Table des matières

1. Caractères consécutifs

Toutes les procédures et fonctions seront totalement paramétrées (i.e aucun effet de bord). Écrire chaque procédure ou fonction avec un programme qui l'appelle et affiche les résultats.

1.1 Occurences d'un caractère

La fonction `nb_single` reçoit en paramètres le caractère `C1` à compter et un caractère de terminaison différent `CFin`.

La fonction lit au clavier une suite de caractères se terminant par `CFin`. Elle renvoie le nombre d'occurences de `C1` dans la suite.

Le programme fait compter les `'L'` dans une suite terminée par `'.'`

Correction

```
PROGRAM Compte1;
VAR n : integer;

FUNCTION nb_single (C1, CFin : char) : integer;
VAR res : integer;
    c   : char;
BEGIN
    res := 0; { init }
    read(c);  { lit le 1er caractere }
    while c <> CFin do
    begin
        if c = C1 then res := res+1;
        read(c); { lit le prochain caractere }
    end;
    readln; { vide le buffer apres le CFin         }
            { et SURTOUT absorbe le retour chariot }
    nb_single := res;  { resultat de la fonction }
END;

BEGIN { Programme principal }
    writeln ('Tapez une suite de caracteres terminee par ''.''');
    n := nb_single ('L', '.');
    writeln ('Nb occurences de ''L'' = ', n);
END.
```

On peut aussi écrire un `repeat until` avec un seul `read` au début (équivalent), ou un `while` avec un seul `read` au début (à proscrire).

1.2 Occurences d'un couple de caractères

La fonction `nb_couple` renvoie le nombre d'occurences de caractères `'C1'` et `'C2'` *consécutifs* dans une suite de caractères lue au clavier se terminant par `CFin` (`C1`, `C2` et `CFin` sont tous différents).

Le programme fait compter les `'L','E'` dans une suite terminée par `'.'`

On demande d'implémenter 2 algorithmes différents pour la fonction, qui devront "survivre" au jeu d'essai : `LEL␣E␣LLE␣L.↩`

Correction

Le jeu d'essai contient les contre-exemples types : `'L␣E'`, `'LLE'` et `'L.'`

Les 3 algorithmes que l'on rencontre souvent sont :

1. Une boucle avec plusieurs `read` : plante systématiquement sur les contre-exemples. Il faut exécuter l'algorithme "à la main" pour s'en rendre compte. Mise au point possible (un `else` bien placé ...), mais relecture difficile.
2. Mémorisation du caractère précédent : très simple et efficace.
3. Petit automate avec une variable `etat` (ou simplement emploi d'un booléen) : bien également.

⟶ Un seul `read` par boucle et on évite des ennuis.

- 1. Solution avec plusieurs `read`

```
PROGRAM Compte2;
VAR n : integer;

FUNCTION nb_couple (C1, C2, CFin : char) : integer;
VAR res  : integer;
    c    : char;
BEGIN
    res := 0;  { init }
    read(c);
    while c <> CFin do
    begin
        if c = C1 then
        begin
            read(c);
            if c = C2 then res := res+1;
        end
        else read(c);  { ne pas oublier ce else !! }
    end;
    readln;
    nb_couple := res;
END;

BEGIN { Programme principal }
    writeln ('Tapez une suite de caracteres terminee par ''.''');
    n := nb_couple ('L', 'E', '.');
    writeln ('Nb occurences de ''LE'' = ', n);
END.
```

- 2. Solution avec mémorisation du caractère précédent :

```
FUNCTION nb_couple (C1, C2, CFin : char) : integer;
VAR res  : integer;
    c, d : char;
BEGIN
    res := 0;  { init }
    d := CFin; { precedent car. lu, init <> C1 et C2 }
    read(c);
    while c <> CFin do
    begin
        if (d = C1) and (c = C2) then res := res+1;
        d := c;
        read(c);
    end;
    readln;
    nb_couple := res;
END;
```

- 2bis. Variante : `if` après le `read`

```
    { init : res, read(c), mais pas d }
    while c <> CFin do
    begin
        d := c;
        read(c);
        if (d = C1) and (c = C2) then res := res+1;
    end;
```

- 3. Solution avec état (petit automate) :

```
FUNCTION nb_couple (C1, C2, CFin : char) : integer;
VAR res, etat : integer;
    c : char;
BEGIN
    res := 0;  { init }
    etat := 0; { init : on n'a pas lu C1 }
    read(c);
    while c <> CFin do
    begin
        case etat of
          0 : if c = C1 then etat := 1;
          1 : case c of
                C2 : begin res := res+1; etat := 0; end;
                C1 : ;  { on retombe sur C1 --> on laisse etat = 1 }
                else etat := 0;
              end;
        end; { case etat }
        read(c);
    end;
    readln;
    nb_couple := res;
END;
```

- 3bis. Variante : avec booléen. Facile à déduire du 2bis.

```
{ init : res, read(c), test := false; }
while c <> CFin do
begin
    test := c = C1;
    read(c);
    if test and (c = C2) then res := res+1;
end;
```

1.3 Occurences d'un triplet de caractères

La fonction `nb_triplet` renvoie le nombre d'occurences de caractères `'C1' 'C2' 'C3'` *consécutifs* dans une suite de caractères lue au clavier se terminant par `CFin`. (`C1`, `C2`, `C3` et `CFin` sont tous différents).

Le programme fait compter les `'L','E','S'` dans une suite terminée par `'.'`

On demande d'implémenter 2 algorithmes différents pour la fonction, et d'imaginer un jeu d'essai pertinent auquel ils devront résister.

Correction

Le fait que `C1`, `C2` et `C3` soient tous différents ne change rien pour l'algorithme avec mémorisation des 2 caractères précédents ; mais cela simplifie la solution de l'automate, qui reste encore lourde à écrire.

Quel jeu d'essai prendre ?

