TD Tous

Exercice 1 (Récursivité)**.**

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

Fonction TOTO(n)

si $n = 0$ **alors**
  retourner 1*;*
sinon
  retourner $n \times$ TOTO($n-1$)*;*

```
/* Fonction toto en C */

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

2. *En remarquant que $n^2 = (n-1)^2 + 2n - 1$ écrire une fonction récursive (en C ou en pseudo-code) qui calcule le carré d'un nombre entier postif.*
3. *Écrire une fonction récursive qui calcule le* PGCD *de deux nombres entiers positifs.*
4. *Que calcule la fonction suivante ?*

Fonction TATA(n)

si $n \leq 1$ **alors**
  retourner 0*;*
sinon
  retourner $1 +$ TATA $\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$*;*

```
/* Fonction tata en C */

unsigned int tata(unsigned int n){
    if (n <= 1) return 0;
    return 1 + tata(n / 2);
}
```

Correction 1.

1. Factorielle.
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);
}
```

4. Pour $n > 0$, la fonction TATA(n) calcule $\lfloor \log n \rfloor$ en base 2.

Exercice 2 (Tours de Hanoï)**.**
On se donne trois piquets, p_1, p_2, p_3 et n disques percés de rayons différents enfilés sur les piquets. On s'autorise une seul opération : DÉPLACER-DISQUE(p_i, p_j) *qui déplace le disque du dessus du piquet p_i vers le dessus du piquet p_j. On s'interdit de poser un disque d sur un disque d' si d est plus grand que d'. On suppose que les disques sont tous rangés sur le premier piquet, p_1, par ordre de grandeur avec le plus grand en dessous. On doit déplacer ces n disques vers le troisième piquet p_2. On cherche un algorithme (en pseudo-code ou en C) pour résoudre le problème pour n quelconque.*

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

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

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

Correction 2.

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

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

A	B
C	D

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

Correction 3.

1. Le gain maximum en D est la valeur en D plus le maximum entre : le gain maximum en A ; le gain maximum en B et le gain maximum en A. On peut oublier de considérer le gain en A, puisqu'avec des valeurs non négatives sur chaque case, c'est toujours au moins aussi intéressant, partant de A, de passer par B ou C pour aller en D plutôt que d'y aller directement.
2. On considère que les coordonnées sont données à partir de zéro.

```
int robot_cupide(int x, int y){

    /* Cas de base */
    if ( (x == 0) && (y == 0) ) then return Damier[x][y];

    /* Autres cas particuliers */
    if (x == 0) then return Damier[x][y] + robot_cupide(x, y - 1);
    if (y == 0) then return Damier[x][y] + robot_cupide(x - 1, y);

    /* Cas général x, y > 0 */
    return Damier[x][y] + max(robot_cupide(x - 1, y),
                              robot_cupide(x, y - 1));
}
```

3. Un appel à `robot_cupide(3, 3)` entraînera six appels à `robot_cupide(1, 1)`. Partant de la dernière case (Sud-Est) du damier, on peut inscrire, en remontant, sur chaque case du damier le nombre d'appels au calcul du gain sur cette case.

...	...	...	1
...	6	3	1
4	3	2	1
1	1	1	1

On retrouve alors le triangle de Pascal à une rotation près. Rien d'étonnant du fait de l'identité binomiale :

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n}{p-1}. \tag{1}$$

Le nombre d'appels à ROBOT-CUPIDE(i, j) fait lors du calcul du gain maximum sur la case (x, y) est

$$\binom{x+y-i-j}{x-i}.$$

Il y a donc un nombre important de répétitions des mêmes appels récursifs.

Une version itérative stockant les gains max dans un nouveau tableau (`gain[n][m]` variable globale) permet d'éviter ces répétitions inutiles.

```
int robot_cupide(int x, int y){
    int i, j;

    gain[0][0] = Damier[0][0];

    /* Bord Nord */
    for (i = 1; i <= x; i++) {
         gain[i][0] = Damier[i][0] + gain[i - 1][0];
    }

    /* Bord Ouest */
    for (j = 1; j <= y; j++) {
         gain[0][j] = Damier[0][j] + gain[0][j - 1];
    }

    /* Autres cases */
    for (j = 1; j <= y; j++) {
        for (i = 1; i <= x; i++) {
            gain[i][j] = Damier[i][j]
                         + max(gain[i - 1][j], gain[i][j - 1];
        }
    }
    // affiche(x, y); <--- pour afficher...
    return gain[x][y];
}
```

Ce n'était demandé mais on peut chercher à afficher la suite des déplacements effectués par le robot. On peut remarquer que le tableau des gains maximaux, c'est à dire le tableau `gain` après exécution de la fonction précédente (robot itératif), permet de retrouver la case d'ou l'on venait quelle que soit la case où l'on se trouve (parmi les provenances possibles, c'est celle ayant la valeur maximum). Il est donc facile de reconstruire le trajet dans l'ordre inverse. Pour l'avoir dans le bon ordre, on peut utiliser une fonction récursive d'affichage qui montre chaque coup *à la remontée* de la récursion c'est à dire *après* s'être appelée.

Le tableau `gain[n][m]` contient les gains maximaux.

```
void affiche(int i, int j){
  if (i == 0 && j == 0) {
     printf("depart");
     return;
  }
  if (i == 0) {
     affiche(i, j - 1);          // <--| noter l'ordre de ces deux
```

```
    printf(", aller à droite"); // <--| instructions. (idem suite)
    return;
  }
  if (j == 0) {
    affiche(i - 1, j);
    printf(", descendre");
    return;
  }
  if (gain[i - 1][j] > gain[i][j - 1]) {
    affiche(i - 1, j);
    printf(", descendre");
  }
  else {
    affiche(i, j - 1);
    printf(", aller à droite");
  }
}
```

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

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

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

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

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

Rappels. *Si n est un entier positif alors $n \mod 2$ (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 (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 4.

