1 Généralités

1.1 Notations préliminaires

On définit les relations de comparaison asymptotique suivantes entre fonctions au voisinage de l'infini :
- $f(n) = \bigcirc(g(n))$ s'il existe $M \geqslant 0$ tel que $|f(n)| \leqslant M|g(n)|$ pour tout n suffisamment grand,
- $f(n) = \Omega(g(n))$ s'il existe $m > 0$ tel que $|f(n)| \geqslant m|g(n)|$ pour tout n suffisamment grand,
- $f(n) = \Theta(g(n))$ si les deux conditions précédentes sont réalisées.

Attention, seule la notation $\bigcirc$ est officiellement au programme.

1.2 Introduction

Pour évaluer l'efficacité d'un algorithme, on introduit des mesures numériques. Elles sont de deux natures :
- *temporelle* : évaluer l'ordre de grandeur du «temps» d'éxécution d'un algorithme. On l'évalue au moyen de l'ordre de grandeur du nombre d'opérations élémentaires, en fonction de la taille des données, nécessaires pour mener à bien l'algorithme. Il est fondamental de préciser l'opération élémentaire en question : comparaison ($\leqslant$), affectation, opération arithmétique...

  Cela dépend bien sûr de la machine utilisée et aussi des données. On exprime cette vitesse par comparaison avec des suites usuelles : un temps d'exécution en $\Theta(n)$ est un temps d'éxécution asymptotiquement linéaire par rapport aux données.
- *spatiale :* évaluer l'ordre de grandeur du nombre de cases mémoires accédées.

Au temps, pas si lointain, où la mémoire des ordinateurs coutait très cher, l'objectif d'une bonne programmation était de réduire la complexité spatiale. Il semblerait que la complexité temporelle soit dorénavant le point sensible.
Ces deux notions sont liées : sur des machines séquentielles (un seul processeur qui exécute les instructions de manière séquentielle) une inscription en mémoire nécessitant au moins une unité de temps, la complexité temporelle est au moins égale à la complexité spatiale.
Ce n'est plus vrai pour des machines parallèles.

Remarque(s). L'étude de la complexité d'un algorithme indique un comportement **asymptotique**. Il arrive qu'un algorithme naïf soit plus rapide sur des données de petite taille qu'un algorithme subtil.

Définition 1.2.1 *Soit $t(n)$ le temps d'exécution d'un algorithme pour une donnée de taille n. Soit f une fonction à valeurs $\mathbb{R}_+^*$. On dit que l'algorithme a une complexité temporelle en $\bigcirc(f(n))$ (resp. $\Theta(f(n))$) si $t(n) = \bigcirc(f(n))$ (resp $= \Theta(f(n))$).*
On dit que l'algorithme est :
- *logarithmique si $t(n) = \Theta(log_2(n))$*
- *linéaire si $t(n) = \Theta(n)$*
- *polynomial si $t(n) = \Theta(n^k)$ $\qquad (k \geqslant 2)$*
- *exponentiel si $t(n) = \Theta(r^n)$, pour un $r > 1$.*

Remarque(s). ATTENTION, les informaticiens pêchent souvent par manque de rigueur mathématique et ils confondent souvent Θ et $\bigcirc$!! (en plus c'est HP alors...).

1.3 Exemples numériques

On considère une machine qui serait capable d'effectuer 10^8 opérations élémentaires par seconde. Le tableau suivant indique le temps d'exécution suivant la complexité :

taille \ compl	$\log_2(n)$	n	$n\log_2(n)$	n^2	2^n	10^n
$n = 10$	2×10^{-8}sec	10^{-7}sec	3×10^{-7}sec	10^{-6}sec	10^{-5}sec	100sec
$n = 20$	3×10^{-8}sec	2×10^{-7}sec	8×10^{-7}sec	4×10^{-6}sec	10^{-2}sec	31709ans
$n = 50$	4×10^{-8}sec	5×10^{-7}sec	3×10^{-6}sec	3×10^{-5}sec	100jours	
$n = 1000$	7×10^{-8}sec	10^{-5}sec	10^{-4}sec	10^{-2}sec		
$n = 100000$	2×10^{-7}sec	10^{-3}sec	$0,016$sec	100 sec		
$n = 1000000$	$2,6 \times 10^{-7}$sec	10^{-2}sec	$0,2$sec	$2,8$ heures		

Le tableau suivant indique, dans les mêmes conditions de machine, suivant l'algorithme, la taille des données traitées en un temps donné.

temps \ compl	$\log_2(n)$	n	$n\log_2(n)$	n^2	n^3	2^n
1 $\mu-$seconde	$> 10^{10000000}$	10^5	77×10^5	316	46	16
1 seconde	$> 10^{10000000}$	10^8	$4,5 \times 10^6$	10^4	464	26
1 minute	$> 10^{10000000}$	6×10^9	2×10^8	$7,7 \times 10^4$	$1,8 \times 10^3$	32
1 heure	$> 10^{10000000}$	36×10^{10}	10^{10}	6×10^5	$7,1 \times 10^3$	38
1 an	$> 10^{10000000}$	3×10^{15}	7×10^{13}	6×10^7	$1,4 \times 10^5$	51

On remarque que les algorithmes logarithmiques, linéaires, en $n\ln(n)$ sont utilisables pour des données de grande taille.
Les algorithmes en n^k ne sont raisonnables que pour $k < 2$. Pour $2 \leqslant k \leqslant 3$ ils peuvent permettre de traiter des algorithmes de taille moyenne. Pour $k > 3$ uniquement de petits problèmes (quand on multiplie la taille des données par 10 le temps est multiplié par 1000 pour un algorithme en n^3 !!).
Étudiez l'évolution du temps de calcul pour une multiplication par 10 de la taille des données dans les cas d'algorithmes logarithmiques, linéaires, en $n\ln(n)$, polynomiaux ou exponentiels... : quel que soit l'augmentation des performances des micro-processeurs, la recherche des algorithmes performants reste d'actualité.