- Solution avec mémorisation du caractère précédent :

```
PROGRAM Compte3;
VAR n : integer;

FUNCTION nb_triplet (C1, C2, C3, CFin : char) : integer;
VAR res  : integer;
    c, d, e : char;
BEGIN
    res := 0;  { init }
    d := CFin; { precedent car. lu, init <> C1 et C2 et C3 }
    e := CFin; { precedent de d }
    read(c);
    while c <> CFin do
    begin
        if (e = C1) and (d = C2) and (c = C3)
        then res := res+1;
        e := d; d := c;    { dans cet ordre !! }
        read(c);
    end;
    readln;
    nb_triplet := res;
END;

BEGIN { Programme principal }
    writeln ('Tapez une suite de caracteres terminee par ''.''');
    n := nb_triplet ('L', 'E', 'S', '.');
    writeln ('Nb occurences de ''LES'' = ', n);
END.
```

- Solution avec état (petit automate) :

```
FUNCTION nb_triplet (C1, C2, C3, CFin : char) : integer;
VAR res, etat : integer;
    c : char;
BEGIN
    res := 0;  { init }
    etat := 0; { init : on n'a pas lu C1 }
    read(c);
    while c <> CFin do
    begin
        case etat of
          0 : if c = C1 then etat := 1;
          1 : case c of
                C2 : etat := 2;
                C1 : ;  { on retombe sur C1 --> on laisse etat = 1 }
                else etat := 0;
              end;
          2 : case c of
                C3 : begin res := res+1; etat := 0; end;
                C1 : etat := 1;  { on retombe sur C1 }
                else etat := 0;
              end;
        end; { case etat }
        read(c);
    end;
    readln;
    nb_triplet := res;
END;
```

2. Nombre parfait

2.1 Test par divisions

Un entier positif est dit *parfait* s'il est égal à la somme de ses diviseurs (excepté lui-même). Par exemple 6 est parfait, car $6 = 1+2+3$; de même 28 est parfait, car $28 = 1+2+4+7+14$.

Écrire une fonction booléenne `parfait` pour un entier `n`. Optimiser.

Correction Solution classique :

```
FUNCTION parfait (n : integer) : boolean;
VAR s, i : integer;
BEGIN
    s := 0;                  { somme des diviseurs }
    for i := 1 to n-1 do
      if n mod i = 0
        then s := s + i;     { ajoute le diviseur i }
    parfait := (n > 0) and (s = n);
END;
```

On peut aussi initialiser s à 1 et démarrer la boucle à 2, mais attention la fonction doit renvoyer `false` pour $i = 1$; on doit alors rajouter un test à l'initialisation.

On peut optimiser la fonction en constatant que si i divise n, alors (n div i) est aussi un diviseur puisque $i \times (n \text{ div } i) = n$.

Il suffit de faire varier i de 2 à $\sqrt{n}$: si i est diviseur, alors le diviseur (n div i) est dans l'intervalle $\sqrt{n}$ à n ; donc on a tous les diviseurs.

Il faut penser à démarrer à 2 pour ne pas ajouter n. Il y a aussi un problème si n est un carré : on aura ajouté 2 fois le diviseur $r = \sqrt{n}$.

```
FUNCTION parfait (n : integer) : boolean;
VAR s, i, r : integer;
BEGIN
    if n <= 1
    then s := 0
    else begin
           s := 1;
           r := trunc(sqrt(n));
           for i := 2 to r do  { beaucoup plus rapide }
             if n mod i = 0
               then s := s + i + n div i;
           if r*r = n          { si n est un carre' }
           then s := s - r;    { on a compte' un diviseur r en trop }
         end;
    parfait := (n > 0) and (s = n);
END;

VAR x : integer;
BEGIN
    for x := 1 to 1000 do if parfait(x) then writeln(x);
END.
```

2.2 Crible sur les nombres parfaits

Faire une procédure `crible_parfait` qui recherche et affiche, à l'aide d'un crible, les nombres parfaits sur un intervalle de 1 à M.

On prévoit de stocker pour chaque entier la somme de ses diviseurs, somme qui sera calculée pendant le crible : le principe du crible est, pour chaque entier i de l'intervalle, de parcourir les multiples de i auxquels on somme le diviseur i.

Correction

```
PROCEDURE crible_parfait (m : integer);
CONST MaxVec = 10000;
TYPE  TVec = array [1..MaxVec] of integer;
VAR   s : TVec;
      i, j : integer;
BEGIN
    if m > MaxVec
    then writeln ('Erreur, ', m, ' > ', MaxVec)
    else begin
           { Initialisation : somme des diviseurs de i }
           for i := 1 to m do s[i] := 0;

           { crible }
           for i := 1 to m do
           begin
             j := i+i;
             while (j <= m) do
             begin
               s[j] := s[j] + i;  { i est un diviseur de j }
               j := j+i;
             end;
           end;

           { affichage }
           writeln ('Nombres parfaits entre 1 et ', m);
           for i := 1 to m do
             if s[i] = i then writeln (i);
         end;
END;
```

On peut afficher au fur et à mesure dans le crible :

```
           { crible et affichage }
           writeln ('Nombres parfaits entre 1 et ', m);
           for i := 1 to m do
           begin
             { Test si parfait }
             if s[i] = i then writeln (i);

             j := i+i;
             while (j <= m) do
             begin
               s[j] := s[j] + i;  { i est un diviseur de j }
               j := j+i;
             end;
           end;
```

Les nombres parfaits entre 1 et 10 000 000 sont 6, 28, 496, 8128.

3. Nombre de max

3.1 Nombre de max dans un vecteur

On déclare le type `vecteur` suivant :

```
CONST VMax = 1000;
TYPE  vecteur = array [1..VMax] of integer;
```

Soit `v` un `vecteur` de `vn` éléments ($1 \leq$ `vn` $\leq$ `VMax`).

Écrire une fonction `nb_de_max` qui renvoie le nombre d'occurences de la valeur maximum présente dans un vecteur `v` non trié.

Exemple

	1							vn
v :	3	5	2	5	7	7	1	5

Correction

a) Procéder en 2 temps : recherche du max puis comptage du max.

```
FUNCTION nb_de_max (v : vecteur; vn : integer) : integer;
VAR i, max, nb : integer;
BEGIN
    { init }
    max := v[1]; nb := 0;

    { recherche du max }
    for i := 2 to vn do
      if v[i] > max then max := v[i];

    { comptage du max }
    for i := 1 to vn do
      if v[i] = max then nb := nb+1;

    nb_de_max := nb;
END;
```

b) Procéder en une seule boucle.

```
    { init }
    max := v[1]; nb := 1;

    for i := 2 to vn do
      if v[i] > max
      then begin
             max := v[i];
             nb := 1;
           end
      else if v[i] = max        { ne pas oublier else !! }
           then nb := nb+1;
```

3.2 Nombre de max dans une suite

Écrire une fonction `nb_de_max` qui renvoie le nombre d'occurences de la valeur maximum présente dans une suite de nombre rentrée au clavier, terminée par -1.

