Première partie

Écriture et comparaison des algorithmes, tris

Chapitre 1

Introduction

1.1 La notion d'algorithme

Du point de vue moderne, un algorithme est généralement un procédé de résolution d'un problème, que l'on peut appliquer de manière mécanique et qui termine en un temps fini en rendant, bien évidemment, un résultat correcte. Dans ce cours nous adoptons ce point de vue relativement restrictif.

Pour commencer nous précisons et détaillons notre définition.

Définition 1.1. Un algorithme est :
- un procédé décrit par un ensemble fini de règles,
- qui s'applique à un certain nombre (éventuellement infini) de données finies représentant les instances d'un problème assez général,
- qui, lorsqu'il est exécuté sur une de ces données, termine en un temps fini
- et produit un résultat (fini) représentant la (ou une) solution du problème donné.

Un algorithme *résout un problème*. Par exemple : rechercher un élément dans un liste, trouver le plus grand commun diviseur de deux nombres entiers, calculer une expression algébrique, compresser des données, tracer un motif, factoriser un nombre entier en nombres premiers, trier un tableau, créer une grille de sudoku, etc.

Ce problème doit être assez général. 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'élément correpondant 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 $[0, \ldots, n-1]$) alors nul besoin de trier, il suffit de lire la taille n de la liste donnée en entrée et de rendre la liste $0, \ldots, n-1$ sans plus considérer l'entrée. Il serait incorrect de dire de ce procédé qu'il est un algorithme de tri.

Un algorithme doit terminer en un temps fini, c'est à dire en un nombre fini d'étapes, toutes de temps fini. Contre-exemple : tri *bogo*. Ce tri (encore dénommé tri stupide) 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 en un temps fini avec une probabilité 1 mais il est toujours possible qu'il ne termine jamais. Par exemple sur la liste $[1 = \text{face}, 0 = \text{pile}]$, il est tout à fait possible que 1 (face) sorte en premier, indéfiniment.

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.

Remarquez toutefois que pour une même fonction il peut y avoir plusieurs algorithmes, parfois très différents. Il y a par exemple plusieurs algorithmes de tri, qui réalisent tous la fonction « trier un tableau ».

1.1.1 Algorithmes et programmes

Dans ce cours, parce que c'est pratique, les algorithmes sont écrits 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 exécuté 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$^+$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 IXe siècle, Al Kwarizmi (Abou Jafar Muhammad Ibn Mūsa al-Khuwārizmī) 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 algorisme (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'algorisme. 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 un 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 (IIIe siècle av. JC) qui permet de trouver la liste des nombres premiers plus petit qu'un entier donné quelconque. On écrit la liste des entiers de 2 à N. (i) On sélectionne le premier entier (2) on l'entour 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 cercle, 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'ont 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.

Parmi les critères de comparaison, les plus déterminants d'un point de vue informatique sont certainement les consommations en temps et en espace des algorithmes. 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 indépendemment de l'ordinateur sur lequelle l'algorithme est implanté.

À coté 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*.

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 principalement 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].

1.2.1 La notion d'invariant de boucle

Pour démontrer les propriétés des algorithmes une notion clé est celle 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 consite 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 raisonnement par récurrence (voir exercice 2.1).

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 série d'exécutions de Opposant. C'est ici qu'intervient notre invariant de boucle : on cherche une condition sur (p, q) qui, si elle est vérifié 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 carré.

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.

1.2.2 De l'optimisation des programmes

Les programmes informatiques s'éxecutent 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 corresponde 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.

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, majorée par une constante d. Ainsi en un temps t un algorithme ne pourra pas occuper plus de $d \times t$ 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 habituellement rapide.

On s'intéressera plus rarement au 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.).

En fait, 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.

En fait, il est beaucoup plus intéressant de faire quelques approximations permettant de mener un raisonnement mathématique grace 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'un 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 (pire cas, meilleur cas) on peut se contenter d'une majoration ou d'une minoration du nombre d'opérations significative.

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 1.2. 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 $0 \leq f(n) \leq c_1 g(n)$, *i.e.*

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

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

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

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.*

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

Les notations asymptotiques à l'aide du signe égal, adoptées ici, sont trompeuses 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 secondes (la fréquence du processeur est meilleure, mais il faut une bonne centaine de cycles d'horloge pour effectuer une opération significative).

Le tableau 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 U. correspond à l'estimation courante de l'âge de l'Univers, c'est à dire $13,7$ milliards d'années.

N	10	50	100	500	1 000	10 000	100 000	1 000 000
$\log N$	3 ms	6 ms	7 ms	9 ms	10 ms	13 ms	17 ms	20 ms
$\log^2 N$	11 ms	32 ms	44 ms	80 ms	0,1 s	0,2 s	0,3 s	0,4 s
N	10 ms	50 ms	0,1 s	0,5 s	1 s	10 s	17 min	17 min
$N \log N$	33 ms	0,3 s	0,7 s	4 s	10 s	2 min	28 min	$6h$
$N^{(3/2)}$	32 ms	0,3 s	1 s	11 s	30 s	17 min	$9h$	12 j
N^2	0,1 s	2,5 s	10 s	4 min	17 min	1,2 j	4 mois	32 a
N^3	1 s	2 min	17 min	1,5 j	12 j	32 a	32 siècles	10^7 a
2^N	1 s	35 siècles	10^9 U.					