1.4 Complexité dans le pire des cas, en moyenne

Pour évaluer la complexité d'un algorithme on étudie les complexité dans le pire des cas, ou en moyenne.
Elles donnent des renseignements différents sur l'efficacité de l'algorithme : une mauvaise complexité dans le pire des cas donnera un algorithme qui risque d'être trop long pour des données défavorables, même si la complexité en moyenne est favorable.
Toutefois c'est la complexité en moyenne, plus lourde à calculer (dans de nombreux problème on ne sait pas la calculer) qui renseigne le mieux. D'autant que son calcul donne souvent des indications sur la fréquence des cas défavorables. Ceci dit, le calcul nécessite fréquemment une répartition uniforme des données et ce n'est souvent pas le cas en pratique...

Définition 1.4.1 *Si $t(d)$ représente, pour l'algorithme A, la complexité en temps sur la donnée d, et D_n l'ensemble des donnée de taille n*

- $\max_{d\in D_n}(t(d))$ *est appelé complexité dans le pire des cas.*
- $\min_{d\in D_n}(t(d))$ *est appelé complexité dans le meilleur des cas.*
- $\sum_{d\in D_n} p(d)t(d)$, *où $p(d)$ est la probabilité d'obtenir d en entrée de l'algorithme, est la complexité en moyenne.*
  En particulier si toutes les données sont équiprobables, c'est $\frac{1}{|D_n|}\sum_{d\in D_n} t(d)$

Il est immédiat que la complexité en moyenne est comprise entre la complexité dans le meilleur et le pire des cas.

1.5 Exemples

- **Tours de Hanoï**

L'algorithme récursif des tours de Hanoï fournit rapidement une formule de récurrence pour sa complexité. Rappel de l'algorithme :
Déplacer n disques de A vers B en se servant de l'intermédiaire C revient à :

- Déplacer les $n-1$ disques supérieurs de A vers C, avec l'intermédiaire B.
- Mettre le disque du bas de A en B
- Ramener les $n-1$ disques de C vers B, avec l'intermédiaire A

L'unité de complexité est ici le déplacement d'un disque.

• Recherche linéaire

Considérons un algorithme qui recherche si, dans un vecteur de taille n, pris parmi p éléments, figure l'élément x.
Le premier algorithme (vraiment très naïf !) consiste à parcourir le vecteur en entier. Les complexités en moyenne et au pire sont en $\Theta(n)$.
Le deuxième parcourt le vecteur et s'arrête à la première occurrence de x. La complexité dans le cas le pire est inchangée.

• Produit

On considère les deux programmes Caml :

```
let prod1(v)=
 let p=ref(1.0) in
 for i=0 to (vect_length(v)-1) do p:=!p*.v.(i) done;
 !p ;;

let prod2(v)=
 let p=ref(1.0) and i=ref(0) in
 while (!i<vect_length(v)) & (!p<>0.0) do
 p:=!p*.v.(!i); i:=!i+1 done;
 !p ;;
```

- Commençons par compter les opérations en considérant (à tort !) qu'elles prennent toutes une unité de temps : dans **prod1** il y a 4 opérations élémentaires dans le corps de la boucle (incrémentation de i, accès à v.(i), multiplication et affectation du résultat à p). Pour une donnée d de taille n, la complexité (aussi bien dans le pire des cas qu'en moyenne) est en $t(d) = 4n+2$, pour des données de taille n.
  dans **prod2** il y a 6 opérations élémentaires dans le corps de la boucle (les tests en plus). La complexité dans le pire des cas est en $t_p(d) = 6n+3$, pour des données de taille n.
  Dans le pire des cas, les deux complexités sont en $\Theta(n)$, (le premier cas étant plus favorable).
  Pour une donnée d de taille n, qui comporte un zéro en position m, la complexité est en $t(d) = 6m+5$: L'algorithme **prod2** est donc plus favorable s'il y a un zéro dans les 2 premiers tiers du vecteur et moins sinon.
- Examinons la complexité en moyenne :
  On va considérer que les éléments de v sont des entiers pris de manière aléatoire dans $[\![0, k-1]\!]$, le tirage pour chaque composante étant équiprobable.
  Il y a k^n vecteurs possibles, chacun d'entre eux étant associé à la même probabilité $\dfrac{1}{k^n}$.
  Soit $A_{p,n}$ l'ensemble de vecteurs de D_n pour lesquels 0 apparaît pour la première fois au rang p. $(0 \leqslant p \leqslant n-1)$ et A les vecteurs sans zéro. On a ainsi une partition de D_n et la complexité en moyenne vaut

$$C_n = \frac{1}{k^n}\left(\sum_{p=0}^{n-1} |A_{p,n}| t(d_p) + |A| t(d_A)\right)$$

  où $t(d_p)$ est le temps de calcul d'un élément quelconque de $A_{p,n}$, soit $t(d_p) = 6p+5$ et $t(d_A) = 6n+3$.
  D'autre part $|A_{p,n}| = (k-1)^p \times k^{n-p-1}$ et $|A| = (k-1)^n$.
  Donc $C_n = \dfrac{1}{k^n}\left(\sum_{p=0}^{n-1}(k-1)^p \times k^{n-p-1}(6p+5) + (k-1)^n(6n+3)\right) = \dfrac{1}{k}\left(\sum_{p=0}^{n-1}\left(\dfrac{k-1}{k}\right)^p(6p+5) + \left(\dfrac{k-1}{k}\right)^n(6n+3)\right)$
  On a donc $\lim_{n\mapsto\infty} C_n = \dfrac{1}{k}\sum_{p=0}^{\infty}\alpha^p(6p+5) = 6k+5$, (avec $\alpha = \dfrac{k-1}{k}$).