La fonction ne doit pas utiliser de vecteur.

Exemple `3 5 2 5 7 7 1 5 -1`

Correction

```
FUNCTION nb_de_max : integer;
VAR x, max, nb : integer;
BEGIN
    { init }
    read (x);
    max := x; nb := 0;   { nb := 0 --> important pour le cas }
                                { où la suite est vide, mais }
    while x <> -1 do            { aussi parce qu'on passe la }
    begin                       { 1ere fois dans le else.    }
      if x > max
      then begin
             max := x;
             nb := 1;
           end
      else if x = max           { ne pas oublier else !! }
           then nb := nb+1;

      read (x);
    end;

    nb_de_max := nb;
END;
```

4. Lecture de chiffres

4.1 Schémas équivalents

Écrire le programme équivalent avec un `while`.

```
PROGRAM lecture1;
CONST CarFin = '.';
VAR c : char;
BEGIN
    repeat
      read(c);
      if (c <> CarFin)
      then writeln (c, ' ', ord(c));
    until c = CarFin;
    readln;
END.
```

Correction Remarque : le dernier `readln` sert à absorber le retour chariot. Il ne faut pas l'oublier, sinon ce retour chariot risque de perturber le `readln` de la fin du programme principal, qui sert à garder la fenêtre à l'écran.

```
{ ... }
BEGIN
    read(c);
    while (c <> CarFin) do
    begin
      writeln (c, ' ', ord(c));
      read(c);
    end;
    readln;
END.
```

4.2 Lecture de chiffres

Lire au clavier une suite de caractères digit se terminant par un `' '`. Convertir au vol chaque caractère en chiffre et l'afficher, ou interrompre la lecture et afficher un message d'erreur.

Correction Le code Ascii de `'0'` est 48, celui de `'1'` est 49, etc. L'important est de savoir que ces valeurs sont consécutives. Pour convertir un digit `c` en chiffre `x` on fait donc `x := ord(c) - ord('0');`

On affiche un message d'erreur si `c` n'est pas un digit.

```
PROGRAM lit_chiffres;
CONST CarFin = ' ';
VAR c : char;
    x : integer;
    erreur : boolean;
BEGIN
    erreur := false; { init }

    repeat
      read(c);
      case c of
        '0'..'9' : begin
                     x := ord(c) - ord('0');
                     writeln (x);
                   end;
        CarFin   : ;  { on ne fait rien }
        else       begin
                     erreur := true;
                     writeln ('Erreur ', c, ' non digit');
                   end;
      end; { case c }
    until (c = CarFin) or erreur;

    readln; { vide le buffer jusqu'au retour chariot }
END.        { qui a declenche' la lecture           }
```

4.3 Horner

Lire au clavier une suite de caractères digit se terminant par un ' '. Calculer et afficher la valeur entière qu'elle représente, ou interrompre la lecture et afficher un message d'erreur.

Correction

– On part du même algorithme.
– Attention, le dépassement de capacité n'est pas géré.

```
PROGRAM Horner_entier;
CONST CarFin = ' ';
VAR c : char;
    n : integer;
    erreur : boolean;
BEGIN
    erreur := false; { init }
    n := 0;

    repeat
      read(c);
      case c of
        '0'..'9' : n := n *10 + ord(c) - ord('0');
        CarFin   : ;
        else       begin
                     erreur := true;
                     writeln ('Erreur ', c, ' non digit');
                   end;
      end; { case c }
    until (c = CarFin) or erreur;

    readln;
    if not erreur then writeln ('Entier lu : ', n);
END.
```

4.4 Horner flottant

Lire au clavier une suite de caractères digit, éventuellement séparés par une `','` et se terminant par un `' '`. Calculer et afficher la valeur réelle qu'elle représente, ou interrompre la lecture et afficher un message d'erreur.

Correction

– On part du même algorithme.
– On détecte une erreur si il y a plus d'une virgule.
– On aurait pu faire une boucle de lecture de partie entière puis une boucle de lecture de partie décimale.
– On se sert de `p10 = 0.0` ou `<> 0.0` comme *flag* pour savoir si on est avant ou après la virgule ; on aurait pu prendre un booléen à la place.

```
PROGRAM Horner_flottant;
CONST CarFin = ' ';
VAR c : char;
    x, p10 : real;
    erreur : boolean;
BEGIN
    { init }
    erreur := false;
    p10 := 0.0;
    x := 0.0;

    repeat
      read(c);
      case c of
        ','      : if p10 = 0.0
                   then p10 := 1.0
                   else begin
                          erreur := true;
                          writeln ('Erreur virgule');
                        end;
        '0'..'9' : if p10 = 0.0
                   then x := x * 10.0 + ord(c) - ord('0')
                   else begin
                          p10 := p10 * 10.0;
                          x := x + (ord(c) - ord('0')) / p10;
                        end;
        CarFin   : ;
        else       begin
                     erreur := true;
                     writeln ('Erreur ', c, ' non digit');
                   end;
      end; { case c }
    until (c = CarFin) or erreur;

    readln;
    if not erreur then writeln ('Reel lu : ', x);
END.
```

5. Le jeu du loto

On se propose d'écrire un programme de jeu de loto. Le jeu se déroule de la façon suivante : chaque joueur mise cinq francs et choisit quatre numéros. Après le tirage de quatre numéros au hasard les mises sont redistribuées entre les joueurs de la façon suivante :

- ▷ s'il y a des joueurs qui ont trouvé les quatre bons numéros alors la moitié des mises est répartie entre eux,
- ▷ si des joueurs ont trouvé trois bons numéros, ils se partagent la moitié du montant restant après la première répartition,
- ▷ enfin, les joueurs qui ont trouvé seulement deux bons numéros se partagent la moitié du montant restant après les deux répartitions précédentes.

L'organisateur empoche la somme restant après les répartitions entre les joueurs.

Exemple : avec 200 francs de mises, 2 joueurs à 4 bons numéros, aucun joueur à 3 bons numéros et 5 joueurs à 2 bons numéros, les gains sont de 50 francs pour les joueurs à 4 numéros, 10 francs pour les joueurs à 2 numéros et 50 francs pour l'organisateur.