Les écart 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 ensuite à départager des algorithme de coût 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, voir le temps cubique N^3 sont éventuellement acceptables sur des tailles moyennes de données. Le coût exponentiel, quand à lui est innacceptable en pratique sauf sur de très petites données.

Ces constations sont exacerbés par l'accroissement de la rapidité des ordinateurs, comme le montre le tableau suivant.

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 suivant (les nombres sans unités sont en milliardième de seconde).

N	90	10^3	10^4	10^5	10^6	10^8	10^9	10^{12}
$\log N$	6	10	13	17	20	27	30	40
$\log^2 N$	42	0,1 μs	0,2 μs	0,3 μs	0,4 μs	0,7 μs	0,9 μs	1,5 μs
N	90	1 μs	10 μs	100 μs	1 ms	100 ms	1 s	17 min
$N \log N$	584	9 μs	132 μs	2 ms	20 ms	3 s	30 s	11 h
$N^{(3/2)}$	854	31 μs	1 ms	32 ms	1 s	17 min	9 h	32 a
N^2	8 μs	1 ms	100 ms	10 s	17 min	4 mois	32 a	10^7 a
N^3	0,7 ms	1 s	17 min	12 j	32 a	10^7 a	2 U.	10^9 U.
2^N	3 U.	10^{274} U.						

Il est à noter que même avec un très grande rapidité de calcul les algorithmes exponentiels ne sont praticables que sur des données de très petite taille.

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 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 1.3. *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 1.4. *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 immédiatement du lemme que si n est la taille du tableau, alors A effectue au moins $n-1$ comparaisons. Il reste à prouver le lemme.

Par l'absurde. Soit A un algorithme tel qu'au moins un élément, disons $t[j]$, différent de $t[i_{\texttt{max}}]$ 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 $t[i_{\texttt{max}}]$ pour que l'algorithme soit faux (il devrait rendre j et non $i_{\texttt{max}}$).

Voilà pour la première partie de la proposition. La seconde partie est immédiate, par écriture de l'algorithme : voir exercice 2.1 page 35 (il suffit d'inverser l'ordre pour avoir un algorithme qui trouve le maximum au lieu du minimum).

1.3 Utilisation du langage C : les tableaux

Dans les exercices qui suivent et dans la prochaine partie, où les algorithmes sont en C, on utilise des tableaux d'éléments.

Le type des éléments, `element_t` n'est pas spécifié, on suppose juste que l'on peut comparer les objets de ce type à l'aide d'une fonction `int cmpelt(element_t *e1, element_t *e2)` qui rend -1 lorsque l'élément pointé par `e1` est plus grand que l'élément pointé par `e2`, 0 lorsque les deux éléments sont similaires et 1 sinon.

En C, contrairement à Java par exemple, il n'y pas de méthode pour retrouver à partir d'une variable de type tableau, la taille du tableau. Ainsi si l'on passe un tableau en paramètre à une fonction C, il faut, en général, passer aussi la taille de ce tableau en paramètre. Pour les besoins de ce cours, nous considérerons un type `tableau_t` et une fonction `int taille(tableau_t * t)` qui rend la taille d'un tableau. Le type `tableau_t` peut être défini comme suit :

```
/* Type des tableaux */
typedef struct {
  int taille;     /* nombre d'éléments du tableau     */
  element_t *tab; /* pointeur vers le premier élément */
} tableau_t;
```

Cette solution apporte son lot de difficultés. La plus importante étant le fait que, pour une variable `t` de type `tableau_t`, on accède à l'élément d'indice `i` du tableau avec `t.tab[i]` et non simplement avec `t[i]`.

Lors de l'appel de fonctions, c'est une bonne habitude de passer les données de type `struct` par adresse, c'est à dire en manipulant des pointeurs vers ces données, plutôt que les données elles-mêmes. Les fonctions que nous employerons sur les tableaux prennent donc des pointeurs vers des tableaux en argumant plutôt que directement des tableaux.

On utilisera en particulier les fonctions suivantes :

– Une fonction pour comparer deux éléments

```
int cmptab(tableau_t *t, int j, int k){
  return cmpelt(t->tab + j, t->tab + k);
}
```

– Une fonction d'échange entre éléments

```
void  echangetab(tableau_t *t, int j, int k){
  element_t e;
  e = t->tab[j];
  t->tab[j] = t->tab[k];
  t->tab[k] = e;
}
```

– Des fonctions de création de tableaux

```
/* Un nouveau tableau non initialisé */
tableau_t *newtab(int n) {
  tableau_t *t;
  t = malloc(sizeof(tableau_t));
  if (t == NULL) perror("malloc t newtab");
  t->taille = n;
  t->tab = malloc(n * sizeof(element_t));
  if (t->tab == NULL) perror("malloc tab newtab");
  return t;
}

/* Une nouvelle copie d'un tableau  */
tableau_t *copietab(tableau_t *tin) {
  tableau_t *t;
  int j;
  t = newtab(taille(tin));
  for (j = 0, j < tin->taille; j++) t->tab[j] = tin->[j];
  return t;
}
```