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 :

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

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

   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 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 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 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 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.

Si u_k compte le temps mis pour aller en 2^k et -2^k (une unité par immeuble) alors :

$$\begin{aligned} u_k &= 2^{k+1} + 2^k + 2^{k-1} + u_{k-1} \\ u_0 &= 3 \end{aligned}$$

En dépliant, on a :

$$\begin{aligned}
u_k = 2^{k+1}&+2^k+2^{k-1}\\
&+2^k+2^{k-1}+2^{k-2}\\
&+2^{k-1}+2^{k-2}+2^{k-3}\\
&\vdots\\
2^4\quad&+2^3\quad+2^2\\
&2^3\quad+2^2+2^1\\
&\qquad 2^2+2^1+2^0+u_0
\end{aligned}$$

D'où l'on déduit que :

$$\begin{aligned}
u_k &= 2^{k+1} + 2 \times 2^k + 3 \times (\sum_{i=0}^{k-1} 2^i) - 2^1 - 2 \times 2^0 + u_0\\
u_k &= 7 \times 2^k - 4 = 7n - 4
\end{aligned}$$

Ceci pour $n = 2^k$. Pour $2^k \leq n \leq 2^{k+1}$, le temps mis sera majoré par $7 \times 2^{k+1} - 4$ expression elle-même majorée par $7 \times (2 \times n) - 4$ puisque $2^k \leq n$. Ce qui donne un temps mis pour aller en $(n, -n)$ majoré par $14n - 4$.

Donc la distance parcourue est strictement majorée par $14 \times n$ ce qui 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, le temps est en $\Theta(n)$.

Exercice 6 (Drapeau, Dijkstra)**.**
Les éléments d'un tableau (indicé à partir de 0*) sont de deux couleurs, rouges ou verts. Pour tester la couleur d'un élément, on se donne une fonction* Couleur(T, j) *qui rend la couleur du* $j + 1$ *ième élément (d'indice* j*) du tableau* T*. On se donne également une fonction* Échange(T, j, k) *qui échange l'élément d'indice* i *et l'élément d'indice* j *et une fonction* Taille(T) *qui donne le nombre d'éléments du tableau.*

En C, on utilisera les fonctions :

- `int couleur(tableau_t T, unsigned int j)` *rendant* 0 *pour rouge et* 1 *pour vert ;*
- `echange(tableau_t T, unsigned int j, unsigned int k)` *;*
- `unsigned int taille(tableau_t T)`

où le type des tableaux `tableau_t` *n'est pas explicité.*

1. *Écrire un algorithme (pseudo-code ou C) qui range les éléments d'un tableau en mettant les 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 6.

1. Une solution :

```
void drapeau2(tableau_t t){
  int j, k;
  j = 0;            /* les éléments 0, ..., j - 1 de t sont rouges */
  k = taille(t) - 1;/* les éléments k + 1, ..., n - 1 sont verts   */
  while ( j < k ){
    if ( couleur(t, j) == 0){
      /* Si t[j] est rouge, il est à la bonne place */
      j++;
    }
    else {
      /* Si t[j] est vert on le met avec les verts */
      echange(t, j, k);
      /* Cet échange fait du nouveau t[k] un vert.  On regardera la
         couleur de l'ancien t[k] à l'étape suivante */
      k--;
    }
  } /* fin du while :
       On sort avec j = k sans avoir regardé la couleur de l'élément à
       cette place. Aucune importance. */
}
```

2. Une solution :

```
drapeau3(tableau_t t){
  int i, j, k;
  i = 0;             /* Les éléments 0, ..., i - 1 sont rouges    */
  j = 0;             /* Les éléments i, ..., j - 1 sont verts     */
  k = taille(t) - 1; /* Les éléments k + 1, ..., n - 1 sont bleus */
  while ( j <= k ){
    switch ( couleur(t, j) ){
    case 0: /* ----------- rouge ----------------------------------*/
                      /*     t[j] doit être mis avec les rouges. */
      if ( i < j ){             /*           Si il y a des verts  */
        echange(t, i, j);       /*           on doit le deplacer, */
      }                         /* sinon il est à la bonne place. */
      i++;                            /* Il y a un rouge de plus. */
      j++;                    /* Le nombre de verts reste le même. */
      break;
    case 1: /* ----------- vert ---------------------------------- */
      j++;                          /* t[j] est à la bonne place. */
      break;
    case 2: /* ----------- bleu ---------------------------------- */
      echange(t, j, k);           /* t[j] est mis avec les bleus. */
      k--;                        /* Le nombre de bleus augmente. */
    }
  }
}
```

Exercice 7 (Tri sélection)**.**
Soit un tableau indicé à partir de 0 *contenant des éléments deux à deux comparables. Par exemple*

des objets que l'on compare par leurs masses. On dispose pour cela d'une fonction

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

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

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

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

Correction 7.

