Exercice 9 ** : Traitement d'image (CC mi-parcours oct. 2012)

Un filtrage est un traitement qui s'applique globalement à toute l'image. Un algorithme de filtrage calcule la nouvelle valeur du pixel en tenant compte des valeurs des pixels voisins. On considère ici un voisinage carré, centré sur le pixel que l'on calcule. Ce carré peut avoir différentes tailles : 3x3 (rayon 1), 5x5 (rayon 2), 7x7 (rayon 3), etc. Dans cet exercice, on considère le filtre « flou moyenneur », qui attribue au pixel la moyenne des valeurs dans le voisinage (sauf pour les pixels en bordure, qui eux deviennent noirs, pour simplifier l'algorithme).

Image originale

Après un filtre « flou moyenneur » de rayon 3 (voisinage 7x7)

Dans tout l'exercice, nous utiliserons le langage algorithmique. Nous supposerons que l'image est en niveaux de gris (chaque pixel est donc caractérisable par un seul entier) et que l'image est stockée dans une matrice (tableau 2D) d'entiers. Une image sera par exemple déclarée de la façon suivante :

monImage : tableau [1..HAUTEUR][1..LARGEUR] d'entiers

On peut accéder au niveau de gris du pixel situé sur la ligne i et la colonne j en utilisant deux fois l'opérateur crochets : monImage[i][j]. Rappel : dans le langage algorithmique, les lignes et de colonnes sont numérotées à partir de 1.

a) Ecrire la procédure flouNaif qui prend en paramètre une image de départ (mode donnée), une image d'arrivée (mode résultat) et le rayon *r* du voisinage (entier, mode donnée). Pour chaque pixel de l'image (sauf ceux des bords), cette procédure parcourt les $(2r + 1)^2$ voisins pour calculer la moyenne du niveau de gris dans le voisinage. La procédure ne fera pas d'allocation mémoire, on suppose que les deux images passées en paramètres ont déjà été allouées en mémoire avant l'appel. Vous pourrez appeler la procédure suivante sans en donner le code :

Procédure mettrePixelsDuBordEnNoir (monImage : tableau [1..HAUTEUR][1..LARGEUR] d'entiers, epaisseurBord : entier)
Paramètres en mode donnée : epaisseurBord
Paramètres en mode donnée-résultat : monImage
Préconditions : epaisseurBord > 0, monImage a déjà été allouée en mémoire
Postconditions : le niveau de gris des pixels du bord de l'image sont mis à 0. On appelle bord de l'image les lignes de pixels allant de 1 à epaisseurBord (bordure du bas), les colonnes allant de 1 à epaisseurBord (bordure de gauche), les colonnes allant de LARGEUR – epaisseurBord + 1 à LARGEUR (bordure de droite), les lignes allant de HAUTEUR – epaisseurBord + 1 à HAUTEUR (bordure du haut). Les autres pixels sont inchangés.

Procédure flouNaif (imageOrig: tableau [1..HAUTEUR][1..LARGEUR] d'entiers, imageFloue : tableau [1..HAUTEUR][1..LARGEUR] d'entiers, r : entier)
Paramètres en mode donnée : imageOrig, r
Paramètres en mode résultat : imageFloue
Préconditions : r > 0, imageOrig et imageFloue ont déjà été allouées en mémoire
Postconditions : A l'exception des pixels situés sur les bords de l'image, chaque pixel de imageFloue a son niveau de gris qui correspond à la moyenne des niveaux de gris de ses voisins, le voisinage étant un carré de largeur 2r+1 centré sur le pixel. Les pixels situés sur les bords, pour lesquels le voisinage n'est pas complet, sont noirs.

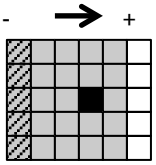
```
Variables locales :
lig, col, l, c, somme, nbvoisins : entiers

Début
        mettrePixelsDuBordEnNoir(imageFloue, r)
        nbvoisins ← (2*r+1)*(2*r+1)
Pour lig allant de (r+1) à (HAUTEUR-r) par pas de 1 Faire
        Pour col allant de (r+1) à (LARGEUR-r) par pas de 1 Faire
                somme ← 0
                Pour l allant de (lig-r) à (lig+r) par pas de 1 Faire
                        Pour c allant de (col-r) à (col+r) par pas de 1 Faire
                                somme ← somme + imageOrig[l][c]
                        FinPour
                FinPour
                imageFloue[lig][col] ← somme / nbvoisins
        FinPour
FinPour
Fin flouNaif
```

b) Si l'image est de dimensions H x L et que le rayon du filtre est r, combien d'additions de niveaux de gris effectue-t-on avec cet algorithme naïf ? Répondez en fonction de L, H et r, puis faîtes l'application numérique pour r = 3, H = 200 et L = 400.