– Des fonctions pour libérer la mémoire

```
/* Nettoyage mémoire */
void videtab(tableau_t *t){
  free(t->tab);
  t->tab = NULL;
  t->taille = 0;
}

void freetab(tableau_t *t){
  videtab(t);
  free(t);
}
```

– Des fonctions pour créer des sous-tableaux d'un tableau

```
/* Sous-tableau d'un tableau, **en place !** */
tableau_t soustab(tableau_t *t, int debut, int taille){
```

```
  tableau_t s;
  s.taille = taille;
  s.tab = t->tab + debut;
  return s;
}

/* Une nouvelle copie d'un sous tableau  */
tableau_t *copiesoustab(tableau_t *tin, int debut, int taille) {
  tableau_t *t;
  int j;
  t = malloc(sizeof(tableau_t));
  if (t == NULL) perror("malloc t copietab");
  t->taille = taille;
  t->tab = malloc(tin->taille * sizeof(element_t));
  if (t->tab == NULL) perror("malloc tab copietab");
  for (j = 0, j < taille; j++) t->tab[j] = tin->[debut + j];
  return t;
}
```

1.4 Exercices et corrigés

Exercice 1.1 (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 nombres d'appels à des opérations arithmétiques sur les réels (addition, soutraction, multiplication, division) et dans une moindre mesure sur les entiers.

1. *Écrire une fonction C de prototype* `double explent(double x,unsigned int n)` *qui calcule x^n.*
2. *Combien de multiplication sur des réels effectuera l'appel* `explent(x, 4)` *?*
3. *Calculer* à la main *et le plus rapidement possible : 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 $n \mod 2$ (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 C de prototype* `double exprapide(double x, unsigned int 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* `explent` *? Et avec la fonction* `exprapide` *?*
9. *Même question, en général, pour x^n (on pourra donner un encadrement du nombre d'opérations effectuées par* `exprapide`*).*

Correction 1.1.

1. Une solution :

```
double explent(double x, unsigned int 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 3×3. 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 :

$$x^{256} = \left(\left(\left(\left(\left(\left(\left(x^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2 \quad \text{et}$$

$$x^{32+256} = \left(\left(\left(\left(x^2\right)^2\right)^2\right)^2\right)^2 \times \left(\left(\left(x^{32}\right)^2\right)^2\right)^2$$

   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 :

$$\begin{aligned} x^{\overline{10011}} &= x^{\overline{1}} \times x^{\overline{10}} \times x^{\overline{10000}} \\ &= x \times x^2 \times \left(\left(\left(x^2\right)^2\right)^2\right)^2. \end{aligned}$$

   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) : $\Pi_{j=0}^{k-1} x^{(2^j)}$. 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 $j+1$ième. Ce qui fait $2k-2$ multiplications dans le pire cas.
7. Une solution :

```
double exprapide(double x, unsigned int 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 $2\times10-1 = 19$ 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 respectives de l'agorithme lent et de l'algorithme rapide sont respectivement en $\Theta(n)$ et en $\Theta(\log n)$ (où n est l'exposant).

Exercice 1.2 (Récursivité)**.**

1. *Écrire un fonction récursive qui calcule la factorielle d'un nombre entier positif.*
2. *En remarquant que $n^2 = (n-1)^2 + 2n - 1$ écrire une fonction récursive qui calcule le carré d'un nombre entier postif.*
3. *Écrire une fonction récursive qui calcule le pgcd de deux nombres entiers positifs.*

Correction 1.2.

1. C'est du déjà vu !

```
unsigned int factorielle(unsigned int n){
  if (n == 0)return 1;
  return n * factorielle(n - 1);
}
```

2. Adaptation facile de la précédente :

```
unsigned int carre(unsigned int n){
  if (n == 0) return 0;
  return carre(n - 1) + 2 * n - 1;
}
```

3. On utilise $\text{pgcd}(a, b) = a$ si $b = 0$ et $\text{pgcd}(a, b) = \text{pgcd}(b, a \mod b)$.

```
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);
}
```

Exercice 1.3 (Tours de Hanoï)**.**
On se donne trois piquets et n disques percés de rayons différents enfilés sur les piquets. On s'autorise une seul opération : prendre le disque du dessus d'un piquet et le déplacer sur un autre piquet. Mais 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 un piquet (disons le premier), par ordre de grandeur avec le plus grand en dessous. On doit déplacer ces n disques vers un autre piquet (disons le troisième). On cherche un algorithme pour résoudre le problème pour n quelconque.

On veut écrire cet algorithme sous la forme d'une fonction C de prototype

```
void deplacertour(unsigned int n, piquet_t p1, piquet_t p2, piquet_t p3);
```

qui déplacera les n disques du piquet `p1` *au piquet* `p3` *en utilisant éventuellement le piquet intermédaire* `p2`*. Le type de donnée* `piquet_t` *n'est pas détaillé, le type représentant les disques non plus. On suppose seulement donnée une fonction C de prototype*

```
void deplacerdisque(piquet_t p, piquet_t q);
```

qui déplace le disque du dessus du piquet `p` *vers le piquet* `q`*, lorsque c'est possible.*

1. *Indiquer une succession de déplacements 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_1 *(quelconque) au dessus d'un autre* p_2 *(quelconque aussi), comment déplacer* n *disques ?*
3. *Écrire la fonction* `deplacertour`.
4. *Combien d'appels à* `deplacerdisque` *fait-on (trouver une forme close en fonction de* n*) ?*
5. *Est-ce optimal (le démontrer) ?*

Correction 1.3.

