40
3.1.3 File de priorité . . . . .	40
3.2 Listes chaînées en C . . . . .	41
3.2.1 Opérations fondamentales . . . . .	43
3.2.2 Réalisation des piles avec des listes chaînées . . . . .	46
3.2.3 Réalisation des files avec des listes chaînées . . . . .	46

**4 Arborescences 48**

4.1 Arbres binaires parfaits et quasi-parfaits . . . . .	51
4.2 Tas et files de priorité . . . . .	53
4.2.1 Changement de priorité d'un élément dans un tas . . . . .	54
4.2.2 Ajout et retrait des éléments . . . . .	56
4.2.3 Formation d'un tas . . . . .	57
4.2.4 Le tri par tas . . . . .	59
4.3 Arbres binaires de recherche . . . . .	64
4.3.1 Rotations . . . . .	66
4.3.2 Arbres rouge noir . . . . .	67

**III Éléments de calculabilité 69**

**5 Éléments de calculabilité 70**

**IV Exercices 77**

**6 Exercices 78**

6.1 Exercices du premier chapitre . . . . .	79
6.1.1 Récursivité . . . . .	79
6.1.2 Optimisation . . . . .	81
6.1.3 Notation asymptotique . . . . .	83
6.2 Exercices du second chapitre . . . . .	86
6.3 Exercices du troisième chapitre . . . . .	99
6.4 Exercices du quatrième chapitre . . . . .	106

<b>7</b>	<b>Correction des exercices</b>	<b>118</b>
7.1	Premier chapitre . . . . .	119
7.2	Deuxième chapitre . . . . .	133
7.3	Troisième chapitre . . . . .	148
7.4	Quatrième chapitre . . . . .	159