1. On écrit une fonction itérative, qui parcourt le tableau de gauche à droite et maintient l'indice du minimum parmi les éléments parcourus.

```
int iminimum(tableau_t t){ /* t ne doit pas être vide */
  int j, imin;
  imin = 0;
  for (j = 1; j < taille(t); j++){
    /* Si T[imin] > T[j] alors le nouveau minimum est T[j] */
    if ( 0 > comparer(t, imin, j) ) imin = j;
  }
  return imin;
}
```

2. On fait exactement taille(tab) $- 1 = N - 1$ appels.
3. On cherche le minimum du tableau et si il n'est pas déjà à la première case, on échange sa place avec le premier élément. On recommence avec le sous-tableau commençant au deuxième élément et de longueur la taille du tableau de départ moins un. ça s'écrit en itératif comme ceci :

```
1  void triselection(tableau_t tab){
2    int n, j;
3    for (n = 0; n < taille(tab) - 1; n++) {
4      j = iminimum(sous_tableau(tab, n, taille(tab) - n));
5      if (j > 0) echanger(tab, n + j, n);
6    }
7  }
```

et en récursif comme ceci :

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

4. Traitons le cas itératif.

   On pose l'invariant : le tableau a toujours le même ensemble d'éléments mais ceux 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.

   Pour le cas récursif on raisonne par récurrence (facile). On travaille dans l'autre sens que pour l'invariant : on suppose que le tri fonctionne sur les tableaux de taille $n-1$ et on montre qu'il marche sur les tableaux de taille n.

5. 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.

6. 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 8 (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 un 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 8.

1. On écrit en C. On ne s'occupe pas de l'allocation mémoire, on suppose que le tableau dans lequel écrire le résultat a été alloué et qu'il est passé en paramètre.

```
/* ------------------------------------------------------------------------ */
/* Interclassement de deux tableaux avec écriture dans un troisième tableau */
/* ------------------------------------------------------------------------ */
void interclassement(tableau_t t1, tableau_t t2, tableau_t t){
  int i, j, k;
  i = 0;
  j = 0;
  k = 0;
  for (k = 0; k < taille(t1) + taille(t2); k++){
    if ( j == taille(t2) ){/* on a fini de parcourir t2 */
      for (; k < taille(t1) + taille(t2); k++){
        t[k] = t1[i];
i++;
      }
      break; /* <-------- sortie de boucle */
    }
    if ( i == taille(t1) ){/* on a fini de parcourir t1 */
      for (; k < taille(t1) + taille(t2); k++){
        t[k] = t2[j];
j++;
      }
      break; /* <-------- sortie de boucle */
    }
```

```
    if ( t1[i] <= t2[j] ){/* choix de l'élément suivant de t : */
      t[k] = t1[i];                    /* - dans t1;           */
      i++;
    }
    else {
      t[k] = t2[i];                    /* - dans t2.           */
      j++;
    }
  }
}
```

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

   Pour trouver un minorant du nombre de comparaisons, 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 1. *Quel que soit l'algorithme, si deux éléments consécutifs* `t[i]`, `t[i+1]` *de* t *ont des provenances différentes alors ils ont nécessairement été comparés.*

   Preuve. Supposons que ce soit faux pour un certain algorithme A. Alors, sans perte de généralités (quitte à échanger `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 9 (Notation asymptotique (devoir 2006))**.**

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

3. *Démontrer :*

$$\textit{si } f(n) = O(g(n)) \textit{ et } g(n) = O(h(n)) \textit{ alors } f(n) = O(h(n)) \tag{2}$$
$$\textit{si } f(n) = O(g(n)) \textit{ alors } g(n) = \Omega(f(n)) \tag{3}$$
$$\textit{si } f(n) = \Omega(g(n)) \textit{ alors } g(n) = O(f(n)) \tag{4}$$
$$\textit{si } f(n) = \Theta(g(n)) \textit{ alors } g(n) = \Theta(f(n)) \tag{5}$$

Correction 9. Pas de correction.

Exercice 10 (Notation asymptotique (partiel mi-semestre 2006))**.**

3 pt

1. *Est-ce que* $(n+3)\log n - 2n = \Omega(n)$ *?* (1,5 pt)
2. *Est-ce que* $2^{2n} = O(2^n)$ *?* (1,5 pt)

Correction 10.

1. On montre qu'il existe une constante positive c et un rang n_0 à partir duquel $(n+3)\log n - 2n \geq cn$.

$$(n+3)\log n - 2n \geq n(\log n - 2) \quad (\forall n > 0)$$
$$\text{pour } n \geq 8, \quad \log n - 2 \geq \log 8 - 2 = 1$$
$$\text{Donc } \forall n \geq n_0 = 8, \quad (n+3)\log n - 2n \geq 1 \times n.$$

2. On montre que c'est faux, par l'absurde. Supposons que ce soit vrai, alors :

$$\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 2^{2n} \leq c \times 2^n.$$

Donc par croissance du log, pour $n \geq n_0$ et $n > 0$:

$$2 \times n \leq \log c + n$$
$$\text{ce qui donne} \quad n \leq \log c$$

L'ensemble des entiers naturels n'étant pas borné et c étant une constante, ceci est une contradiction.

Exercice 11.
Montrer que :

$$\log(n!) = \Omega(n \log n) \tag{6}$$

Correction 11.

Solution 1 Soit $n > 0$. Si n est pair alors $n = 2k$ avec $k > 0$. Dans ce cas :

$$(2k)! \geq (2k) \times \ldots \times k \geq \underbrace{k \times \ldots \times k}_{k+1 \text{ termes}} \geq k^k$$

Donc $\log((2k)!) \geq k \log k$, c'est à dire $\log(n!) \geq \frac{1}{2} n \log(n/2)$.

Si n est impair, alors $n = 2k+1$ avec $k \geq 0$. Dans ce cas :

$$(2k+1)! \geq (2k+1) \times \ldots \times (k+1) \geq \underbrace{(k+1) \times \ldots \times (k+1)}_{k+1 \text{ termes}} = (k+1)^{k+1}$$

Donc $\log((2k+1)!) \geq (k+1)\log(k+1)$ et donc $\log(n!) \geq \frac{1}{2}n\log(n/2)$.