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 int 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 */
  }
}
```

4. 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 $u_n = 2^n - 1$.
5. 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.

Exercice 1.4 (Drapeau)**.**
On suppose maintenant que les éléments d'un tableau sont de deux couleurs, rouges ou verts. On se donne une fonction `couleur(tableau_t *tab, int j)` *qui regarde la couleur du* $j+1$ *ième élément du tableau (d'indice* j*) et rend 0 si c'est rouge et 1 si c'est vert.*

1. *Écrire un algorithme (écrire aussi la fonction C) qui range les éléments d'un tableau en mettant les verts en premiers et les rouges 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.*

Correction 1.4.

1. Une solution :

```
drapeau2(tableau_t *t){
  int j, k;
  j = 0;             /* Jusqu'à t[j] - 1, les éléments sont rouges  */
  k = taille(t) - 1; /* À partir de t[k] + 1 éléments 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  */
      echangetab1(t, j, k);
      /* Cet échange fait du nouveau t[k] un vert   */
      k--;
    }
  } // fin du while
  /* On sort avec j = k sans avoir regardé la couleur de l'élément à
     cette place : cela n'a pas d'importance. */
}
```

2. Une solution :

```
drapeau3(tableau_t *t){
  int i, j, k;
  i = 0;             /* Jusqu'à t[i] - 1, les éléments sont rouges  */
  j = 0;             /* De t[i] à t[j - 1] les éléments sont verts  */
  k = taille(t) - 1; /* À partir de t[k] + 1, ils sont bleus        */
  while ( j <= k ){
    switch ( couleur(t, j) ){
    case 0: /* ----------- rouge ---------- */
            /* t[j] est mis avec les rouges */
      if ( i < j ) echangetab1(t, i, j);
      i++;  /* un rouge de plus             */
      j++;  /* nb de verts constant         */
      break;
    case 1: /* ----------- vert ----------- */
            /* t[j] est à la bonne place    */
      j++;
      break;
    case 2: /* ----------- bleu ----------- */
            /* t[j] est mis avec les bleus  */
      echangetab1(t, j, k);
      k--;  /* le nombre de bleus augmente  */
    }
  }
}
```

Exercice 1.5 (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, ... 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 coté 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 objectif 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.

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.*

Correction 1.5.
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 d'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.

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. Pour simplifier l'estimation de la distance parcourue, on décide de compter la distance que l'on parcourt entre chaque passage en 0 fait au cours d'un déplacement vers les positifs. Ces passages sont notés dans la suite suivante : 0, 1, -1, 0, 2, -2, 0, 4, -4, 0, 8, -8, 0, ..., 2^k, -2^k, 0, etc. Entre deux de ces passages successifs en 0 on parcourt exactement une distance de 4×1, 4×2, 4×4, 4×8, ..., 4×2^k, etc. Soit $k = \lceil \log n \rceil$. On a alors $n \leq 2^k < 2n$. Le total de la distance parcourue lorsqu'on va en 2^k, puis en -2^k et enfin en 0 est :

$$4 \times \sum_{i=0}^{i=k} 2^i = 4 \times 2^{k+1} - 4 = 8 \times 2^k - 4$$

Il est inutile de retourner en 0 après être passé en -2^k mais tout ce qu'on cherche c'est une majoration linéaire en n de la distance parcourue. Or $2^k < 2n$. Donc la distance parcourue est strictement majorée par $16 \times n$ ce qui montrer que la distance parcourue est cette fois en $O(n)$. Conclusion, ce dernier algorithme est en $\Theta(n)$ et, en complexité asymtotique 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 :

$$u_k = \sum_{i=0}^{i=k} (-2)^i = \frac{(-2)^k - 1}{-2-1}.$$

On s'arrête lorsque $|u_k| \geq n$. On en déduit, après calcul, que $k = \Theta(\log n)$ et que, là aussi, on parcourt une distance en $\Theta(n)$.

Exercice 1.6.
Imaginer des algorithmes de tri de tableaux...

1. *en utilisant uniquement des comparaisons et des échanges ;*
2. *en autorisant aussi la création de tableaux intermédiaires et le déplacement de et vers ces tableaux ;*
3. *Ou bien en utilisant tout ce qui vous semblera utile et en faisant l'hypothèse que l'on ne tri que des tableaux d'entiers entre* 0 *et* $N-1$ *de taille* N*, sans répétitions (permutation), puis avec répétitions.*
4. *Démonter la correction de vos algorithmes.*
5. *En fonction de la taille* N *du tableau à trier, exprimer un ordre de grandeur (ou une majoration) pour le nombres d'étapes importantes (échange, comparaison, copie, ...) qu'effectue chacun de vos algorithmes.*