La complexité en moyenne de **prod2** est donc **bornée** tandis que celle de **prod1** est en $\bigcirc(n)$
[On a utilisé $\sum_{k=0}^{\infty} \alpha^k = \dfrac{1}{1-\alpha}$ et $\sum_{k=0}^{\infty} k\alpha^{k-1} = \dfrac{1}{(1-\alpha)^2}$]

2 Études de récurrences usuelles

2.1 $\mathbf{t(n) = at(n-1) + f(n)}$, $\mathbf{a \in \mathbb{N}^*}$

C'est le cas d'un algorithme pour lequel la résolution pour un système de données de taille n se ramène à a sous-problèmes de taille $n-1$. $f(n)$ représentant le temps nécessaire pour découper puis recomposer le problème initial en sous problèmes.
C'est le cas (plus haut) des tours de Hanoï (a=2) et (plus bas) du tri par selection.

- **le cas a=1**

Par récurrence on a $t(n) = t(0) + \sum_{k=1}^{n} f(k)$. Il faut donc estimer asymptotiquement $\sum_{k=1}^{n} f(k)$.
Pour cela on pourra essayer d'utiliser les comparaisons des sommes partielles des séries à termes positifs divergentes (ou Cesaro). Ceci permet de remplacer, éventuellement, par équivalent, $\sum_{k=1}^{n} f(k)$, par une somme plus simple à évaluer.
Par exemple, si $f(k) \sim \alpha k^p$ ($\alpha > 0$), alors $\sum_{k=1}^{n} f(k) \sim \alpha \sum_{k=1}^{n} k^p$. On peut alors utiliser le résultat du cours de maths :

Proposition 2.1.1 $\sum_{k=1}^{n} k^p \underset{\infty}{\sim} \dfrac{n^{p+1}}{p+1}$, *pour* $p \in \mathbb{N}$

Preuve. En utilisant la croissance de $x \mapsto x^p$, on a $x \in [\![k, k+1]\!]$, $(x-1)^p \leqslant x^p \leqslant (k+1)^p$. On intègre entre k et $k+1$:

$$\int_{k-1}^{k} x^p \leqslant k^p \leqslant \int_{k}^{k+1} x^p \Longrightarrow \int_{0}^{n} x^p \leqslant \sum_{k=1}^{n} k^p \leqslant \int_{1}^{n+1} x^p$$

d'où l'équivalent annoncé. □
d'où

Proposition 2.1.2 *Si* $a = 1$ *et* $f(n) = \Theta(n^p)$, *les solutions de* (1) *vérifient* $t(n) = \Theta(n^{p+1})$.

- **a>1**

On peut poser $u(n) = \dfrac{t(n)}{a^n}$: on a alors $u(n) = u(n-1) + \dfrac{f(n)}{a^n}$, ce qui permet d'utiliser le résultat précédent. On a $t(n) = a^n(t(0) + \sum_{k=1}^{n} \dfrac{f(k)}{a^k})$. La complexité est au moins exponentielle.

Proposition 2.1.3 *Si* $a \geqslant 2$ *et si la série* $\sum_{k=1}^{n} \dfrac{f(k)}{a^k}$ *converge alors* $t(n) = \Theta(a^n)$.

2.2 Diviser pour régner

Le calcul de la complexité pour des algorithmes du type « diviser pour régner » conduit à des équations de récurrence de la forme :

$$(E) \qquad t(n) = at(\lfloor n/2 \rfloor) + bt(\lceil n/2 \rceil) + h(n) \qquad (n \geqslant 2)$$

où $(a, b) \in \mathbb{N}^2, a + b \geqslant 1$ et h une fonction de $\mathbb{N}^*$ dans lui-même.
Les premiers termes correspondent au calcul des complexités des 2 sous problèmes et $h(n) = p(n) + f(n)$ où $p(n)$ est le temps nécessaire pour la partition d'un problème de taille n et $f(n)$ celui nécessaire pour la fusion de deux objets de taille $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$. Le plus souvent $a = b = 1$ (on applique l'algorithme à chaque moitié une fois et une seule) ou $a + b = 1$ (une et une seule moitié se voit appliquer l'algorithme).

On s'intéresse aux cas où $h(n) \sim \gamma \times n^{\beta}$ ($\beta > 0$). Commençons par un résultat fort utile :

- **Monotonie de t**

Proposition 2.2.1 *Si t vérifie (E) et si h est croissante, alors t est croissante.*

Preuve. Se démontre par récurrence.
Posons H_n : t est croissante sur $[\![1, n]\!]$.
H_1 est vraie, ainsi que H_2, puisque $a + b \geqslant 1$.
Supposons H_{n-1} vraie et $n \geqslant 3$. Il suffit de vérifier $t(n-1) \leqslant t(n)$ pour en déduire H_n.
On a $t(n) = at\lfloor n/2\rfloor + bt\lceil n/2\rceil + h(n)$ et $t(n-1) = at\lfloor (n-1)/2\rfloor + bt\lceil (n-1)/2\rceil + h(n-1)$
$1 \leqslant \lfloor (n-1)/2\rfloor \leqslant \lfloor n/2\rfloor < n$ et $1 \leqslant \lceil (n-1)/2\rceil \leqslant \lceil n/2\rceil < n$ et donc H_{n-1} s'applique. Comme d'autre part h est croissante, on déduit $t(n-1) \leqslant t(n)$. □

Cas a=b=1

- **Cas où n est une puissance de 2**

On supposera pour l'instant que $n = 2^p$, avec $a = b = 1$. Ce qui conduit à une équation de récurrence :

$$t(2^p) = 2t(2^{p-1}) + h(2^p)$$

Posons $u_p = \dfrac{t(2^p)}{2^p}$.On a $u_p = u_{p-1} + \dfrac{h(2^p)}{2^p}$ et donc $u_p = u_0 + \sum\limits_{k=1}^{p} \dfrac{h(2^k)}{2^k}$.