Les choix des `NbJoueurs` joueurs sont stockés dans le tableau `Choix` de la manière suivante : la $i^{ème}$ ligne du tableau contient les quatre numéros choisis par le $i^{ème}$ joueur (on suppose que le nombre de joueurs n'excède pas `MaxJoueurs`). On utilisera le tableau `NbBonsNumeros` pour stocker les nombres de bons numéros trouvés par les joueurs. Les numéros du tirage sont stockés dans le tableau `Tirage`. Les valeurs correspondant aux gains des joueurs qui ont trouvé quatre, trois ou deux bons numéros seront stockées dans le tableau `TabGain`, et la variable `GainOrganisateur` recevra le gain de l'organisateur.

On utilisera les déclarations suivantes :

```
PROGRAM Loto;

CONST MaxJoueurs  = 50;  { nombre maximal de joueurs }
      DepartGain  =  2;  { minimum de numéros pour gagner }
      NbNumTirage =  4;  { nombre de numeros du tirage }
      PrixUnJeu   =  5;  { en francs }
      MaxNumero   = 20;  { plus grand numero jouable }

VAR Choix : array[1..MaxJoueurs,1..NbNumTirage] of integer;
    NbBonsNumeros : array[1..MaxJoueurs] of integer;
    Tirage : array[1..NbNumTirage] of integer;
    NbJoueurs : integer;
    Mises : real;
    TabGain : array [DepartGain .. NbNumTirage] of real;
    GainOrganisateur : real;
```

Ecrivez les procédures et fonctions suivantes (les paramètres sont indiqués) :

1. procédure `Saisie` : lecture du nombre de joueurs `NbJoueurs`, lecture et stockage dans le tableau `Choix` des numéros choisis par chaque joueur,

2. procédure `CalculTirage` : calcul des numéros du tirage et stockage dans le tableau `Tirage` (les numéros du tirage sont tous distincts et compris entre 1 et `MaxNumero` ; on utilisera la fonction `random(n)` qui renvoie un nombre compris entre 0 et `n-1`),
3. fonction `BonsNumeros(n)` : calcul du nombre de numéros choisis par le joueur `n` qui figurent dans `Tirage`,
4. procédure `Resultats` : calcul et stockage dans le tableau `NbBonsNumeros` des nombres de bons numéros choisis par chacun des joueurs (on utilisera la fonction `BonsNumeros`),
5. fonction `NbGagnants(n)` : calcul du nombre de joueurs qui ont choisis `n` bons numéros.
6. procédure `CalculGains` : calcul des valeurs du tableau `TabGain` et de la variable `GainOrganisateur` à partir des règles de répartition (cette procédure utilise la fonction `NbGagnants`),
7. procédure `Affichage` : affichage des gains de chacun des joueurs et de l'organisateur.

Correction

2. Algorithme du sac :

```
PROCEDURE CalculTirage;
VAR sac : array [1..MaxNumero] of integer;
    p, i, nsac : integer;
BEGIN
    { init }
    nsac := MaxNumero;
    for i := 1 to nsac do sac[i] := i;

    { tirage }
    for i := 1 to NbNumTirage do
    begin
        p := random(nsac) + 1;
        Tirage[i] := sac[p];
        sac[p] := sac[nsac];
        nsac := nsac-1;
    end;
END;
```

On peut remplacer `nsac` par `NbNumTirage - i + 1`.

3.
```
FUNCTION BonNumero (n : integer) : integer;
VAR i, j, k : integer ;
BEGIN
    k := 0;
    for i := 1 to NbNumTirage do
      for j := 1 to NbNumTirage do
        if choix[n,i] = Tirage[j] then k := k+1;
    BonNumero := k;
END;
```

4.
```
PROCEDURE Resultats;
VAR i : integer;
BEGIN
    for i := 1 to NbJoueurs do
      NbBonsNumeros[i] := BonsNumeros(i);
END;
```

5.
```
FUNCTION NbGagnants (n : integer) : integer;
VAR i, s : integer;
BEGIN
    s := 0;
    for i := 1 to NbJoueurs do
      if NbBonsNumero[i] = n then s := s + 1;
    NbGagnants := s;
END;
```

```
6.  PROCEDURE CalculGains;
    VAR mise : real;
        i, n : integer;
    BEGIN
        mise := PrixUnJeu * NbJoueurs;

        for i := NbNumTirage downto DepartGain do
        begin
          n := NbGagnants(i);
          if n = 0
          then TabGain[i] := 0
          else begin
                 mise := mise / 2;
                 TabGain[i] := mise / n;
               end;
        end;

        GainOrganisateur := mise;
    END;
```

6. Calcul de π

6.1 Proportion de points dans un cercle

On considère un carré Q dans lequel on positionne aléatoirement un grand nombre de points. Le rapport entre le nombre de points figurant dans le cercle inscrit dans Q et le nombre de points tirés s'approche du rapport entre la surface du disque inscrit dans Q et la surface du carré Q. On en déduit une valeur approchée de π.

Correction

Soit r le rayon du cercle. L'aire du cercle est πr^2 et l'aire du carré est $4r^2$. Le rapport est donc $\frac{\pi r^2}{4r^2} = \frac{\pi}{4}$. Pour simplifier on prend $r = 1$, et on se limite au quart de plan $\mathbb{R}^+ \times \mathbb{R}^+$; le rapport $\frac{\pi r^2/4}{4r^2/4}$ est le même.

`random(m)` fournit un entier entre 0 et $m-1$, tandis que `random` sans paramètre fournit un réel entre 0 et 1.

Pour savoir si un point $(x, y) \in$ au quart de cercle on teste si $\sqrt{x^2+y^2} \leq 1$. Comme le calcul de la racine carrée est couteux, on se contente de faire le test équivallent $x^2 + y^2 \leq 1$.

n est le nombre total de points tirés (tous dans le quart de carré) ; c est le nombre de ces points qui est dans le quart de cercle.

```
FUNCTION calcpi (n : integer) : real;
VAR i, c : integer;
    x, y : real;
BEGIN
    c := 0;
    for i := 1 to n do
    begin
        x := random; y := random;
        if sqr(x) + sqr(y) <= 1 then c := c + 1;
    end;
    calcpi := 4 * c / n;
END;
```

6.2 Calcul itératif

$$\frac{\pi}{4} \simeq 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \cdots$$

Faire une fonction `calcpi` qui calcule la valeur approchée de π en s'arrêtant lorsque le terme $\frac{1}{x}$ est plus petit que ε.

Faire un programme qui lit ε réel, appelle `calcpi` puis affiche le résultat.