Correction 1.6.
Tout ceci fait l'objet d'un nouveau chapitre !

Chapitre 2

Les algorithmes élémentaires de recherche et de tri

Dans ce chapitre on s'intéresse à la recherche et au tri d'éléments dans des tableaux d'éléments à une dimension.

On considère des éléments qui 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 : soient 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'un balance à deux plateaux.

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 dans un groupe d'éléments organisés en tableau.

2.1.1 Recherche par parcours

Pour chercher un élément 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 N comparaisons en pire cas et $N/2$ comparaisons en moyenne pour trouver l'élément recherché. Ainsi rechercher un élément dans un tableau est un problème linéaire en la taille tableau.

2.1.2 Recherche dichotomique

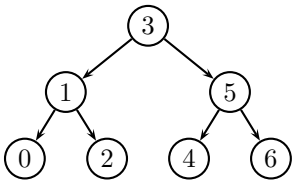
Lorsque le tableau, de taille N est déjà trié, disons par ordre croissant, on peut appliquer une recherche dichotomique. Dans ce cas, nous allons voir que la recherche est au pire cas en $\log N$, et en moyenne en $\Theta(\log n)$. À cet occasion nous introduisons la notion d'arbre de décision, dont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

Rappelons ce qu'est la recherche dichotomique. Étant donné le tableau (trié) et un 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 considèrer est vide et on rend une valeur spéciale, 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.

Supposons que l'on applique cet algorithme à un tableau de taille $N = 2^k - 1$. On représente tous les branchements conditionnels possibles au cours de l'exécution sous la forme d'un arbre. Ce type d'arbre 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.

Toutes les branches de l'arbre terminant sur un noeud cerclé sont possibles. Si l'élément recherché n'est pas dans le tableau on parcourt tout une branche de l'arbre de la racine à une feuille. La hauteur de l'arbre est k. Si l'élément n'est pas dans le tableau on fait donc systématiquement k comparaisons. Sinon on peut faire moins de comparaisons. 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 complet 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 moy(N).

On utilise une technique dite des séries génératrices. Elle consiste à remarquer 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 :

$$S(z) = \frac{z^{k+1} - 1}{z - 1} \quad \text{et, par conséquent} \quad S'(z) = \frac{(k+1)z^k(z-1) - (z^{k+1} - 1)}{(z-1)^2}$$

En fixant $z = 2$ on obtient :

$$\begin{aligned} \text{moy}(N) &= \frac{(k+1)2^k - 2^{k+1} + 1}{1 \times N} \\ &= \frac{(k-1)2^k + 1}{N} \\ &= \frac{(k-1)N + k}{N} \\ &= k - 1 + \frac{k}{N} \end{aligned}$$

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 moyens de comparaisons effectuées par une recherche dichotomique 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 $c \times \log N$. Nous ne rentrons pas dans les détails de cette généralisation. En règle générale, on pourra toujours supposer que les complexités asymptotiques 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. Avec notre type `tableau_t`, on utilisera la fonction `echangetab()` définie au premier chapitre. Lorsque un tri n'est pas place, sa mémoire auxiliaire croît avec la taille du tableau passé en entré : 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. Avec le type `tableau_t`, les fonctions utilisées seront typiquement `newtab()`, `copietab()` ou `copiesoustab()`.

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 même 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.

Les opérations significatives des tris en place sont la comparaison et l'échange. Pour l'échange il s'agit d'une opération élémentaire, et sauf avis contraire, on considérera aussi la comparaison comme une opération élémentaire. 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 généralistes

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 croit au cours du tri.

Le principe du tri sélection est de construire la partie triée en rangeant à leur place définitive les éléments. La partie triée contient les n premiers éléments du tableau dans l'ordre, 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 deux éléments de la partie non triée ont la même clé, lorsque cette clé est le minimum dans la partie non triée, on choisit celui de plus petit indice comme nouvel élément élément de la partie triée.

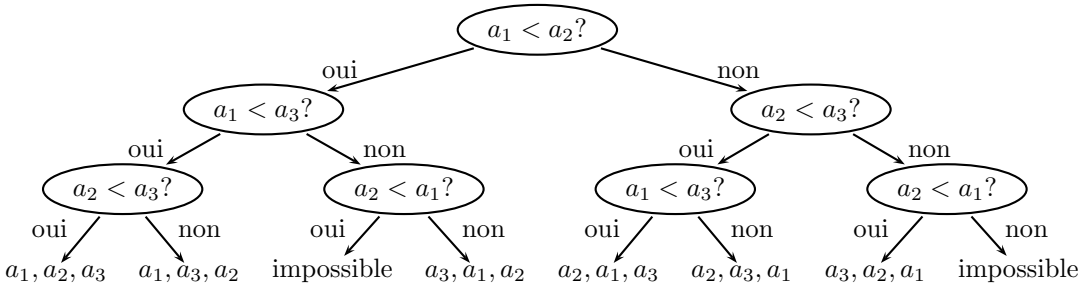
Voici une version en C du tri sélection, voir aussi l'exercice 2.1.