$(H - 2r)(L - 2r)(2r + 1)^2$ additions de niveaux de gris
Application numérique : 194 * 394 * 49 = 3 745 364 additions de niveaux de gris

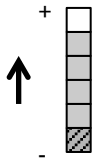
c) On peut optimiser cet algorithme en remarquant que la somme des niveaux du voisinage carré peut se décomposer comme la somme des sommes des colonnes. Quand on déplace la fenêtre d'un vers la droite, il suffit d'ajouter la somme de la nouvelle colonne et de retrancher la somme de la colonne la plus à gauche :



Exemple avec un voisinage 5x5 (r = 2) :

Pour calculer la somme relative au pixel noir, on part de la somme calculée pour le pixel précédent (zone grisée), on retire la colonne

Les sommes des colonnes peuvent être mises à jour efficacement quand on change de ligne, en retranchant le pixel du bas de la colonne et en ajoutant le pixel juste au-dessus de la colonne :



Voici la version optimisée du filtrage en langage algorithmique :

Procédure flouOptim (imageOrig: tableau [1..HAUTEUR][1..LARGEUR] d'entiers, imageFloue : tableau [1..HAUTEUR][1..LARGEUR] d'entiers, r : entier)
Paramètres en mode donnée : imageOrig, r
Paramètres en mode résultat : imageFloue
Préconditions : r > 0, imageOrig et imageFloue ont déjà été allouées en mémoire
Postconditions : A l'exception des pixels situés sur les bords de l'image, chaque pixel de imageFloue a son niveau de gris qui correspond à la moyenne des niveaux de gris de ses voisins, le voisinage étant un carré de largeur 2r+1 centré sur le pixel. Les pixels situés sur les bords, pour lesquels le voisinage n'est pas complet, sont noirs.

```
Variables locales :
  lig, col, somme, nbvoisins, largeurVoisinage : entiers
  sommeColonne : tableau [1..LARGEUR] d'entiers

Début
        mettrePixelsDuBordEnNoir(imageFloue, r)
  nbvoisins ← (2*r+1)*(2*r+1)
  largeurVoisinage ← 2*r + 1

  {On initialise la somme des colonnes}
        Pour col allant de 1 à LARGEUR par pas de 1 Faire
                sommeColonne[col] ← 0
                Pour lig allant de 1 à largeurVoisinage par pas de 1 Faire
                        sommeColonne[col] ← sommeColonne[col] + imageOrig[lig][col]
                FinPour
        FinPour

        Pour lig allant de (r+1) à (HAUTEUR-r) par pas de 1 Faire
        {On traite séparément le premier pixel à flouter de la ligne, pour initialiser
         la somme intercolonnes}
        somme ← 0
        Pour col allant de 1 à largeurVoisinage par pas de 1 Faire
                somme ← somme + sommeColonne[col]
        FinPour
        imageFloue[lig][r + 1] ← somme / nbvoisins

        {Pour les autres pixels de la ligne, la somme intercolonnes est mise à jour
         de façon incrémentale}
        Pour col allant de (r+2) à (LARGEUR-r) par pas de 1 Faire
                somme ← somme – sommeColonne[col – r - 1] + sommeColonne[col+r]
                imageFloue[lig][col] ← somme / nbvoisins
        FinPour

        Si (lig < HAUTEUR – r)
                {On prépare les sommes des colonnes pour la ligne suivante,
                de façon incrémentale (on ne le fait pas si on vient de traiter la dernière ligne)}
                Pour col allant de 1 à LARGEUR par pas de 1 Faire
                        sommeColonne[col] ← sommeColonne[col] – imageOrig[lig-r][col]

                         + imageOrig[lig+r+1][col]
                FinPour
        FinSi

  FinPour

Fin flouOptim
```

d) Si l'image est de dimensions *H* x *L* et que le rayon du filtre est *r*, combien d'additions de niveaux de gris effectue-t-on avec cet algorithme optimisé ? Dans votre comptage, considérez qu'une soustraction de niveaux de gris compte pour une addition. Répondez en fonction de *L*, *H* et *r*, puis faîtes l'application numérique pour r = 3, H = 200 et L = 400.