Correction

```
PROGRAM calcule_pi;

FUNCTION calcpi (e : real) : real;
VAR s, q, r, t : real;
BEGIN
    { init }
    s := 1.0; { signe }
    q := 1.0; { diviseur }
    r := 1.0; { somme partielle }

    repeat
      s := -s;     { inverse le signe }
      q := q + 2;  { diviseur de 2 en 2 }
      t := s / q;  { terme }
      r := r + t;  { somme partielle }
    until abs(t) < e;

    calcpi = r*4;
END;

VAR e : real;
BEGIN
    write ('epsilon = '); readln (e);
    writeln ('pi = ', calcpi(e));
END.
```

7. Liste chaînée

On dispose des types suivants :

```
TYPE ptr = ^maillon;
     maillon = record
                 val  : integer;
                 suiv : ptr;
               end;
```

On suppose qu'une liste chaînée se termine toujours par `nil`. Une liste peut être vide, son début étant `nil`.

Rappel On n'a jamais le droit de tester dans la même expression si (un pointeur est non `nil`) et (son contenu).

7.1 Créer un maillon et l'insérer en tête

```
Function creerTete (d : ptr; v : integer) : ptr;
```

Créé un maillon, stocke `v` dans le champ `val`, insère le maillon en tête de liste puis renvoie le début de la liste chaînée résultante.

Correction

```
Function creerTete (d : ptr; v : integer) : ptr;
Var q : ptr;
Begin
    new(q) ; q^.val := v; q^.suiv := d;
    creerTete := q;
End;
```

7.2 Créer une liste

```
Function creerListe (n : integer) : ptr;
```

Créé une liste chaînée (bien entendu terminée par `nil`) de `n` maillons à 1, puis renvoie l'adresse de début.

Correction

```
Function creerListe (n : integer) : ptr;
Var p : ptr;
    i : integer;
Begin
    p := nil;
    for i := 1 to n do
        p := creerTete (p, 1);
    creerListe := p;
End;
```

7.3 Suppression du maillon de tête

`Function supprTete (d : ptr) : ptr;`

Supprime le maillon de tête puis renvoie le début de la liste chaînée résultante.

Correction

```
Function supprTete (d : ptr) : ptr;
Begin
    if d = nil
    then supprTete := nil
    else begin supprTete := d^.suiv; dispose (d); end;
End;
```

7.4 Vider la liste

`Procedure viderListe (d : ptr);`

Supprime tous les maillons de la liste.

Correction

```
Procedure viderListe (d : ptr);
Begin
    while d <> nil do d := supprTete (d);
End;
```

7.5 Afficher les valeurs d'une liste

`Procedure affiListe (d : ptr);`

Affiche les champs `val` des maillons d'une liste chaînée.

Correction

```
Procedure affiListe (d : ptr);
Begin
    while d <> nil do
    begin writeln (d^.val); d := d^.suiv; end;
End;
```

7.6 Recherche d'un maillon

`Function rechercher (d : ptr; v : integer) : ptr;`

Renvoie l'adresse du 1^{er} maillon dont le champ `val` est égal à `v`, sinon `nil`.

Correction

```
Function rechercher (d : ptr; v : integer) : ptr;
Begin
    rechercher := nil;
    while d <> nil do
        if d^.val = v
        then begin rechercher := d; d := nil; end
        else d := d^.suiv;
End;
```

7.7 Suppression d'un maillon

`Function supprimer (d : ptr; v : integer) : ptr;`

Cherche le 1^{er} maillon dont le champ `val` est égal à `v`, le supprime, met à jour le chaînage dans la liste, puis renvoie le début de la liste chaînée résultante.

Correction

```
Function supprimer (d : ptr; v : integer) : ptr;
Var p, q : ptr;
Begin
    supprimer := d;
    if d <> nil
    then if d^.val = v
    then begin
           { Valeur en tete --> on modifie le début }
           supprimer := d^.suiv; dispose(d);
         end
    else begin
           p := d; q := d^.suiv;
           while q <> nil do
             if q^.val = v
             then begin p^.suiv := q^.suiv; dispose(q); q := nil; end
             else begin p := q; q := q^.suiv; end;
         end;
End;
```

7.8 Dupliquer une liste

`Function dupliquer (d : ptr) : ptr;`

Duplique une liste chaînée (créé des maillons, recopie les bonnes valeurs dans les champs `val` et chaîne ces nouveaux maillons dans le même ordre), puis renvoie le début de la nouvelle liste.

Correction

```
Function dupliquer (d : ptr) : ptr;
Var p : ptr;
Begin
    if d = nil
    then dupliquer := nil
    else begin
           new (p); dupliquer := p;
           p^ := d^; d := d^.suiv;

           while d <> nil do
           begin
             new (p^.suiv); p := p^.suiv;
             p^ := d^; d := d^.suiv;
           end;

           { On a bien p^.suiv = nil puisqu'on a fait p^ := d^ }
         end;
End;
```

7.9 Concaténer 2 listes

`Function concatener (d1, d2 : ptr) : ptr;`

Concatène deux listes chaînées en rattachant le début de la seconde liste à la fin de la première, puis renvoie le début de la nouvelle liste.

Correction

```
Function Function concatener (d1, d2 : ptr) : ptr;
Begin
    if d1 = nil then concatener := d2
    else if d2 = nil then concatener := d1  { gain de temps }
    else begin
           concatener := d1;
           while d1^.suiv <> nil do d1 := d1^.suiv;
           d1^.suiv := d2;
         end;
End;
```

8. Liste chaînée triée

On dispose des types définis au §7.

On suppose que les listes chaînées sont triées par ordre croissant sur le champ `val`, et on s'arrange pour conserver cette propriété. On utilise le fait qu'une liste est triée pour s'arrêter le plus tôt possible lors d'un parcours de la liste.

8.1 Recherche d'un maillon

`Function rechercher (d : ptr; v : integer) : ptr;`

Renvoie l'adresse du 1^{er} maillon dont le champ `val` est égal à `v`, sinon `nil`.

Correction

```
Function rechercher (d : ptr; v : integer) : ptr;
Begin
    rechercher := nil;
    while d <> nil do
        if d^.val < v
        then d := d^.suiv
        else if d^.val = v
        then begin rechercher := d; d := nil; end
        else d := nil;
End;
```

8.2 Suppression d'un maillon

`Function supprimer (d : ptr; v : integer) : ptr;`

Cherche le 1^{er} maillon dont le champ `val` est égal à `v`, le supprime, met à jour le chaînage dans la liste, puis renvoie le début de la liste chaînée résultante.