Ainsi pour $n > 0$ on a

$$\log(n!) \geq \frac{1}{2}n\log(n/2).$$

Pour $n \geq 4$, $n/2 \geq \sqrt{n}$, et ainsi $\log(n/2) \geq \log(\sqrt{n}) = \frac{1}{2}\log n$.

Finalement, pour $n \geq 4$,

$$\log(n!) \geq \frac{1}{4}n\log n.$$

Ce qui montre bien que $\log(n!) = \Omega(n\log n)$.

Solution 2 On a (faire un dessin)

$$\log(n!) = \sum_{k=2}^{n} \log k \geq \int_1^n \log t dt.$$

Donc $\log(n!) \geq [t\log t]_1^n = n\log n - n + 1$. Pour $n \geq 4$, $\log n - 1 \geq \frac{1}{2}\log n$. Ainsi pour $n \geq 4$, $\log(n!) \geq \frac{1}{2}n\log$. Ce qui conclue.

Solution 3 On suppose que $n > 0$. On écrit

$$\log(n!) = \sum_{k=1}^{n} \log k$$

Et comme pour la sommation $\sum_{k=1}^{n} k$ on somme deux fois :

$$\begin{aligned} 2\log(n!) &= \sum_{k=1}^{n} \log k + \sum_{k=1}^{n} \log(n+1-k) \\ &= \sum_{k=1}^{n} \log(k(n+1-k)) \end{aligned}$$

Mais lorsque $1 \leq k \leq n$, $k(n+1-k)$ est maximal pour $k = \frac{n+1}{2}$ et minimal pour $k = 1$ et $k = n$ (on raisonne sur $(a+b)(a-b)$ avec $a = \frac{n+1}{2}$ et $b = k - a$).

Ainsi pour tout $1 \leq k \leq n$, $\log(k(n+1-k)) \geq \log(n)$.

On en déduit qu'à partir du rang $n = 1$:

$$\log(n!) \geq \frac{n\log n}{2}.$$

Ce qui montre bien que $\log(n!) = \Omega(n\log n)$.

Exercice 12 (Complexité en moyenne du tri bulle (devoir 2006))**.**
Le but de cet exercice est de déterminer le nombre moyen d'échanges effectués au cours d'un tri bulle.

On considère l'implémentation suivante du tri bulle :

```
0   void tribulle(tableau_t *t){
1     int n, k, fin;
2     for (n = taille(t) - 1; n >= 1; n--) {
3       /* Les éléments d'indice > n sont à la bonne place. */
4       fin = 1;
5       for (k = 0; k < n; k++){
```

```
6         if ( 0 > comparer(t[k], t[k + 1]) ){ /* t[k] > t[k + 1] */
7           echangertab(t, k, k + 1);
8           fin = 0;
9         }
10      }
11      if (fin) break; /* Les éléments entre 0 et n - 1 sont bien ordonnés */
12    }
13  }
```

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

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

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

1. *Combien y a t-il d'inversions dans la permutation* 0, 1 ..., $n-1$ *? Et dans la permutation* $n-1$, $n-2$, ..., 0 *?*
2. *Montrer que chaque échange dans le tri bulle élimine exactement une inversion.*
3. *En déduire une relation entre le nombre total d'inversions dans toutes les permutations de* 0, ..., $n-1$ *et le nombre moyen d'échanges effectués par le tri bulle sur une entrée de taille* n.

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

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

Correction 12. Pas de correction voir celle du partiel 2006 (tri gnome similaire).

Exercice 13 (Complexité en moyenne du tri gnome (partiel mi-semestre 2006))**.**
Le but de cet exercice est d'écrire le tri gnome en C et de déterminer le nombre moyen d'échanges effectués au cours d'un tri gnome. 7 pt

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

1. *Écrire une fonction C de prototype* `void trignome(tableau_t t)` *effectuant le tri gnome.* (2,5 pt)

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

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

2. *Si le tri gnome effectue un échange entre deux éléments, que peut on dire de l'évolution du nombre d'inversions dans ce tableau avant l'échange et après l'échange (démontrer) ?* (2,5 pt)

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

3. *Si un tableau* t *de taille* n *contient* $f(n)$ *inversions, combien le tri gnome effectuera d'échanges sur ce tableau (démontrer) ? En déduire le nombre moyen d'échanges effectués par le tri gnome sur des tableaux de taille* n. (2 pt)

Correction 13.

```
1.1  void trignome(tableau_t *t){
 2     int i = 0; //                  On part du début du tableau t
 3     while ( i < taille(t) - 1 ){// Tant qu'on a pas atteint la fin de t
 4       if ( cmptab(t, i, i + 1) < 0 ){// Si (i, i + 1) inversion alors :
 5         echangetab(t, i, i + 1); // 1) on reordonne par échange;
 6         if (i > 0) i--;//           2) on recule, sauf si on était
 7         else i++; //                   au début, auquel cas on avance.
 8       }
 9       else i++; //                   Sinon, on avance.
10     }
11   }
```