- Si $h(n) \sim \gamma n$ [c'est à dire le temps nécessaire à la partition et à la fusion est proportionnel à la taille des données] (respectivement $h(n) = \bigcirc(n)$), alors $\sum\limits_{k=1}^{p} \dfrac{h(2^k)}{2^k} \sim \gamma p$ (voir cas $a = 1$) et donc $u_p \sim \gamma p$ (respectivement $u_p = \bigcirc(p)$). D'où $t(2^p) \sim \gamma p 2^p$ (respectivement $t(2^p) = \bigcirc(p2^p)$) soit $\underline{t(n) = \Theta(n \log_2(n))}$ (resp. $\bigcirc(n \log_2(n))$).
- Si $h(n) \sim \gamma n^{\beta}$ ($\beta > 1$) (respectivement $h(n) = \bigcirc(n^{\beta})$) alors $\dfrac{h(2^k)}{2^k} \sim \gamma 2^{(\beta-1)k}$ et donc :
  $\sum\limits_{k=1}^{p} \dfrac{h(2^k)}{2^k} \sim \gamma \sum\limits_{k=1}^{p} 2^{(\beta-1)k} = \gamma \times 2^{\beta-1} \times \dfrac{2^{(\beta-1)p} - 1}{2^{\beta-1} - 1}$
  En posant $\delta = \dfrac{\gamma \times 2^{(\beta-1)}}{2^{\beta-1} - 1}$, on a donc $u_p \sim \delta \times 2^{(\beta-1)p}$ puis $t(2^p) \sim \delta \times 2^{\beta p}$ (respectivement $t(2^p) = \bigcirc(2^{\beta p})$) soit $\underline{t(n) = \Theta(n^{\beta})}$ (resp. $\bigcirc(n^{\beta})$).

- **Cas où n n'est pas une puissance de 2**

On suppose toujours que $a = b = 1$ et on suppose, d'autre part que $h(n) = \gamma n^{\beta}$

$$t(n) = at\lfloor n/2\rfloor + bt\lceil n/2\rceil + h(n)$$

Il existe k tel que $2^{k-1} \leqslant n < 2^k$ et donc (proposition 2.2.1), $t(2^{k-1}) \leqslant t(n) \leqslant t(2^k)$.
D'après l'encadrement de n, $\dfrac{t(2^{k-1})}{2^{k\beta}} \leqslant \dfrac{t(n)}{n^{\beta}} \leqslant \dfrac{t(2^k)}{2^{(k-1)\beta}}$.

- Si $\beta = 1$: on a $t(2^k) \sim \gamma k 2^k$ et comme $k = [\log_2 n]$, on a $k \sim \log_2 n$. On en déduit que $\underline{t(n) = \Theta(n \log_2(n))}$
- $\beta > 1$: par la même méthode d'encadrement, on montre $\underline{t(n) = \Theta(n^{\beta})}$

On a donc :

Proposition 2.2.2 *On considère un algorithme « diviser pour régner » pour lequel :*

$$t(n) = t(\lfloor n/2\rfloor) + t(\lceil n/2\rceil) + h(n)$$

Si le temps de partition et de fusion (représenté par h) est un $\Theta(n^{\alpha})$, alors le temps de calcul est

- $\Theta(n \log_2 n)$ *si* $\alpha = 1$

– $\Theta(n^\alpha)$ *si* $\alpha > 1$

Cas général : a et b quelconques

Le cas $a = b = 1$, représente le cas où chaque moitié se voit appliquer l'algorithme une fois et une seule.
Si $a + b = 1$, une et une seule des deux moitiés se voit appliquer l'algorithme. Nous étudions ici $a + b$ quelconque dans $\mathbb{N}^*$.

- **Cas où n est une puissance de 2**

La formule de récurrence devient : $t(2^p) = (a + b)t(2^{p-1}) + h(2^p)$. Posons $a + b = 2^\alpha$ et $u_p = \dfrac{t(2^p)}{2^{\alpha p}}$, la suite (u_p) vérifie la relation

$$u_p = u_{p-1} + \frac{h(2^p)}{2^{\alpha p}}$$

et donc $u_p = u_0 + \sum\limits_{k=1}^{p} \dfrac{h(2^k)}{2^{\alpha k}}$.

- Si $h(n) = \bigcirc(n^\beta)$, pour $\beta < \alpha$ alors $\sum \dfrac{h(2^k)}{2^{\alpha k}}$ converge et donc $u_p = \bigcirc(1)$ et $t(2^p) = \bigcirc(2^{\alpha p})$ (idem avec les Θ).
- Si $h(n) \sim \gamma n^\alpha$ alors $\sum\limits_{k=1}^{p} \dfrac{h(2^k)}{2^{\alpha k}} \sim \gamma p$ et donc $t(2^p) \sim \gamma p 2^{\alpha p}$
- Si $h(n) \sim \gamma n^\beta$, pour $\beta > \alpha$, alors $\sum\limits_{k=1}^{p} \dfrac{h(2^k)}{2^{\alpha k}} \sim \gamma \sum\limits_{k=1}^{p} \dfrac{2^{\beta k}}{2^{\alpha k}} \sim \gamma 2^{(\beta-\alpha)p} \dfrac{2^\beta}{2^\beta - 2^\alpha}$ et $t(2^p) \sim \delta 2^{\beta p}$ avec $\delta = \gamma \dfrac{2^\beta}{2^\beta - 2^\alpha}$