Correction

```
Function supprimer (d : ptr; v : integer) : ptr;
Var p, q : ptr;
Begin
    supprimer := d;
    if d <> nil
    then if d^.val = v
    then begin supprimer := d^.suiv; dispose(d); end
    else begin
           p := d; q := d^.suiv;
           while q <> nil do
             if q^.val < v
             then begin p := q; q := q^.suiv; end
             else if q^.val = v
             then begin p^.suiv := q^.suiv; dispose(q); q := nil; end
             else q := nil;
         end;
End;
```

8.3 Insertion d'un maillon

`Function inserer (d, m : ptr) : ptr;`

Reçoit un maillon `m` qui est non `nil` et qui n'appartient pas à la liste `d`, trouve l'endroit où insérer `m` d'après son champ `val`, l'insère, puis renvoie le début de la liste chaînée résultante.

Correction

```
Function inserer (d, m : ptr) : ptr;
Var p, q : ptr;
Begin
    if d = nil
    then begin m^.suiv := nil; inserer := m; end
    else if m^.val <= d^.val
    then begin m^.suiv := d; inserer := m; end
    else begin
           p := d; q := d^.suiv; inserer := d;
           while q <> nil do
               if q^.val < m^.val
               then begin p := q; q := q^.suiv; end
               else q := nil;
           m^.suiv := p^.suiv; p^.suiv := m;
         end;
End;
```

8.4 Tri par insertion

`Function triInsertion (d : ptr) : ptr;`

Fait le tri par insertion de la liste `d`, puis renvoie le début de la liste chaînée résultante.

Correction

```
Function triInsertion (d : ptr) : ptr;
Var p, q : ptr;
Begin
    p := nil;
    while d <> nil do
      begin q := d; d := d^.suiv; p := inserer (p, q); end;
    triInsertion := p;
End;
```

8.5 Inversion

`Function inverser (d : ptr) : ptr;`

Reçoit le début de la liste chaînée dans l'ordre croissant, inverse l'ordre de chaînage des maillons puis renvoie le début de la liste chaînée dans l'ordre décroissant.

Correction

```
Function inverser (d : ptr) : ptr;
Var p, q, r : ptr;
Begin
    p := nil; q := d;
    while q <> nil do
    begin
      r := q^.suiv; q^.suiv := p;
      p := q; q := r;
    end;
    inverser := p;
End;
```

8.6 Décomposition

`Function decompose (d : ptr) : ptr;`

Décompose la liste chaînée en deux listes chaînant chacune un maillon sur deux de la liste initiale. Le début de la première nouvelle liste reste `d`, et le début de la seconde nouvelle liste est renvoyé en résultat.

Correction

```
Function decompose (d : ptr) : ptr;
Var p, q : ptr;
Begin
    if d = nil then decompose := nil
    else begin
           p := d; q := d^.suiv ; decompose := q;
           while q <> nil do
           begin
              p^.suiv := q^.suiv; p := q; q := q^.suiv;
           end;
         end;
End;
```

8.7 Fusion

`Function fusion (d1, d2 : ptr) : ptr;`

Effectue la fusion de 2 listes triées, le résultat étant le début de la liste fusionnée.

Correction

```
Function fusion (d1, d2 : ptr) : ptr;
Var p : ptr;
Begin
    if d1 = nil then fusion := d2
    else if d2 = nil then fusion := d1
    else begin
           if d1^.val < d2^.val
           then begin p := d1; d1 := d1^.suiv; end
           else begin p := d2; d2 := d2^.suiv; end;
           fusion := p;

           while (d1 <> nil) and (d2 <> nil) do
             if d1^.val < d2^.val
             then begin p^.suiv := d1; p := d1; d1 := d1^.suiv; end
             else begin p^.suiv := d2; p := d2; d2 := d2^.suiv; end;

           id d1 = nil then p^.suiv := d2
                       else p^.suiv := d1;
         end;
End;
```

9. Calculs récursifs

On demande une implémentation récursive : les procédures ou fonctions se rappellent elles-mêmes et n'utilisent aucune boucle.

9.1 Factorielle

`Function fact (n : integer) : integer`

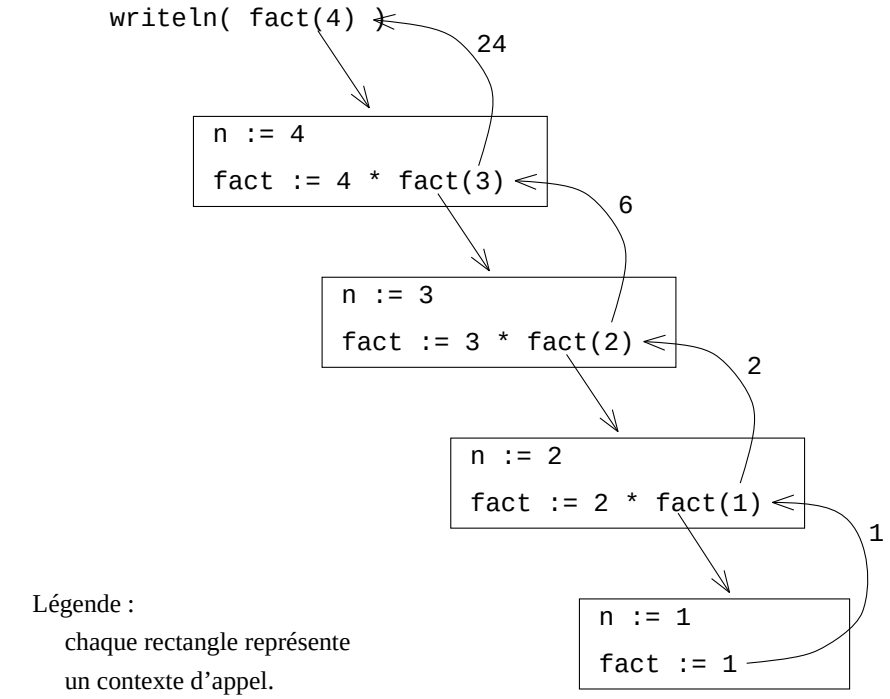
Calcule la factorielle de `n` positif d'après la définition récurrente :

$$n! = \begin{cases} 1 & \text{si } n \leq 1 \\ n \,.\, (n-1)! & \text{si } n > 1 \end{cases}$$

Correction

```
Function fact (n : integer) : integer;
Begin
    if n <= 1 then fact := 1
              else fact := n * fact (n-1);
End;                            { récursif }
```

Appels récursifs dans la fonction `fact`



Légende :
chaque rectangle représente un contexte d'appel.