2. La seule hypothèse est, qu'au cours de l'exécution du tri grome (sur une entrée non spécifiée), à une certaine étape, un échange à eu lieu. Soit t' le tableau juste avant cet échange, t'' le tableau juste après cet échange, et i_0 la valeur de la variable i au moment de l'échange. Un échange ne se produit que lorsque $t[i] > t[i+1]$ et il s'agit d'un échange entre $t[i]$ et $t[i+1]$. Ainsi, dans t', on avait $t'[i_0] < t'[i_0+1]$ et t'' est égal à t' dans lequel on a procédé à l'échange entre les éléments d'indices i_0 et i_0+1. Cet échange élimine exactement une inversion. En effet :
   - dans t', (i_0, i_0+1) est une version, pas dans t''
   - les autres inversions sont préservées :
     - lorsque $i < j$ et i, j tous deux différents de i_0 et i_0+1, (i,j) est une inversion dans t' ssi c'est une inversion dans t'' ;
     - lorsque $i < i_0$ alors : (i, i_0) est une inversion dans t' ssi (i, i_0+1) est une inversion dans t'' et (i, i_0+1) est une inversion dans t' ssi (i, i_0) est une inversion dans t'' ;
     - lorsque $i_0 + 1 < j$ alors : (i_0, j) est une inversion dans t' ssi (i_0+1, j) est une inversion dans t'' et (i_0+1, j) est une inversion dans t' ssi (i_0, j) est une inversion dans t'' ;
   - et ceci prend en considération toutes les inversions possibles dans t' et t''.
3. Nous venons de voir que tout échange au cours du tri gnome élimine exactement une inversion. Le nombre d'échanges effectués au cours du tri gnome de t est donc la différence entre le nombre d'inversions au départ, $f(n)$, et le nombre d'inversions à fin du tri. Mais le tableau final est trié et un tableau trié ne contient aucune inversion. Donc le nombre d'échange est simplement $f(n)$.

   Le nombre d'échanges est égal au nombre d'inversions dans le tableau inital. Donc le nombre moyen d'échanges sur des tableaux de taille n est $\frac{n(n-1)}{4}$.

Exercice 14 (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 14.

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 15 (Tri par base (partiel mi-semestre 2006))**.**
Soit la suite d'entiers décimaux 141, 232, 045, 112, 143. *On utilise un tri stable pour trier ces entiers selon leur chiffre le moins significatif (chiffre des unités), puis pour trier la liste obtenue selon le chiffre des dizaines et enfin selon le chiffre le plus significatif (chiffre des centaines).* 10 pt

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

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

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

On se donne la fonction auxiliaire :

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

2. *Que valent* `cle(123, 0)`, `cle(123, 1)`, `cle(123, 2)` *(inutile de justifier votre réponse) ? Plus généralement, que renvoie cette fonction ?* (1,5 pt)

On suppose que l'on dispose d'une fonction auxiliaire de tri `void triaux(tableau_t t, int i)` *qui réordonne les éléments de t de manière à ce que*

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

On suppose de plus que ce tri est stable.

3. *Écrire l'algorithme de tri par base du tableau t (utiliser la fonction* `triaux`*). On pourra considérer que k est un paramètre entier passé à la fonction de tri.* (2 pt)
4. *Si le temps d'exécution en pire cas de* `triaux` *est majoré asymptotiquement par une fonction $f(N)$ de paramètre la taille de t, quelle majoration asymptotique pouvez donner au temps d'exécution en pire cas de votre algorithme de tri par base ?* (1 pt)
5. *Démontrer par récurrence que ce tri par base trie bien le tableau t. Sur quelle variable faites vous la récurrence ? Où utilisez vous le fait que* `triaux` *effectue un tri stable ?* (3 pt)
6. *La fonction* `triaux` *utilise intensivement la fonction à deux paramètres* `cle`*. Si on cherche un majorant $f(N)$ au temps d'exécution de* `triaux`*, peut on considérer qu'un appel à* `cle` *prend un temps borné par une constante ?* (1 pt)
7. *Décrire en quelques phrases une méthode pour réaliser la fonction* `triaux` *de manière à ce qu'elle s'exécute en un temps linéaire en fonction de la taille du tableau (on pourra utiliser une structure de donnée).* (1,5 pt)

Correction 15.

1. Liste 1 : 141, 232, 112, 143, 045. Liste 2 : 112, 232, 141, 143, 045. Liste 3 : 045, 112, 141, 143, 232. Ce tri s'appelle un tri par base.
2. On a `cle(123, 0) = 3`; `cle(123, 1) = 2`; `cle(123, 2) = 1`. Cette fonction prend un nombre n et un indice i en entrée et renvoie le $i + 1$ ième chiffre le moins significatif de l'expression de n en base 10. Autrement dit, si n s'écrit $\overline{b_{k-1} \ldots b_0}$ en base 10, alors `cle(n, i)` renvoie b_i si i est inférieur à $k - 1$ et 0 sinon.

```
3.1  void tribase(tableau_t t, int k){
  2    int i;
  3    for (i = 0; i < k; i++){// k passages
  4      triaux(t, i);
  5    }
  6  }
  7
```

4. Sur un tableau de taille N, le tri base effectue k appels à `triaux` et chacun de ces appels se fait aussi sur un tableau de taille N. Les autres opérations (contrôle de boucle) sont négligeables. Chacun de ces appels est majoré en temps par $f(N)$. Dans le pire cas, le temps d'exécution du tri gnome est donc majoré par $kf(N)$.

5. Étant donné un entier n d'expression $\overline{b_{k-1}\dots b_0}$ en base 10, pour $0 \leq i \leq k-1$, on lui associe l'entier $c_i(n)$ d'expression $\overline{b_i \dots b_0}$ (les $i+1$ premiers digits les moins significatifs de n).