- initialisation du tableau sommeColonne :	(2r + 1)L
- initialisation de 'somme' en début de ligne :	(2r + 1)(H – 2r)
- mise à jour de 'somme' le long d'une ligne :	2(L – 2r – 1)(H – 2r)
- mise à jour du tableau sommeColonne en fin de ligne :	2L(H – 2r – 1)

Total = (2r + 1) (L + H – 2r) + 2(L – 2r – 1)(H - 2r) + 2L(H – 2r – 1)
Total = 4LH - (6r + 1)L - (2r + 1)H + $4r^2$ + 2r additions de niveaux de gris

Application numérique :
320000 – 7600 - 1400 + 36 + 6 = 311 042, soit environ 8% du nombre d'additions effectuées par l'algorithme naïf. C'est au prix, ici, du stockage en mémoire d'un tableau de L entiers en plus des deux images.

e) On décide d'implémenter cet algorithme optimisé en C en stockant les images sous forme de matrices (tableaux 2D) d'unsigned char. Peut-on utiliser aussi le type unsigned char pour les éléments du tableau sommeColonne ? Justifiez votre réponse.

Non, car le type unsigned char ne permet que de stocker que des nombres strictement inférieurs à 256. Ici, chaque élément du tableau sommeColonne est une somme de 5 unsigned char, chacun étant compris entre 0 et 255. La somme des 5 a donc de grandes chances d'excéder 255. Si on utilisait le type unsigned char pour stocker cette somme, on aurait un dépassement de capacité et le calcul de la moyenne serait faux.

Algorithmes sur les fichiers

Exercice 1 ** : Fusion de deux monotonies sur fichiers (CC mi-parcours nov. 2009)

On dispose de deux fichiers contenant chacun une séquence triée de réels. En d'autres termes, chacun des deux fichiers contient une monotonie et une seule – il s'agit donc d'un problème plus simple que celui du tri fusion sur fichiers. On veut écrire une procédure qui fusionne ces deux monotonies et écrit la monotonie résultante dans un troisième fichier. Cette procédure prend comme *seuls* paramètres les noms des deux fichiers d'entrée (**nomfic1** et **nomfic2**) et le nom du fichier de sortie (**nomficSortie**).

Préconditions :

- nomfic1 et nomfic2 sont des fichiers binaires qui contiennent chacun une séquence triée (monotonie) de nombres au format IEEE 754 double précision. Si l'un de ces fichiers ne peut pas être ouvert (par exemple parce qu'il n'existe pas), alors le programme se termine avec un code d'échec.
- Les deux séquences peuvent être de longueur différente, mais elles sont au moins de longueur 1 : autrement dit, chaque fichier contient au moins un élément.

Postconditions :

- Un nouveau fichier binaire nommé nomficSortie est créé (s'il existait déjà, son contenu est écrasé). Ce fichier contient une monotonie correspondant à la fusion des deux monotonies contenues dans nomfic1 et nomfic2. Si ce fichier ne peut pas être créé (par exemple à cause d'un problème de droits insuffisants dans le répertoire courant), le programme se termine avec un code d'échec.
- Les contenus des fichiers nomfic1 et nomfic2 sont inchangés.

Donnez le code C de cette procédure. Vous utiliserez les variables locales suivantes (à vous d'en préciser le type) : **e1, e2, fic1, fic2, ficSortie, succeslect1, succeslect2**. Vous pouvez en ajouter d'autres si nécessaire.

On rappelle l'ordre des paramètres pour les sous-programmes suivants, à vous de déterminer lequel ou lesquels doivent être utilisés ici :