9.2 PGCD

```
Function pgcd (a, b : integer) : integer;
```

Calcule le plus grand commun diviseur entre 2 nombres a et b positifs, en faisant décroître ce couple jusqu'à l'obtention du pgcd.

1) Par soustractions

$$\text{pgcd}(a,b) = \begin{cases} a & \text{si } a = b \text{ ou } b = 0 \\ b & \text{si } a = 0 \\ \text{pgcd}(a, b-a) & \text{si } a < b \\ \text{pgcd}(a-b, b) & \text{si } a > b \end{cases}$$

Ex $\text{pgcd}(85,25) = (60,25) = (35,25) = (10,25) = (10,15) = (10,5) = (5,5) = 5$

2) par divisions

$$\text{pgcd}(a,b) = \begin{cases} a+b & \text{si } a * b = 0 \\ \text{pgcd}(a, b \bmod a) & \text{si } a \leq b \\ \text{pgcd}(a \bmod b, b) & \text{si } a > b \end{cases}$$

Ex $\text{pgcd}(85,25) = (35,25) = (10,25) = (10,5) = (0,5) = 5$

Correction

```
1)  Function pgcd (a, b : integer) : integer;
    Begin
        if (a = b) or (b = 0)
        then pgcd := a
        else if a = 0
        then pgcd := b
        else if a < b
        then pgcd := pgcd(a,b-a)
        else pgcd := pgcd(a-b,b);
    End;

2)  Function pgcd (a, b : integer) : integer;
    Begin
        if a * b = 0
        then pgcd := a + b
        else if a <= b
        then pgcd := pgcd(a,b mod a)
        else pgcd := pgcd(a mod b,b);
    End;
```

9.3 Nombre de combinaisons

`Function Cnp (n, p : integer) : integer;`

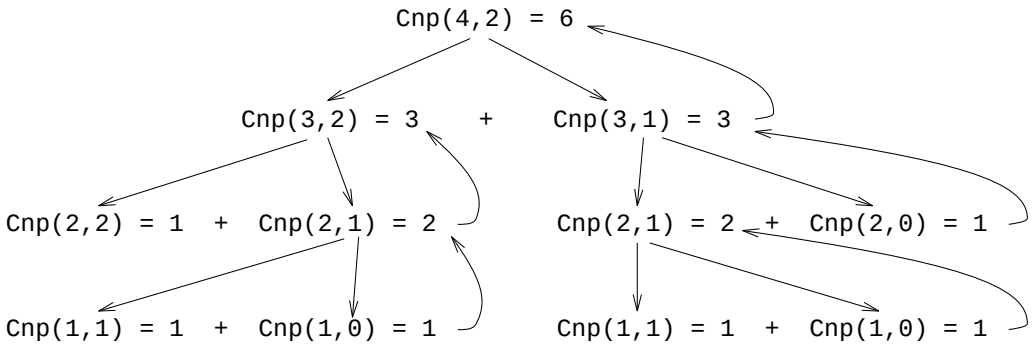
Calcule le nombre de combinaisons C_n^p de 2 nombres positifs d'après la définition récurrente (triangle de Pascal) :

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{si } 0 < p < n \\ 0 & \text{si } p > n \end{cases}$$

Correction

```
Function Cnp (n, p : integer) : integer;
Begin
    if (p = 0) or (p = n)
    then Cnp := 1
    else if (0 < p) and (p < n)
    then Cnp := Cnp (n-1, p) + Cnp (n-1, p-1)
                  { 2 appels récursif }
    else Cnp := 0;
End;
```

Appels récursifs dans la fonction `Cnp`



Ce calcul ce C_n^p est donc très inefficace, car de nombreux termes intermédiaires sont évalués plusieurs fois, et le nombre d'appels récursif croît très vite.

9.4 Fibonacci

`Function fibo (n : integer) : integer;`

Calcule le terme U_n de la suite de Fibonacci définie par :

$$\begin{cases} U_0 = U_1 = 1 \\ U_n = U_{n-1} + U_{n-2} \end{cases}$$

Les premiers termes sont 1, 1, 2, 3, 5, 8, 13, 21, ... Faire une version récursive et une version itérative (sans tableau). Comparer leur efficacité.

Correction

```
Function fibo (n : integer) : integer;   { Récursif }
Begin
    if n <= 1 then fibo := 1
              else fibo := fibo(n-1) + fibo(n-2);
End;

Function fibo (n : integer) : integer;   { Itératif }
VAR u, v, t : integer;
Begin
    if n <= 1 then fibo := 1
    else begin
           u := 1; v := 1;
           for k := 2 to n do
             begin t := u+v ; u := v; v := t; end;
           fibo := v;
         end;
End;
```

La version récursive est bien plus simple à écrire, mais le nombre d'appels « explose » et les termes intermédiaires sont recalculés beaucoup de fois : cette version récursive est donc très inefficace, contrairement à la version itérative.

Hors exercice Voici une méthode récursive efficace ; il s'agit de la méthode itérative, qu'on a « récursivisée ».

```
Function fibo (n, a, b : integer) : integer;
Begin
    if n = 0 then fibo := a
    else if n = 1 then fibo := b
    else fibo := fibo (n-1, b, a+b);
End;
```

Exemple d'appel : `writeln (fibo (5,1,1));` Les paramètres deviendront :

```
5  1  1  ;  4  1  2  ;  3  2  3  ;  2  3  5  ;  1  5  8
```

d'où le résultat 8.

9.5 Miroir

`Procedure miroir;`

Lit au clavier une suite de caractères terminée par un `'.'` et les affiche dans l'ordre inverse (y compris le point).

Correction

```
Procedure miroir;
Var c : char;
Begin
    read(c);
    if c <> CFin
    then miroir
    else readln;
    write(c);
End;
```

10. Récursivité sur liste chaînée triée

On dispose des types définis au §7. Comme au §8 on suppose que les listes chaînées sont triées par ordre croissant sur le champ `val`.

On demande une implémentation récursive : les procédures ou fonctions se rappellent elles-mêmes et n'utilisent aucune boucle.