- **Cas où n n'est pas une puissance de 2**

On procède comme précédemment par encadrement :

Proposition 2.2.3 *Pour une méthode diviser pour régner qui conduit à une équation*

$$(E) \qquad t(n) = at(\lfloor n/2 \rfloor) + bt(\lceil n/2 \rceil) + h(n) \qquad (n \geqslant 2)$$

où $(a, b) \in \mathbb{N}^2, a + b \geqslant 1$ *et* $h = \bigcirc(n^\beta)$ *(resp.* $\Theta(\cdot)$*) ce qui correspond à résoudre* $q = a + b$ *problèmes de taille* $n/2$*, si* $a + b = 2^\alpha$*,*

- *Si* $\beta < \alpha$*, alors* $t(n) = \bigcirc(n^\alpha)$
- *Si* $\alpha = \beta$*,* $t(n) = \bigcirc(n^\alpha \log_2 n)$
- *Si* $\beta > \alpha$*,* $t(n) = \bigcirc(n^\beta)$

(resp. $\Theta(\cdot)$*).*

Remarque(s). Il vaut bien mieux savoir retrouver les résultats précédents sur un exemple, que d'apprendre par cœur le théorème !

3 Complexité des algorithmes de tris

Nous allons examiner la complexité d'algorithmes de tris. Pour cela rappelons les différentes méthodes classiques qui opèrent sur une donnée de taille n placée dans un vecteur ou dans une liste.

3.1 Différentes méthodes de tri par comparaison

Pour chacune des méthodes de tri, on propose une implémentation Caml en utilisant comme structure de données sous-jacente tantot des listes tantot des vecteurs pour la commodité de la présentation. Vous compléterez les programmes manquants...

- **tri par sélection**

Ce tri consiste à chercher le plus petit élément du vecteur et à l'échanger avec le premier, puis l'élément supérieur qu'on échange avec le deuxième et ainsi de suite...

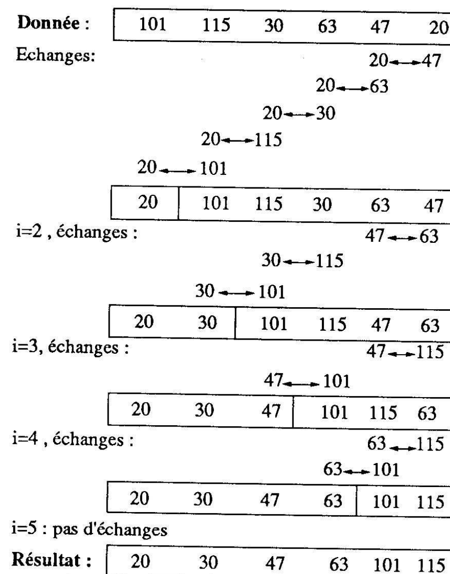
```
let echange i j v =
 let t=v.(i) in
 begin
  v.(i) <- v.(j);
  v.(j) <- t
 end;;

let minimum a  v =
 let b = (vect_length v)-1 in
 let mini=ref a in
 for i=a to b  do
 if v.(i) < v.(!mini) then mini:=i  done;
 !mini;;

let tri_selection_vect v =
 let N = vect_length v and minimum_temporaire = ref 0 in
 for i=0 to N-1 do
 minimum_temporaire := minimum i v;
 echange !minimum_temporaire i v
 done;
 v;;
```

- **tri à bulles**

Il consiste à parcourir le vecteur en commençant par la fin et à échanger les deux premiers éléments consécutifs qui ne sont pas dans le bon ordre, et à recommencer tant qu'on peut le faire. À la fin de l'exécutionde `une_passe_vect`, le minimum du vecteur se trouve en tête.
Les petits éléments remontent, comme des bulles vers le début du vecteur. On verra que ce n'est pas un bon algorithme, par l'étude de complexité.
On propose ici une amélioration afin d'éviter les parcours sans échange.




```
(* tri bulle classique !*)
let une_passe_vect fin v =
 for j=1 to fin do
 if v.(j-1) > v.(j) then echange (j-1) j v
 done ;;

let tri_bulle_vect v =
 for i= (vect_length v)-1 downto 0 do
 une_passe_vect i v
 done;;
```

```
(* tri bulle deuxieme version : pour s'arr\^{e}ter d\'{e}s que possible !*)
let une_passe_vect2 fin v =
 let a_t_on_fait_qqch = ref false in
 for j=1 to fin do
 if v.(j-1) > v.(j) then
  begin echange (j-1) j v;
  a_t_on_fait_qqch := true end
 done;
 !a_t_on_fait_qqch;;

let tri_bulle_vect2 v =
 let i = ref ((vect_length v)-1) in
 while (!i>=0)  & (une_passe_vect2 !i v) do
 i:=!i-1
 done;;
```

• tri par insertion

C'est le tri du joueur de carte qui range ses cartes au fur et à mesure en insérant chaque nouvelle carte dans celles déjà triées...
Il existe plusieurs variantes :

- si l'on choisit comme structure de données sous jacente des listes, l'insertion du $(i+1)$-ième élément e dans la liste ℓ des i premiers éléments qui est déjà triée se fait en comparant e avec le début de ℓ puis le cas échéant en itérant cette opération sur le reste de ℓ. Ici on comptera le nombre de comparaisons et le nombre d'appels à l'opérateur : : (« cons »).
- si l'on choisit comme structure de données sous jacente des vecteurs, la version classique de l'algorithme consiste à considérer, les i premiers éléments étant déjà triés, le $(i+1)$-ième élément que l'on compare au i-ième puis au précédent, en décalant au fur et à mesure les éléments vers la droite tant qu'ils sont plus grands. On s'arrête dès que l'on trouve un élément inférieur ou égal. **Attention** aux problèmes de débordement : si l'élément testé est plus petit que tous les précédents, on pourra tester la place de l'élément comparé et s'arrêter éventuellement au premier, ou prévoir une « sentinelle » en premier terme $(a(0))$ dont on sait qu'elle est inférieure à tous les éléments du vecteur et qu'elle arrêtera ainsi les tests.