```
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 ( cmptab(t, k, j) < 0 ) k = j;
    }
    /* Sa place est à l'indice n */
    echangetab(t, k, n);
  }
}
```

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

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

Voici 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_1, a_2 et a_3 (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_1 se retrouve à l'indice 1, et a_3 à l'indice 2, on notera $a_1 < a_3$ 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 faîtes 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 (quelque 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 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 (comme pour le tri sélection) 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 une fonction C effectuant le tri bulle :

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

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.

Tri peigne

En anglais : *comb sort,* avril 1991*, Byte magazine,* Stephen Lacey et Richard Box.

Comme le tri bulle ce tri réordonne différentes paires d'éléments. Mais il ne s'agit pas d'éléments consécutifs du tableau. À chaque passe on fixe un écart e entre éléments et on réordonne les paires d'éléments séparées par cet écart : si `t[i] > t[i + e]` on les échange. Une passe commence à `i = 0` et termine à `i + e < taille(t)`. L'écart initial est de l'ordre de 3/4 de `taille(t)`. Il est réduit d'un facteur d'environ $1,3$ entre chaque passe. Lorsqu'on arrive à un écart de 1 on termine comme dans le tri bulle. En choisissant finement les écarts successifs (indépendamment de la donnée) on obtient un algorithme rapide (en $N \log N$).

2.3.3 Tri insertion

Le tri insertion est le tri du joueur de cartes. On maintien 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. Par contre on peu chercher la place de l'élément inséré par dichotomie. Il est facile de voir que ce tri est en $\Theta(N^2)$ en pire cas.

En voici une version C, sans l'amélioration qui consiste à utiliser la dichotomie :

```
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->tab[n];
    /* --2) Décalage des éléments plus grands que e du sous-tableau */
    k = n - 1;
    while ( (k >= 0) && (cmpelt(&e, casetab(t, k)) > 0) ){
      t->tab[k + 1] = t->tab[k];
      k--;
    }
    /* --3) La nouvelle place de l'élément e est à l'indice k + 1   */
    *casetab(t, k + 1) = e; // <-------------------------- t[n] = e
  }
}
```

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. Comme pour le tri peigne, il s'agit d'un tri à incrément variable.

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 $2N$, où N est la taille du tableau fourni en entrée.

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 2.2.

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 effectué par le tri fusion sur cette entrée. Le nombre maximum de comparaisons effectuées est majoré par deux fois le nombre de comparaisons nécessaire 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 0×2^0. Par ailleurs on a :

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

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 conclue 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.

Partionner un tableau de taille n coûte un temps n (on fait comme dans l'exercice 1.4 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 partionnement,le deuxième demandera $N-2$, etc. Le nombre total de comparaisons est alors en $\Theta(N^2)$. C'est le pire cas.

Mais on peut montrer (on ne le fera pas ici) que le tri rapide est en $\Theta(N \log N)$ en moyenne lorsque toutes les permutations possibles sont équiprobables en entrée. Comme il est en place, 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 et 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 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 une version en C du tri rapide.