10.1 Afficher les valeurs d'une liste

```
Procedure affiListe (d : ptr);
```

Affiche les champs `val` des maillons d'une liste (a) dans l'ordre croissant (b) dans l'ordre décroissant.

Correction

```
(a) Procedure affiListe1 (d : ptr);
    Begin
        if d <> nil then
        begin writeln (d^.val); affiListe (d^.suiv); end;
    End;

(b) Procedure affiListe2 (d : ptr);
    Begin
        if d <> nil then
        begin affiListe (d^.suiv); writeln (d^.val); end;
    End;
```

10.2 Dupliquer une liste

```
Function dupliquer (d : ptr) : ptr;
```

Duplique une liste chaînée (créé des maillons, recopie les bonnes valeurs dans les champs `val` et chaîne ces nouveaux maillons dans le même ordre), puis renvoie le début de la nouvelle liste.

Correction

```
Function dupliquer (d : ptr) : ptr;
Var q : ptr;
Begin
    if d = nil then dupliquer := nil
    else begin
           new (q); q^ := d^;
           dupliquer := q;
           q^.suiv := dupliquer (d^.suiv);
End;
```

10.3 Inversion

`Function inverser (d, p : ptr) : ptr;`

Inverse l'ordre de chaînage des maillons puis renvoie le début de la liste chaînée dans l'ordre décroissant. En cours d'appel récursif, `d` est le début de la portion de liste à inverser, `p` est le début de la portion de liste déjà inversée. L'appel principal est donc `deb := inverser (deb, nil)` où `deb` est le début de la liste à inverser.

Correction

```
Function inverser (d, p : ptr) : ptr;
Var q : ptr;
Begin
    if d = nil then inverser := p
    else begin
           q := d^.suiv; d^.suiv := p;
           inverser := inverser (q, d);
         end;
End;
```

10.4 Dupliquer en inversant l'ordre

`Function dupInv (d, p : ptr) : ptr;`

Duplique une liste chaînée en inversant l'ordre de chaînage des maillons (sans appeler `dupliquer` ni `inverser`) puis renvoie le début de la nouvelle liste. En cours d'appel récursif, `d` est le début de la liste à dupliquer en inversant l'ordre, `p` est le début de la liste déjà dupliquée dans l'ordre inverse. L'appel principal est donc `deb2 := dupInv (deb1, nil)`.

Correction

```
Function dupInv (d, p : ptr) : ptr;
Var q : ptr;
Begin
    if d = nil then dupInv := p
    else begin
           new (q); q^ := d^;
           q^.suiv := p;
           dupInv := dupInv (d^.suiv, q);
         end;
End;
```

10.5 Insertion d'un maillon

`Function inserer (d, m : ptr) : ptr;`

Reçoit un maillon `m` qui est non `nil` et qui n'appartient pas à la liste `d`, trouve l'endroit où insérer `m` d'après son champ `val`, l'insère, puis renvoie le début de la liste chaînée résultante.

Correction

```
Function inserer (d, m : ptr) : ptr;
Begin
    if d = nil
    then begin m^.suiv := nil; inserer := m; end
    else if m^.val <= d^.val
    then begin m^.suiv := d; inserer := m; end
    else begin
           d^.suiv := inserer (d^.suiv, m);
           inserer := d;
         end;
End;
```

10.6 Détacher un maillon

`Function detacher (d : ptr; v : integer; var m : ptr) : ptr;`

Cherche le 1^{er} maillon dont le champ `val` est égal à `v`, met son adresse dans `m`, le détache de la liste, puis renvoie le début de la liste chaînée résultante.

Correction

```
Function detacher (d : ptr; v : integer; var m : ptr) : ptr;
Begin
    if d = nil
    then begin m := nil ; detacher := nil; end
    else if v < d^.val
    then begin m := nil ; detacher := d; end
    else if v = d^.val
    then begin m := d ; detacher := d^.suiv; end
    else begin
           d^.suiv := detacher (d^.suiv, v, m);
           detacher := d;
         end;
End;
```

10.7 Décomposition

`Function decompose (d : ptr) : ptr;`

Décompose la liste chaînée en deux listes chaînant chacune un maillon sur deux de la liste initiale. Le début de la première nouvelle liste reste `d`, et le début de la seconde nouvelle liste est renvoyé en résultat.

Correction Le résultat de la fonction est le maillon suivant (ou `nil`), donc l'appel `decompose (d^.suiv)` renvoie le suivant du suivant (en gérant `nil`) :

```
Function decompose (d : ptr) : ptr;
Begin
    if d = nil then decompose := nil
    else begin
           decompose := d^.suiv;
           d^.suiv := decompose (d^.suiv);
         end;
End;
```

10.8 Fusion

`Function fusion (d1, d2 : ptr) : ptr;`

Effectue la fusion de 2 listes triées, le résultat étant le début de la liste fusionnée.

Correction Le résultat de la fonction est soit `d1`, soit `d2`.

```
Function fusion (d1, d2 : ptr) : ptr;
Var p : ptr;
Begin
    if d1 = nil then fusion := d2
    else if d2 = nil then fusion := d1
    else if d1^.val < d2^.val
         then begin
                fusion := d1;
                d1^.suiv := fusion (d1^.suiv, d2)
              end
         else begin
                fusion := d2;
                d2^.suiv := fusion (d1, d2^.suiv);
              end;
End;
```

10.9 Tri par fusion

`Function triFusion (d : ptr) : ptr;`

Fait le tri par fusion de la liste `d` en appelant les fonctions `decompose` et `fusion` puis renvoie le début de la liste chaînée résultante.

Correction

```
Function triFusion (d : ptr) : ptr;
Var c : ptr;
Begin
    if d = nil then triFusion := nil
    else if d^.suiv = nil then triFusion := d
    else begin
           c := decompose (d);
           triFusion := fusion (triFusion (c), triFusion (d));
         end;
End;
```

Les fonctions `decompose` et `fusion` sont linéaires : chacune parcourt une seule fois chaque maillon de la liste.

Si la liste `d` comporte n maillons, il faudra $\log_2 n$ étapes pour la décomposer en n listes de 1 éléments, puis $\log_2 n$ étapes pour les fusionner en une liste de n éléments. Le coût total est donc $n \times 2 \times \log_2 n$.

Conclusion : la complexité est $O(n \log n)$ et l'algorithme est optimal. Le tri par fusion ne s'implémente que récursivement sur liste chaînée. Sur des vecteurs, il vaut mieux utiliser le *quicksort*.