  En profitant de la structure vectorielle, on peut « pratiquer » le tri par *insertion dichotomique* : la place du $(i+1)$-ième élément est recherchée en le comparant d'abord à l'élément figurant au milieu de ceux qui sont déjà triés, puis dans la moitié droite ou dans la moitié gauche. Si la recherche continue dans la moitié gauche, on peut déjà décaler les éléments de la partie droite d'une place vers la droite.

```
let rec insere element = function
 [] -> [element] |
 x::reste -> if element <= x then element::x::reste else x::(insere element reste);;

let rec tri_insertion = function
 [] -> [] |
 x::reste -> insere x (tri_insertion reste);;
```

Version vecteur :

```
let insere_element v i =
 let j=ref i and e=v.(i) in
 while (!j>0) & (v.(!j-1) > e) do
 v.(!j) <- v.(!j-1) ;  j:=!j-1
                              done;
 v.(!j) <-e;;

let tri_insertion_vect v =
 for i=1 to (vect_length v)-1 do
 insere_element v i
 done;;
```

• tri fusion

C'est un algorithme du type «diviser pour régner».
Étant donné un vecteur de taille n, on le sépare en deux vecteurs, approximativement de taille $n/2$, qu'on trie récursivement et que l'on fusionne ensuite.
Illustrons la procédure de fusion...
Un enseignant récupère les fiches triées alphabétiquement des élèves suivant l'option info en MPSI1 et MPSI2, passant en spé, et souhaite les trier en un seul paquet de fiches : il met les 2 paquets devant lui et prend à chaque étape la première fiche dans l'ordre alphabétique, au sommet des 2 paquets, jusqu'à épuisement de l'un des deux.

```
let rec divise = function
  |[]->([],[])
  |[e]->([e],[])
  |a::b::r->let (m1,m2)=divise r in (a::m1,b::m2);;
let rec fusion = fun
  |l []->l
  |[] l->l
  |(a::r as l1) (b::s as l2)->if a<b then a::(fusion r l2) else b::(fusion l1 s);;
let rec tri_fusion = fun
  |[]->[]
  |[e]->[e]
  |l->let (m1,m2) = divise l in fusion (tri_fusion m1) (tri_fusion m2);;
```

• tri rapide

C'est une démarche proche de celle du tri fusion. Dans le tri fusion la partition ne nécessite aucun test de comparaison (avec des vecteurs, on coupe immédiatement en deux), le travail est dans la fusion. Tandis que pour le tri rapide, c'est la partition qui est cruciale, la réunion étant immédiate.
L'idée est la suivante : on transfère un élément à sa place définitive et on recommence l'algorithme sur les deux sous listes obtenues. Précisons.
Dans le cas d'une liste, le programme se lit tout seul.

```
let rec partition (e:int)  = function
  | [] -> ([], [])
  | d::r -> let l1,l2 = partition e r in
   if (d < e) then (d::l1,l2) else (l1,d::l2);;

let rec tri_rapide_liste  = function
  |  []    -> []
  |  [e]   -> [e]
  |  e::r  -> let l1,l2 = partition  e r in
             (tri_rapide_liste l1)@(e::(tri_rapide_liste l2));;
```

Pour les amoureux des vecteurs (et pour eux seulement!) : on trie un sous-vecteur des éléments d'indice g jusqu'a d (g<d). Le pivot est situe au «milieu» de ce vecteur. L'indice k part de g et progresse vers la droite tant que l'élément rencontré est strictement inférieur au pivot. L'indice l part de d et progresse vers la gauche tant que l'élément rencontré est strictement supérieur au pivot (ces deux conditions permettent de s'assurer que chacun des deux s'arrêtera au plus tard sur le pivot lui-même, qui appartient au vecteur et évite donc le controle de la valeur des indices). Si k <= l, on échange (s'il y a lieu) les éléments correspondants et on fait encore varier les indices d'une unité avant de faire le test de continuation de l'application de l'algorithme. On arrête lorsque k > l. Il reste à trier les sous-vecteurs de gauche (indices g jusqu'à l) et de droite (indices k jusqu'à d).
Au départ, évidemment, g = 0 et d = n-1.

```
let quick_sort_vec a =
    qs_vec_rec 0 (vect_length a - 1 )
        where rec qs_vec_rec g d =
            if g < d then
                let pivot = a.((g+d)/2) and k = ref g and l = ref d in
```

```
                    while !k <= !l do
                        while a.(!k) < pivot do
                            k := !k + 1
                        done ;
                        while a.(!l) > pivot do
                            l := !l - 1
                        done ;
                        if !k <= !l then (
                            if !k < !l then (
                                let ech = a.(!k) in
                                    a.(!k) <- a.(!l) ;
                                    a.(!l) <- ech
                            ) ;
                            k := !k + 1 ;
                            l := !l - 1
                        )
                    done ;
                    qs_vec_rec g !l ;
                    qs_vec_rec !k d
;;
```

- **tri en tas**

Voir chapitre sur les arbres.

3.2 Complexité

- **les tris par sélection, insertion, à bulle**

La donnée considérée est de taille n. Suivant les cas, il faudra remplacer « échange » par « affectation » ou « cons »...