```
void trirapide (tableau_t *t){
  if (taille(t) > 1) {
    int k;
    tableau_t t1;
    int p = 0;
    /* Randomisation: on choisit le pivot au hasard */
    initrandom();
    echangetab(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 ( 0 > cmptab(t, 0, k) ){
        p++;
        echangetab(t, p, k);
      }
    }
    /* Range le pivot à sa place, p. ------------------------------ */
    echangetab(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] ------------------------- */
    t1 = soustab(t, p + 1, taille(t) - p - 1);
    trirapide(&t1);
  }
}
```

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 partionnement) mais *régner* est simple (il n'y a rien besoin de faire).

On récapitule les principaux algorithmes de tri généralistes dans le tableau suivant (nous n'avons pas tout démontré) :

algorithme	en 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
peigne	$N \log N$	$N \log N$	en place	pas stable
rapide (*quicksort*)	$N \log N$	N^2	en place	pas stable
fusion	$N \log N$	$N \log N$	$\times 2$	stable
par tas	$N \log N$	$N \log N$	en place	stable, non local

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

2.4 Une borne minimale pour les tris par comparaison : $N \log N$

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).

Pour ce faire nous utilisons 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 peuvent être remis dasn l'ordre par une même permutation : ainsi, sur des entiers les entrées $11, 14, 13, 12$ ou $-10, 1, 20, 0$ ou $100, 20000, 10000, 1000$ sont considérées comme équivalents car remettre ces tableaux dans l'ordre consiste à appliquer la même permutation. Finalement on considère que toutes les permutations sont équiprobables.

Soit n'importe quel algorithme de tri A. Considérons l'arbre de décision de A sur une entrée 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 l'entrée doit correspondre à une feuille de cet arbre. De plus, une permutation apparaît au plus comme une feuille, puisque celle-ci 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. Dans la suite on appelle hauteur de la permutation ce nombre.

Il est possible que certaines comparaisons dans cet arbre soient inutiles et qu'elles fassent apparaître des branchements impossibles. On supprime ces comparaisons. Ainsi la hauteur d'une permutation est désormais un minorant du nombre réel de comparaisons effectuées (en un sens on optimise A), chaque feuille est une permutation, et tous les branchements sont binaires.

La plus grande hauteur de permutation minore le nombre de comparaisons en pire cas. Et la moyenne des hauteurs minore le nombre moyen de comparaisons.

Il y a $N!$ permutations donc $N!$ feuilles.

Si la hauteur maximale d'une permutation est k alors son nombre de feuilles est au plus 2^k (démonstration facile par récurrence). Ainsi la hauteur maximale de permutation dans l'arbre est au moins $\log N!$.

On admet que $\log(N!)$ est en $\Omega(N \log N)$ (ceci se démontre à partir de la formule de Stierling).

On en déduit tout de suite qu'en pire cas le nombre de comparaisons est en $\Omega(N \log N)$.

Reste à trouver un minorant pour le nombre moyen de comparaisons. Ce nombre est la somme de toutes les hauteurs de toutes les feuilles (permutations) divisé par $N!$.

On montre que si il existe deux feuilles quelconques dont la hauteur excède 2 alors en modifiant l'arbre de manière à ramener la différence maximale des hauteurs à 1, on diminue la moyenne des hauteurs.

Soit une feuille de hauteur maximale dans l'arbre a. Et soit n le nœud interne juste au dessus de cette feuille. Soit h la hauteur de n. Comme a est de hauteur maximale, les deux branchements issus de n sont des feuilles. Il y a donc a et une autre feuille b toutes les deux de hauteur $h+1$. S'il existe une feuille c de hauteur $h' \leq h-1$, alors on échange c avec n et ses deux branches a et b. La hauteur de c passe de h' à h et la hauteur de a et de b passe de $h+1$ à $h'+1$. La différence de somme des hauteurs entre l'arbre avant transformation et l'arbre après est alors de $h - h' + 2(h'+1) - 2(h+1) = h' - h < 0$. Ainsi au cours de cette transformation la moyenne des hauteurs ne fait que décroître. En répétant la transformation jusqu'à ce qu'elle ne puisse plus avoir lieu, on obtient donc un arbre dont la moyenne des hauteurs minore la moyenne des hauteurs de l'arbre de départ.

Un arbre binaire dans lequel la différence des hauteurs des feuilles est d'au plus un, et qui a K feuilles est tel que chaque feuille est au moins de hauteur $\log K - 1$. Ainsi la moyenne des hauteurs est minorée par la somme de $N!$ fois le terme $\log(N!) - 1$ divisé par $N!$. Autrement dit la moyenne des hauteurs est minorée par $\log(N!) - 1$ qui est aussi en $\Omega(N \log N)$.

Ce qui achève la démonstration du théorème suivant.

Théorème 2.1. *La complexité en moyenne (et en pire cas) d'un tri par comparaison, est (au moins)* $\Omega(N \log N)$.

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 de meilleurs résultats de complexité 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 (la 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 2.4.

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.

2.6 Exercices et corrigés

Exercice 2.1 (Tri sélection)**.**
On suppose donnés :

- *un type* `element_t` *représentant des éléments ;*
- *un type* `tableau_t` *représentant les tableaux d'éléments indicés à partir de* 0 *;*
- *une fonction* `int taille1(tableau_t tab)` *qui rend la taille du tableau* `tab` *;*
- *une fonction* `int cmptab1(tableau_t tab, int j, int k)` *qui compare le* $j+1$*ème élément et le* $k+1$ *ième élément du tableau, rend* 0 *s'ils sont égaux,* -1 *si le premier est plus grand,* $+1$ *si le second est plus grand.*

1. *Écrire une fonction de prototype* `int iminimum(tableau_t tab)` *qui rend le premier indice auquel apparaît le plus petit élément du tableau* `tab`*.*
2. *Combien d'appels à la fonction de comparaison* `cmptab` *effectue votre fonction sur un tableau de taille* N *?*

On se donne de plus :

- *une fonction* `void echangetab(tableau_t tab, int j, int k)` *qui échange l'élément* j *et l'élément* k *du tableau. Notez que cette fonction ne renvoie pas de valeurs particulière, le type* `tableau_t` *est géré avec des pointeurs et la fonction d'échange agit directement sur les cases mémoires du tableau (les adresses de ces cases sont obtenues à partir de la valeur de la variable* `tab`*) ;*
- *une fonction* `tableau_t soustab(tableau_t tab, int j, int L)` *qui rend le sous-tableau de* `tab` *commençant à l'indice* j *et de taille* L *: il ne s'agit pas d'une copie, une modification de ce tableau sera répercuté par une modification du tableau* `tab`*.*

3. *Imaginer un algorithme de tri des tableaux qui utilise la recherche du minimum du tableau. L'écrire sous la forme d'une fonction itérative* `void tri1(tableau_t tab)` *et sous la forme d'une fonction récursive* `void tri2(tableau_t tab)`*.*
4. *Combien d'appels à la fonction* `iminimum` *et sur des entrées de quelle taille effectuent chacun de vos tris sur un tableau de taille* N *? Combien d'appels à la fonction* `cmptab` *cela représente t-il ? Combien d'appels à* `echangetab` *(donner un encadrement et décrire un tableau réalisant le meilleur cas et un tableau réalisant le pire cas) ?*
5. *Démontrer à l'aide d'un invariant de boucle que votre fonction de tri itérative est correcte.*
6. *Démontrer que votre fonction de tri récursive est correcte. Quelle forme de raisonnement très courante en mathématiques utilisez-vous à la place de la notion d'invariant de boucle ?*
7. *Vos algorithmes fonctionnent-ils dans le cas où plusieurs éléments du tableau sont égaux ?*

Correction 2.1.

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 *tab){ /* tab 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 ( cmptab(t, imin, j) < 0 ) imin = j;
  }
  return imin;
}
```