- fscanf(pointeur sur le flot, chaîne de format, adresse variable 1, adresse variable 2...)
- fprintf(pointeur sur le flot, chaîne de format, valeur variable 1, valeur variable 2...)
- fread(adresse 1er bloc, taille d'un bloc, nombre de blocs, pointeur sur le flot)
- fwrite(adresse 1er bloc, taille d'un bloc, nombre de blocs, pointeur sur le flot)

```
/* Preconditions :
   - nomfic1 et nomfic2 sont des fichiers binaires qui contiennent
     chacun une sequence triee de nombres au format double (monotonie).
   - les deux sequences peuvent etre de longueur differente, mais
     elles sont au moins de longueur 1.
   Postcondition :
   - un nouveau fichier nomme nomficSortie est cree (s'il existait
     deja, son contenu est ecrase). Ce fichier contient la fusion
     des deux monotonies.  */
void fusionner(const char * nomfic1, const char * nomfic2, const char * nomficSortie)
{
  double e1, e2;
  int succeslect1 = 1, succeslect2 = 1;
  FILE *fic1, *fic2, *ficSortie;

  fic1 = fopen(nomfic1, "rb");
  if (fic1 == NULL)
    {
      printf("Impossible d'ouvrir le fichier %s en lecture \n",nomfic1);
      exit(EXIT_FAILURE);
    }

  fic2 = fopen(nomfic2, "rb");
  if (fic2 == NULL)
    {
      printf("Impossible d'ouvrir le fichier %s en lecture \n",nomfic2);
      exit(EXIT_FAILURE);
    }
```

```
  ficSortie = fopen(nomficSortie, "wb");
  if (ficSortie == NULL)
    {
      printf("Impossible d'ouvrir le fichier %s en ecriture \n",nomficSortie);
      exit(EXIT_FAILURE);
    }

  fread(&e1, sizeof(double), 1, fic1);
  fread(&e2, sizeof(double), 1, fic2);

  while ((succeselect1 != 0) && (succeselect2 != 0))
    {
      if (e1 < e2)
        {
          fwrite(&e1, sizeof(double), 1, ficSortie);
          if (fread(&e1, sizeof(double), 1, fic1) != 1) succeselect1 = 0;
        }
      else
        {
          fwrite(&e2, sizeof(double), 1, ficSortie);
          if (fread(&e2, sizeof(double), 1, fic2) != 1) succeselect2 = 0;
        }
    }

  if (succeselect1 == 0)
    {
      /* On a epuise le fichier1, il reste a recopier la fin du fichier2 */
      fwrite(&e2, sizeof(double), 1, ficSortie);
      while (fread(&e2, sizeof(double), 1, fic2) == 1)
        {
          fwrite(&e2, sizeof(double), 1, ficSortie);
        }
    }
  else
    {
      /* On a epuise le fichier2, il reste a recopier la fin du fichier1 */
      fwrite(&e1, sizeof(double), 1, ficSortie);
      while (fread(&e1, sizeof(double), 1, fic1) == 1)
        {
          fwrite(&e1, sizeof(double), 1, ficSortie);
        }
    }

  fclose(fic1);
  fclose(fic2);
  fclose(ficSortie);

}
```

Exercice 2 *** : Tri fusion sur fichiers

On considère le fichier non trié contenant la séquence d'éléments suivante :

65 5 89 56 7 15 28 2 98 33

f) Donnez le contenu des 3 fichiers (appelés A, B et X dans les diapositives de cours) intervenant dans le tri fusion du fichier ci-dessus, après chaque étape d'éclatement et chaque étape de fusion.

Fichier original : (65) (5) (89) (56) (7) (15) (28) (2) (98) (33)
Remarque : les parenthèses ne sont pas réellement dans le fichier, je les mets juste pour indiquer les différentes monotonies. On a donc au départ 10 monotonies de longueur 1.

Eclatement sur deux fichiers (1 monotonie sur 2 est copiée dans le fichier A, 1 sur 2 est copiée dans le fichier B) :
Fichier A : (65) (89) (7) (28) (98)
Fichier B : (5) (56) (15) (2) (33)

Fusion des monotonies 2 à 2 : la première du fichier A avec la première du fichier B, etc. On obtient 5 monotonies de longueur 2 que l'on écrit dans le fichier X.
Fichier X : (5 65) (56 89) (7 15) (2 28) (33 98)

Eclatement sur 2 fichiers (on écrase le contenu précédent des fichiers A et B) :
Fichier A : (5 65) (7 15) (33 98)
Fichier B : (56 89) (2 28)

Fusion dans le fichier X (on écrase le contenu précédent du fichier X) : on obtient 2 monotonies de longueur 4 et une de longueur 2.
Fichier X : (5 56 65 89) (2 7 15 28) (33 98)

Eclatement sur 2 fichiers (on écrase le contenu précédent des fichiers A et B) :
Fichier A : (5 56 65 89) (33 98)
Fichier B : (2 7 15 28)

Fusion dans le fichier X (on écrase le contenu précédent du fichier X) : on obtient 1 monotonies de longueur 8 et une de longueur 2.
Fichier X : (2 5 7 15 28 56 65 89) (33 98)

Eclatement sur 2 fichiers (on écrase le contenu précédent des fichiers A et B) :
Fichier A : (2 5 7 15 28 56 65 89)
Fichier B : (33 98)

Fusion dans le fichier X (on écrase le contenu précédent du fichier X) : on obtient 1 monotonies de longueur 10 : le fichier est complètement trié.
Fichier X : (2 5 7 15 28 33 56 65 89 98)

g) Combien de comparaisons d'éléments effectue-t-on au pire lorsqu'on fusionne une monotonie de longueur L_a avec une autre monotonie de longueur L_b ?