   On démontre par récurrence sur i qu'en $i+1$ étapes de boucle le tri gnome range les éléments du tableau t par ordre croissant des valeurs de c_i. Cas de base. Pour $i = 0$ (première étape de boucle) le tri gnome a fait un appel à `triaux` et celui-ci a rangé les éléments de t par ordre croissant du digit le moins significatif. Comme c_0 donne ce digit le moins significatif, l'hypothèse est vérifiée. On suppose l'hypothèse vérifié après l'étape de boucle i ($i+1$ ème étape). En une étape supplémentaire par l'appel à `triaux`, les éléments de t sont trié selon leur digit d'indice $i+1$ ($i+2$ ème digit). Soient $t[j]$ et $t[k]$ deux éléments quelconques du tableau après cette étape, avec $j < k$. Comme le tri auxilliaire, `triaux` est stable si $t[j]$ et $t[k]$ ont même digits d'indice $i+1$ alors ils sont rangés dans le même ordre qu'à l'étape précédente. Mais par hypothèse à l'étape précédente ces deux éléments étaient rangés selon des c_i croissants, il sont donc rangés par c_{i+1} croissants. Si les digits d'indices $i+1$ de $t[j]$ et $t[k]$ différent alors celui de $t[k]$ est le plus grand et ainsi $c_{i+1}(t[j]) < c_{i+1}(t[k])$. Dans les deux cas $t[j]$ et $t[k]$ sont rangés par c_{i+1} croissants. Ainsi l'hypothèse est vérifiée après chaque étape de boucle.

   Lorsque $i = k-1$, pour tout indice j du tableau $c_i(t[j]) = t[j]$ ainsi le tableau est bien trié à la fin du tri gnome.

   La récurrence est faite sur i qui va de 0 à $k-1$ (on peut aussi considérer qu'elle porte sur k). La stabilité de `triaux` a servi à démonter le passage de l'étape i à l'étape $i+1$ de la récurrence.

6. L'exécution de la fonction `cle` demande $i+1$ étapes de boucle. Comme, lors d'un appel à `triaux`, i est borné par k, le temps d'exécution de `cle` est linéaire en k (). Mais k est considéré comme une constante. Le temps d'exécution de `cle` peut don être considéré comme borné par une constante ().

7. Pour réaliser `triaux` en temps linéaire, il suffit de faire un tri par dénombrement, avec données satellites. Il est alors pratique d'utiliser une structure de pile : on crée dix piles vides numérotées de 0 à 9, on parcours t du début à la fin en empilant chaque élément sur la pile de numéro la valeur de la clé (son $i+1$ ème digit). Ceci prend N étapes. Lorsque c'est fini on dépile en commençant par la pile numéroté 9 et en stockant les éléments dépilés dans t en commençant par la fin. Ceci prend encore N étapes.

Exercice 16 (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 aababb et ababba sont deux sous-suites équilibrées de s de cette longueur.

Pour faciliter l'écriture de l'algorithme, on considérera que :
- *la suite en entrée est donnée dans un tableau de taille n, avec un élément par case ;*
- *chaque cellule de ce tableau est soit l'entier 1 soit l'entier -1 (et non pas a et b).*

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

6. *Pour que* $(s_k)_{i \leq k \leq j}$ *soit équilibrée que faut-il que* `aux[i]` *et* `aux[j]` *vérifient ?*

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

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

Correction 16. Pas de correction.

Exercice 17 (Listes Chaînées)**.**
Soit la structure ***liste*** *définie en C par :*

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

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

Correction 17.

1.
```
liste_t entrelacement(liste_t liste1,
    liste_t liste2){
 if (liste1 == NULL) return liste2;
 if (liste2 == NULL) return liste1;
 if ( cmpListe(liste1, liste2)>=0 ){
    liste1->suivant = entrelacement(
             liste1->suivant, liste2);
    return liste1;
  }
 else {
    liste2->suivant = entrelacement(
             liste1, liste2->suivant);
    return liste2;
  }
}


liste_t entrelacIter(liste_t liste1,
  liste_t liste2) {
 liste_t out, finale;

 if(liste1==NULL)
   return liste2;
 if(liste2==NULL)
   return liste1;

if(cmpListe(liste1, liste2)>=0){
  out=liste1;
  liste1=liste1->suivant;
}
else{
  out=liste2;
  liste2=liste2->suivant;
}

finale=out;
while((liste1!=NULL)&&
      (liste2!=NULL) ){
  if(cmpListe(liste1, liste2)>=0){
    out->suivant=liste1;
    liste1=liste1->suivant;
  }
  else{
    out->suivant=liste2;
    liste2=liste2->suivant;
  }
 out=out->suivant;
}
```

```
 if(liste1==NULL)
    out->suivant=liste2;
 else
    out->suivant=liste1;

 return finale;
}
```

2.
```
void elimineRepetition(liste_t liste){
  liste_t courant, aux, suiv;
  courant=liste;
  while(courant!= NULL) {
    aux=courant;
    do {
        suiv=aux->suivant;
        if((suiv!=NULL) &&
     (suiv->element==courant->element)){
            aux->suivant=suiv->suivant;
            free(suiv);
          }
        else
           aux=suiv;
      } while(aux!=NULL);

    courant=courant->suivant;
  }
}
```

Exercice 18 (Primitives de pile). *1. Définir une structure pile à l'aide d'un tableau d'éléments (de type element_t) de hauteur maximum N. On considérera que N est une constante donnée.*

2. Écrire les fonctions suivantes :
- *pile_t creerPile() ; qui crée une pile vide,*
- *int pileVide(pile_t pile) ; qui retourne* 1 *si la pile est vide et* 0 *sinon,*
- *element_t depiler(pile_t pile) ; qui retourne le dernier élément après l'avoir retiré de la pile,*
- *void empiler(pile_t pile, element_t elt) ; qui empile l'élément elt,*
- *element_t dernierElement(pile_t pile) ; qui retourne le dernier élément empilé,*
- *void viderPile(pile_t pile) ; qui vide la pile,*
- *unsigned int hauteurPile(pile_t pile) ; qui retourne la hauteur de la pile,*

Correction 18.