Proposition 3.2.1 – *Le tri par sélection procède à* $\dfrac{n(n-1)}{2}$ *(soit équivalent à* $(n^2/2)$*) comparaisons et* $n-1$ *(soit équivalent à* (n)*) échanges*

– *Pour le tri à bulle classique, le nombre de comparaisons est équivalent à* $(n^2/2)$ *et le nombre d'échanges est équivalent à* $n^2/2$ *dans le pire des cas et à* $n^2/4$ *dans le cas moyen.*

– *Pour le tri par insertion le nombre de comparaisons et le nombre d'échanges sont équivalents à* $n^2/2$ *dans le pire des cas et à* $n^2/4$ *dans le cas moyen.*

Preuve.

- Pour le tri par sélection, la recherche du minimum dans une donnée de longueur p nécessite $p-1$ comparaisons. La boucle est exécutée N fois et entraîne à chaque fois un échange et une recherche de minimum sur une donnée de longueur $N-i+1$ lors de sa i-ième exécution. Ce qui donne exactement $\dfrac{n(n-1)}{2}$ comparaisons et $n-1$ échanges dans tous les cas.
  Remarquons que le tri par selection est progressif. A l'étape i de la boucle les i premiers éléments du vecteur sont triés.
- Pour le tri à bulle, pour tout jeu de données, on procède comme pour le tri par sélection, en ce qui concerne les comparaisons. Ce qui donne exactement $\dfrac{n(n-1)}{2}$ comparaisons.
  En ce qui concerne les échanges, le cas le plus défavorable est celui d'un vecteur dans l'ordre inverse : chaque comparaison donne lieu à un échange soit $\dfrac{n(n-1)}{2}$ échanges. Le cas le plus favorable est celui du vecteur déjà trié : pas d'échange. Pour la complexité en moyenne considérons uniquement les vecteurs pour lesquels toutes les valeurs sont différentes : tout tableau de valeurs peut être par exemple représenté par une permutation de $\{1, 2, \cdots, n\}$.

Tous les vecteurs étant considérés comme équiprobables la moyenne du nombre d'échanges est égale à $\frac{1}{n!}\sum_{\sigma\in S_n} nb_ech(\sigma)$.

Considérons pour chaque vecteur t de taille n le vecteur t' miroir $(t[1]=t'[n],\cdots,t[n]=t'[1])$, et $m : t \longmapsto t'$
Si on exécute l'algorithme sur t, deux éléments ($t(i)$ et $t(j)$, $i<j$) sont échangés (une fois et une seule au cours de l'algorithme) si et seulement si $t[i]>t[j]$. Toute paire d'éléments est donc échangée, soit dans t, soit dans t'. Pour tout $(t,m(t))$, il y a donc $\frac{n(n-1)}{2}$ échanges.
Soit T l'ensemble des vecteurs et $A=\{t|t[1]<t[n]\}$. $(A,m(A))$ est évidemment une partition de T, m est une bijection de A sur $m(A)$ et donc $|A|=\frac{n!}{2}$

$$\frac{1}{n!}\sum_{t\in T} nb_ech(t)=\frac{1}{n!}(\sum_{t\in A} nb_ech(t))+\sum_{t\in m(A)} nb_ech(t))=\frac{1}{n!}\sum_{t\in A} nb_ech(t)+nb_ech(m(t))$$

$\frac{1}{n!}\sum_{t\in T} nb_ech(t)=\frac{1}{n!}\times\frac{n!}{2}\times\frac{n(n-1)}{2}=\frac{n(n-1)}{4}$.
On vient en fait de calculer le nombre moyen d'inversions dans une permutation ; il est exactement égal au nombre moyen d'échanges du tri bulle car chaque échange entre deux éléments consécutifs de la permutation supprime exactement une inversion.

- L'insertion dans une liste déjà triée n'impose pas son parcours complet contrairement à une recherche de minimum. Le tri par insertion devrait donc mieux se comporter que le tri par sélection.

  Pour le tri par insertion classique version vecteur, le cas favorable correspond à une liste déjà triée et le cas le pire à une liste inversée (c'est l'inverse pour la version liste). Ceci donne $n^2/2$ comparaisons et échanges.

  Pour l'analyse en moyenne, on constate que le nombre de comparaisons pour insérer le i-ième élément dans la liste triée des i premiers éléments correspond au nombre d'éléments (+1) de la permutation d'origine d'indice inférieur à i et de valeur plus grande soit au total au nombres d'inversions (+ n-1) de la permutation. Le nombre de comparaisons en moyenne est donc à nouveau équivalent à $n^2/4$.

  Dans la boucle while on effectue une comparaison de plus que l'on fait d'affectation (<-). Finalement à nouveau en moyenne ce nombre est équivalent à $n^2/4$.

□

3.3 tri fusion, tri rapide

– Pour le tri fusion, on est ramené au calcul, ci dessus, de la complexité de l'algorithme « diviser pour régner ».
  Pour la fusion, on a un corps de boucle dont la complexité ne dépend pas de l'indice qu'on effectue $n/2$ fois. La complexité de la fusion est en $\Theta(n)$. La partition se fait en temps constant. D'où une complexité partition fusion en $\Theta(n)$ $(h(n)=\Theta(n))$. On a $t(n)=t(\lfloor n/2\rfloor)+t(\lceil n/2\rceil)+h(n)$: on a vu qu'alors $t(n)=\Theta(n\log_2 n)$.
  Par contre le tri fusion n'est pas avantageux en complexité spatiale. Il nécessite n places auxiliaires.