Le cas le pire est celui de 2 monotonies dont les éléments sont « entremêlés », c'est-à-dire le cas où on n'épuise pas une monotonie « prématurément ».
Exemple 1 : fusion de {1, 6} avec {8, 23, 51}, 2 comparaisons, puis on sait qu'on peut recopier directement la deuxième monotonie sans faire davantage de comparaisons.
Exemple 2 : fusion de {1, 8} avec {2, 6, 23} : on doit faire 4 comparaisons.
On a le cas le pire quand le dernier élément de A ne peut être « éliminé » (des éléments à traiter) que par une comparaison avec le dernier élément de B, et réciproquement. Plus formellement, les éléments 1..(L_b-1) de B sont inférieurs au dernier élément de A, et réciproquement, les éléments 1..(L_a-1) sont inférieurs au dernier élément de B.
Lors de la fusion des deux monotonies, on écrit $L_a + L_b$ éléments dans le fichier X. Dans le cas le pire, on n'écrit jamais plus d'un élément à la fois. On a donc $L_a + L_b$ étapes d'écriture. Chaque écriture d'élément est précédée d'une comparaison, sauf pour le dernier élément. **On a donc au pire $L_a + L_b - 1$ comparaisons à effectuer**.

h) Combien de comparaisons d'éléments effectue-t-on au pire lorsqu'on trie par fusion un fichier de *n* éléments, dans le cas particulier où *n* est une puissance de 2 ($n=2^p$) ?

L'intérêt de choisir n puissance de 2 est qu'on aura à chaque étape un nombre pair de monotonies à fusionner, et que les monotonies d'une étape donnée auront toutes la même longueur, même la dernière. Dans le cas plus général où n n'est pas une puissance de 2 (cf question a pour un exemple), FicA peut contenir une monotonie de plus (éventuellement incomplète) que FicB. Et si au contraire FicA et FicB contiennent le même nombre de monotonies, alors la dernière monotonie de FicB peut être incomplète.

Première passe : On a n monotonies de longueur 1 à fusionner 2 à 2, soit n/2 opérations de fusion. On a donc (n/2)*(1+1-1) = n/2 comparaisons à effectuer.

Deuxième passe : On a n/2 monotonies de longueur 2, à fusionner 2 à 2, soit n/4 opérations de fusion. On va donc effectuer (n/4)*(2+2-1) = 3n/4 comparaisons.

Troisième passe : On a n/4 monotonies de longueur 4 à fusionner 2 à 2, soit (n/8)*(4+4-1)=7n/8 comparaisons à réaliser.

k-ieme passe : A chaque passe, on multiplie par deux la longueur des monotonies et on divise par deux le nombre de monotonies. Avant la k-ieme passe, on a $n/2^{k-1}$ monotonies de longueur 2^{k-1} à fusionner 2 à 2, soit $n/2^k$ opérations de fusion à effectuer. On a donc $(n/2^k)*(2^{k-1} + 2^{k-1} -1) = (n/2^k)*(2^k - 1)$ comparaisons à réaliser. A l'issue de la k-ieme passe, on a $n/2^k$ monotonies de longueur 2^k.

On s'arrête quand on n'a plus qu'une monotonie de longueur n : $2^{k_{final}} = n = 2^p$, soit $k_{final} = p$.
On doit donc effectuer p passes sur le fichier.

Le nombre total de comparaisons est donc :

$$C = \sum_{k=1}^{p} \frac{n(2^k-1)}{2^k} = n\sum_{k=1}^{p} \frac{(2^k-1)}{2^k} = n\left(\sum_{k=1}^{p} 1 - \sum_{k=1}^{p} \frac{1}{2^k}\right) = n\left(p - \sum_{k=1}^{p} \frac{1}{2^k}\right)$$

On reconnaît la somme partielle d'une suite géométrique de raison q=1/2. Soit S cette somme.