2. On fait exactement `taille1(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 triselection1(tableau_t *tab){
2    int n, j;
3    for (n = 0; n < taille(tab) - 1; n++) {
4      j = iminimum1(soustab(tab, n, taille(tab) - n));
5      if (j > 0) echangetab(tab, n + j, n);
6    }
7  }
```

et en récursif comme ceci :

```
1  void triselectionrec(tableau_t *tab){
2    int n, j;
3    if ( taille(tab) > 1 ){
4      j = iminimum(tab);
5      if (j > 0) echangetab(tab, j, 0);
6      triselectionrec(soustab(tab, 1, taille(tab) - 1));
7    }
8  }
```

4. 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 de comparaisons u_n 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 $N - 1$ comparaisons pourun tableau de taille N. On fait au plus un échange à chaque appel et le pire et le meilleur cas sont réalisés comme précedemment. Cela donne entre 0 et $N - 1$ échanges.

On pose l'invariant : le tableau a toujours le même ensemble d'éléments mais ceux indicés de 0 à $n-1$ sont les n plus petits éléments dans le bon ordre et les autres sont indicés de n à $N - 1$ (où on note N pour `taille(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 élements 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 indicés de 0 à $N - 2$ sont à leur place, (ii) que l'élément indicé $N - 1$ 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.

6. Raisonnement 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.
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 $k - j$.

Exercice 2.2 (Interclassement)**.**
Soient deux tableaux d'éléments comparables `t1` *et* `t2` *de tailles respectives n et m, tous les deux triés dans l'ordre croissant.*

1. *Écrire (en C) une algorithme d'interclassement des tableaux* `t1` *et* `t2` *qui rend le tableau trié de leurs éléments (de taille $n + m$).*
2. *Dans le pire des cas, combien de comparaisons faut-il faire au minimum, quel que soit l'algorithme choisi, pour réussir l'interclassement lorsque $n = m$? Votre algorithme est-il optimal ? (Démontrer vos résultats).*

Correction 2.2.

1. 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->tab[k] = t1->tab[i];
i++;
      }
      break; /* <-------- sortie de boucle */
    }
    if ( i == taille(t1) ){/* on a fini de parcourir t1 */
      for (; k < taille(t1) + taille(t2); k++){
        t->tab[k] = t2->tab[j];
j++;
      }
      break; /* <-------- sortie de boucle */
    }
    if ( t1[i] <= t2[j] ){/* choix de l'élément suivant de t : */
      t->tab[k] = t1->tab[i];          /* - dans t1;           */
      i++;
    }
    else {
      t->tab[k] = t1->tab[i];          /* - dans t2.           */
      j++;
    }
  }
}
```

2. En pire cas, notre algorithme effectue $n + m - 1$ comparaisons.

   Pour trouver un minorant du nombre de comparaisons, quel que soit l'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 `t1`, blanc s'ils proviennent du tableau `t2`. 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 2.2. *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 `t1` et `t2`), il existe un indice `i` tel que : `t[i]` est un élément du tableau `t1`, disons d'indice `j` dans `t1` ; `t[i+1]` est un élément du tableau `t2`, disons d'indice `k` dans `t2` ; ces deux éléments ne sont pas comparés au cours de l'interclassement. (Remarque : `i` est égal à `j + k`). On modifie les tableaux `t1` et `t2` en échangeant `t[i]` et `t[i+1]` entre ces deux tableaux. Ainsi `t1[j]` est maintenant égal à `t[i+1]` et `t2[k]` est égal à `t[i]`. Que fait A sur cette nouvelle entrée ? Toute comparaison autre qu'une comparaison entre `t1[j]` et `t2[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 `t1[j]` sera placé en `t[i]` tandis que `t2[k]` sera placé en `t[i + 1]`. Puisque maintenant `t1[j]` est plus grand que `t2[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 `t1` on met tous les éléments de `t` d'indices pairs et dans `t2` 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 `t1` serait plus efficace). par contre, il est clair que le nombre minimal d'affectations sera toujours $n + m$, ce qui correspond à notre algorithme.

Exercice 2.3.
Écrire les algorithmes de tri suivants, démontrer leur correction, et donner leur complexité en temps, en pire cas puis en meilleur cas. On considérera que la donnée initiale est un tableau, on utilisera une fonction de comparaison comme dans la première planche de TD.

1. *Tri à bulle.*
2. *Tri insertion.*
3. *Tri fusion.*
4. *Ces tris sont-ils stables ? En place ?*
5. *Parmi les algorithmes précédents lesquels sont écrit en récursif, en itératif ? Pouvez-vous facilement donner une version itérative de ceux qui sont récursif et une version récursive de ceux qui sont itératifs ?*
6. *Écrire les arbres de décisions associés à chacun de vos algorithmes pour des tableaux de trois éléments.*

Correction 2.3.
La plupart des solutions se trouvent dans le cours qui précède.

Exercice 2.4 (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, $\Theta(n)$, en fonction de 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 ?*

Correction 2.4.

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 `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 `aux`. Pour cela il suffit de montrer que la somme des éléments du tableau `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 `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 `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.

Exercice 2.5 (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 équillibrées de s est 6. Les suites aabab 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 complétement l'algorithme.*

Correction 2.5.
Pas de correction.