– Pour le tri rapide la situation est défavorable si les deux sous vecteurs sont très déséquilibrés en taille. En particulier si à chaque étape un des 2 sous vecteurs est vide, par exemple si le vecteur initial est déjà trié par ordre croissant ou décroissant. Examinons par exemple un vecteur trié par ordre croissant. Le placement de $a(1)$ (qui ne bougera pas) demande n comparaisons, puis celui de $a(2)$, $n-1$ etc.. on a $\frac{n(n-1)}{2}$ comparaisons.
  Soit $\Theta(n^2)$ comparaisons dans ce cas...
  L'analyse en moyenne en nombre de comparaisons requiert un peu plus d'efforts.
  Nous supposerons ici que l'algorithme de tri rapide est appliqué à un vecteur contenant l'image de l'intervalle discret $[\![1,n]\!]$ par l'une des $n!$ permutations de cet intervalle. Notons C_n le coût moyen :

$$C_n=\frac{1}{n!}\sum\mathcal{C}(s)$$

  où s décrit l'ensemble des $n!$ permutations possibles, et $\mathcal{C}(s)$ est le coût de l'application de l'algorithme de tri rapide à la liste `[s(1);s(2);...;s(n)]`.
  Chaque élément de l'intervalle peut être le premier pivot, avec une probabilité égale à $1/n$. Si l'on note $c_n(k)$ le coût moyen lorsque le premier pivot est égal à k, on a clairement :

$$C_n = \frac{1}{n} \sum_{1 \leqslant k \leqslant n} c_n(k)$$

$c_n(k)$ est la somme du coût du partage (égal à $n-1$) et des coûts moyens des deux tris qui restent à effectuer (car l'hypothèse d'équiprobabilité s'applique à chacun des sous-tableaux issus du partage) ; en convenant que $C_0 = 0$, il vient :

$$c_n(k) = n - 1 + C_{k-1} + C_{n-k}$$

Par sommation, on obtient :

$$C_n = n - 1 + \frac{1}{n} \sum_{1 \leqslant k \leqslant n} (C_{k-1} + C_{n-k})$$

Avec le changement d'indice $k \to n-k$, et les simplifications d'usage :

$$C_n = n - 1 + \frac{2}{n} \sum_{1 \leqslant k < n} C_k$$

On calcule alors

$$nC_n - (n-1)C_{n-1} = [(n-1)n + 2\sum_{k=1}^{n-1} C_k] - [(n-2)(n-1) + 2\sum_{k=1}^{n-2} C_k] = 2(n-1) + 2C_{n-1}$$

Soit

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + 2\frac{n-1}{n(n+1)}$$

On somme alors et comme $2\frac{n-1}{n(n+1)} \sim \frac{2}{n}$ en utilisant le théorème sur l'équivalence des sommes partielles de séries divergentes :

$$\frac{C_n}{n+1} \sim 2 \ln n$$

On en déduit que C_n est équivalent à $2n \ln n$ lorsque n tend vers l'infini. □

Signalons pour finir que le tri rapide reste le plus employé pour sa simplicité et son encombrement mémoire. Il possède de nombreux raffinements : on a intérêt, on vient de le voir, à ce que le pivot par la méthode du tri rapide coupe le vecteur en 2 sous vecteurs les moins déséquilibrés possibles en taille ; par exemple, on peut prendre comme pivot l'élément médian des 3 premiers éléments distincts du vecteur.

4 Optimalité des algorithmes de tri par comparaison

Le résultat fondamental est le suivant :

La complexité, en nombre de comparaisons, aussi bien en moyenne qu'au pire, de tout algorithme de tri qui procède par comparaisons deux à deux des clés des éléments à trier est d'ordre de grandeur supérieur ou égal à $n \lg n$.

Il est obtenu à l'aide de l'arbre binaire de décision d'un algorithme de tri. Il s'agit d'un arbre qui représente toutes les exécutions différentes de l'algorithme sur toutes les données possibles d'une certaine taille. Ici il y a $|\mathfrak{S}_n| = n!$ données possibles.

- Les feuilles de l'arbre indiquent les résultats de ces différentes exécutions (deux exécutions différentes peuvent donner le même résultat).
- Les noeuds internes de l'arbre représentent les opérations de comparaisons entre deux éléments, effectuées par cet algorithme E ; en particulier, la racine correspond à la première comparaison effectuée par E.
- Pour un noeud interne correspondant à la comparaison des éléments x_i et x_j, si la comparaison effectuée entre x_i et x_j a un résultat vrai alors l'exécution de l'algorithme se poursuit dans le sous-arbre gauche de ce noeud et sinon dans son sous-arbre droit.

- A toute exécution de l'algorithme E sur une donnée correspond une branche de l'arbre.
- Le nombre de comparaisons effectuées par E pour une donnée d est donc égal à la longueur de la branche correspondant à l'exécution de E sur d.

La profondeur de l'arbre de décision correspondant à un algorithme donné E est donc le nombre maximum de comparaisons effectuées par E.
Considérons l'ensemble AD de tous les arbres de décision correspondant aux algorithmes de tri par comparaison. Notons $h(A)$, la profondeur d'un arbre A. On obtient que la complexité optimale de la classe dans le pire des cas, notée $M_{max}^{opt}(n)$ vaut :

$$M_{max}^{opt}(n) = inf_{A \in AD} h(A)$$

En conséquence, si on connaît un minorant des profondeurs des arbres de AD, on a un minorant de $M_{max}^{opt}(n)$.
Les arbres de décision qui nous intéressent ont $n!$ feuilles. Leur hauteur est donc $\geqslant \log_2 n! = \Theta(n \ln n)$. On voit par exemple que le tri fusion atteint cette borne dans le pire des cas (comme le tri par tas ou le tri par insertion dichotomique[1]).

[1] Mais attention à la complexité spatiale !