1.
```
typedef struct {
  element_t tab[N];
  unsigned int hauteur;
} pile_s;
typedef pile_s * pile_t;
```

2.
```
pile_t creerPile(){
 pile_t pile = (pile_t)malloc(sizeof(pile_s));
 pile->hauteur=0;
 return pile;
}

int pileVide(pile_t pile){
 if(pile->hauteur==0)
   return 1;
 else
   return 0;
}

element_t depiler(pile_t pile){
 if(hauteurPile(pile)==0){
  printf("Erreur de pile\n"); exit(-1);
 }
 pile->hauteur--;
 return pile->tab[pile->hauteur];
}

void empiler(pile_s pile, element_t elt){
 if(hauteurPile(pile)>=N){
  printf("Memoire de pile insuffisante\n");
  exit(-1);
 }
 pile->tab[pile->hauteur++]=elt;
}

element_t dernierElement(pile_t pile){
 if(hauteurPile(pile)==0){
  printf("Erreur de pile\n"); exit(-1);
 }
 return pile->tab[pile->hauteur-1];
}

void viderPile(pile_t pile){
 pile->hauteur=0;
}

unsigned int hauteurPile(pile_t pile){
  return pile->hauteur;
}
```

Exercice 19 (Déplacement de pile)**.**
On se donne trois piles P_1, P_2 et P_3. La pile P_1 contient une suite de nombres entiers positifs. Pour la suite de l'exercice, on utilisera la structure et les opérations définies dans l'exercice précédent avec des éléments de type entier.

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

Correction 19.

1.
```
void pilePaireImpaire(pile_t P1,
      pile_t P2, pile_t P3){
 int aux;

 viderPile(P2);
 viderPile(P3);
 while(pileVide(P1)==0){
  aux=depiler(P1);
  if( (aux&1)==1)
     empiler(P2, aux);
  else
     empiler(P3, aux);
 }

 while(pileVide(P3)==0){
  aux=depiler(P3);
  empiler(P2, aux);
 }
```

2.
```
void pilePaire(pile_t P1,
     pile_t P2, pile_t P3) {
 int aux;

 viderPile(P2);
 viderPile(P3);

 while(pileVide(P1)==0){
  aux=depiler(P1);
  empiler(P3, aux);
 }
 while(pileVide(P3)==0){
  aux=depiler(P3);
  empiler(P1, aux);
  if( (aux&1)==0)
     empiler(P2, aux);
 }
}
```

Exercice 20 (Test de bon parenthésage)**.**
Soit une expression mathématique dont les éléments appartiennent à l'alphabet suivant :

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

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

Correction 20.

```
int verifieParenthese(char *expr){
 int i, aux;
 pile_t p=creerPile();

 for(i=0; expr[i]!=0; i++){
  switch(expr[i]){
   case '(' :
      empiler(p,1); break;
   case '[' :
      empiler(p,2); break;
   case ')' :
      if(pileVide(p)==1)
         return -1;
       else{
         aux=depiler(p);
         if(aux!=1)
            return -1;
      }
      break;
   case ']' :
      if(pileVide(p)==1)
         return -1;
      else{
         aux=depiler(p);
         if(aux!=2)
           return -1;
      }
      break;
   default:
```

```
 }
}

if(pileVide(p)==0)
    return -1:

 return 0;
}
```

Exercice 21 (Calculatrice postfixe)**.**
On se propose de réaliser une calculatrice évaluant les expressions en notation postfixe. L'alphabet utilisé est le suivant : $\mathcal{A} = \{0, \ldots, 9, +, -, *, /\}$ *(l'opérateur* $-$ *est ici binaire). Pour un opérateur n-aire* P *et les opérandes* $O_1, \ldots, O_n$*, l'expression, en notation postfixe, associée à* P *sera :* $O_1, \ldots, O_n P$*. Ainsi, la notation postfixe de l'expression* $(2*5)+6+(4*2)$ *sera :* $2\ 5\ *\ 6\ +\ 4\ 2\ *\ +$*. On suppose que l'expression est valide et que les nombres utilisés dans l'expression sont des entiers compris entre* 0 *et* 9*. De plus, l'expression est donnée sous forme de chaînes de caractères terminée par un zéro. Par exemple* $(2*5)+6+(4*2)$ *sera donnée par la chaîne "*$25*6+42*+$*".*
Écrire un algorithme qui évalue une expression postfixe à l'aide d'une pile d'entiers. (On pourra utiliser la fonction int ctoi(char c){return (int)(c-'0');} pour convertir un caractère en entier).

Correction 21.

```
int calculePostfixe(char *expr){
  int i, op1, op2;
  pile_t p=creerPile();

 for(i=0; expr[i]!=0; i++){
  switch(expr[i]){
   case '+' :
      op2=depiler(p);
      op1=depiler(p);
      empiler(p, op1+op2);
      break;
   case '-' :
      op2=depiler(p);
      op1=depiler(p);
      empiler(p, op1-op2);
      break;
   case '*' :
      op2=depiler(p);
      op1=depiler(p);
      empiler(p, op1*op2);
      break;
   case '/' :
      op2=depiler(p);
      op1=depiler(p);
      empiler(p, op1/op2);
      break;
   default :
      op1=ctoi(expr[i]);
      empiler(p, op1);
  }
 }
 return depiler(p);
}
```

Exercice 22 (Primitives de file)**.**
On souhaite implémenter une file à l'aide d'un tableau circulaire.

1. *Définir une structure file à l'aide d'un tableau circulaire d'éléments (de type element_t) de taille* N*. Comment doit-on choisir* N *pour que la navigation dans le tableau circulaire se ramène à des opérations très élémentaires en informatique ?*
2. *Écrire les fonctions suivantes :*
   - *file_t creerFile() ; qui crée une file vide,*
   - *int fileVide(file_t file) ; qui retourne* 1 *si la file est vide et* 0 *sinon,*
   - *element_t defiler(file_t file) ; qui retourne le premier élément après l'avoir retiré de la file,*
   - *void enfiler(file_t file, element_t elt) ; qui enfile l'élément elt,*
   - *element_t premierElement(file_t file) ; qui retourne le premier élément enfilé,*
   - *void viderFile(file_t file) ; qui vide la file,*
   - *unsigned int tailleFile(file_t file) ; qui retourne la taille de la file.*

Correction 22. On choisira N sous forme de puissance de deux. Ainsi, une opération de type x *modulo* N se résumera à une opération "et" logique : $x \ \&\ (N-1)$.
Avec un tableau circulaire, la file pleine et la file vide remplissent la même condition : indexe de début = indexe de fin. Il faudra donc soit maintenir une case vide soit avoir un champ taille dans la structure file. Dans la suite, on maintiendra une case vide pour différencier la file pleine de la file vide.

1.
```
typedef struct {
  element_t tab[N];
  unsigned int debut;
  unsigned int fin;
} file_s;
typedef file_s * file_t;
```

2.
```
file_t creerFile(){
 file_t file = (file_t)malloc(sizeof(file_s));
 file->debut=0;
 file->fin=0;
 return file;
}

int fileVide(file_t file){
 if(file->debut==file->fin)
   return 1;
 else
   return 0;
}

element_t defiler(file_t file){
 element_t val;
 if(fileVide(file)==1){
  printf("Erreur de file\n"); exit(-1);
 }

 val=file->tab[file->debut];
 file->debut=(file->debut+1)&(N-1);
 return val;
}

void enfiler(file_t file, element_t elt){
 if(tailleFile(file)>=(N-1)){
  printf("Memoire de file insuffisante\n");
  exit(-1);
 }
 file->tab[file->fin]=elt;
 file->fin=(file->fin+1) & (N-1);
}

element_t premierElement(file_t file){
 if(tailleFile(file)==0){
  printf("Erreur de file\n"); exit(-1);
 }
 return file->tab[file->debut];
}

void viderFile(file_t file){
 file->debut=0;
 file->fin=0;
}

unsigned int tailleFile(file_t file){
  unsigned int taille;
  taille=((file->fin+N)-file->debut)&(N-1);
  return taille;
}
```

Exercice 23 (Tri avec files)**.**
On se donne une file d'entiers que l'on voudrait trier avec le plus grand élément en fin de file. On n'est autorisé à utiliser que fileVide et les opérations suivantes :

- *defilerEnfiler : Défile le premier élément de la première file et l'ajoute à la deuxième file.*
- *comparer : Retourne 1 si le premier élément de la première file est plus grand ou égal au premier élément de la deuxième file et 0 sinon.*

1. *À partir des primitives de la structure file, proposer un algorithme pour chacune des deux dernières opérations (fileVide ayant été définie dans l'exercice précédent).*
2. *Donner un algorithme de tri qui utilise seulement ces trois opérations et 3 files. La pile f_1 contiendra les entiers à trier, f_2 contiendra les entiers triés après exécution et la file f_3 pourra servir de file auxiliaire. On pourra, aussi, utiliser la fonction void permuteFile(file_t f1, file_t f2) qui permute le contenu des deux files.*
3. *Le tri est-il stable ?*

Correction 23.

1.
```
void defilerEnfiler(file_t f1, file_t f2){
  if(fileVide(f1)==0){
    enfiler(f2, defiler(f1));
  }
}
```

```
int comparer(file_t f1, file_t f2){
  if( (fileVide(f1)==0) ||
      (fileVide(f2)==0) ){
    printf("Erreur de file\n");
```

```
      exit(-1);
    }
   if(premierElement(f1) >=
      premierElement(f2))
        return 1;
   return 0;
}
```

2.
```
void trieFile(file_t f1,
     file_t f2, file_t f3){

  while(fileVide(f1)==0){
    while( (fileVide(f2)==0)&&
        (comparer(f1,f2)==1))
           defilerEnfiler(f2,f3);
     defilerEnfiler(f1,f3);
     while(fileVide(f2)==0)
       defilerEnfiler(f2,f3);
     permuteFile(f2,f3);
    }
}
```

3. Soit a_1 et a_2 deux éléments (de clés égaux) appartenant à la file f_1 avec a_1 situé avant a_2. À une itération p, a_1 est trié dans f_2 et a_2 se trouve en tête de file dans f_1. La deuxième boucle while va déplacer tous les éléments de la tête de file jusqu'à l'élément a_1 (non inclus) de f_2 vers f_3. L'élément a_2 étant égale à a_1, $comparer(f_1, f_2)$ retournera 1, par conséquent a_1 sera aussi déplacé vers f_3. À la fin de la deuxième boucle a_2 sera placé dans f_3. Ainsi, a_2 sera placé après a_1 dans la file triée. On peut en conclure que le tri est stable.