S = $1/2 + 1/4 + 1/8 + \ldots + 1/2^p$
qS = $1/4 + 1/8 + \ldots + 1/2^p + 1/2^{p+1}$

S – qS = $1/2 - 1/2^{p+1}$ (les autres termes s'éliminent)
S = $(1/2 - 1/2^{p+1}) / (1 - q)$
S = $1 - 1/2^p$ (sachant que q=1/2)

Donc C = $n\left(p - 1 + \frac{1}{2^p}\right) = n\left(\log_2(n) - 1 + \frac{1}{n}\right)$

Au final, on obtient donc $C = n\log_2(n) - n + 1 = O(n\log_2(n))$ comparaisons pour trier par fusion un fichier de n éléments, dans le cas particulier où n est une puissance de 2.

Remarque : dans le cas général, c'est-à-dire pour n quelconque, il est possible de montrer que l'ordre de grandeur est inchangé, l'algorithme est toujours $O(n\log_2(n))$. Pour en savoir plus, voir le chapitre 4 de Cormen et al., Introduction à l'algorithmique, 2e édition. L'ouvrage est disponible à la BU.

i) Ecrire en langage algorithmique la procédure de tri par fusion d'un fichier, en supposant que vous disposez déjà des procédures « eclatement » et « fusion » (voir les entêtes ci-dessous). La procédure de tri doit appeler les procédures « eclatement » et « fusion » jusqu'à ce que le fichier soit complètement trié.

Procédure eclatement(nomFicX : chaîne de caractères, lg : entier, nomFicA : chaîne de caractères, nomFicB : chaîne de caractères)
Précondition : le fichier appelé nomFicX contient des monotonies de longueur lg, sauf peut-être la dernière qui peut être plus courte
Postcondition : Les monotonies de nomFicX sont réparties (1 sur 2) dans des fichiers appelés nomFicA et nomFicB : : la première monotonie est copiée dans le fichier A, la seconde dans le fichier B, la troisième dans le A, etc. Si ces fichiers existaient déjà, leur contenu est écrasé. Le fichier A peut recueillir une monotonie de plus le fichier B, et cette dernière monotonie peut être de longueur inférieure à lg. Si au contraire ficA et ficB recueillent le même nombre de monotonies, la dernière monotonie écrite dans ficB peut être de longueur inférieure à lg.
Paramètres en mode donnée : nomFicX, nomFicA, nomFicB, lg
{Les *chaînes de caractères* contenant les noms des fichiers ne vont pas être affectées par la procédure, d'où le passage en mode donnée, mais bien sûr, les *fichiers* désignés par nomFicA et nomFicB vont être affectés, comme cela est précisé dans les post-conditions.}

Procédure fusion(nomFicA : chaîne de caractères, nomFicB : chaîne de caractères, lg : entier, nomFicX : chaîne de caractères, nbMonoDansX : entier
Préconditions **:** les fichiers appelés nomFicA et nomFicB contiennent des monotonies de longueur lg. FicA peut contenir une monotonie de plus (éventuellement incomplète) que FicB. Si au contraire FicA et FicB contiennent le même nombre de monotonies, alors la dernière monotonie de FicB peut être incomplète.
Postconditions : le fichier appelé nomFicX contient nbMonoApres monotonies de longueur 2*lg, la dernière pouvant être plus courte. Ces monotonies résultent de la fusion 2 à 2 des monotonies de FicA et de FicB. Si FicX existait déjà, son ancien contenu est écrasé.
Paramètres en mode donnée : nomFicX, nomFicA, nomFicB, lg
Paramètre en mode résultat : nbMonoDansX

Procédure tri_par_fusion(nomFic : chaîne de caractères)
Précondition : le fichier appelé nomFic contient des Elements
Postcondition : le fichier appelé nomFic contient les mêmes éléments, mais triés.
Paramètres en mode donnée : nomFic
Variables locales :
    longueur, nbMonotonies : entier
    nomA, nomB : chaînes de caractères
Début
    longueur ← 1
    nomA ← « fichierAnnexeA.txt »
    nomB ← « fichierAnnexeB.txt »
    Répéter
        eclatement(nomFic, longueur, nomA, nomB)
        fusion(nomA, nomB, longueur, nomFic, nbMonotonies)
        longueur ← longueur * 2
    Jusqu'à ce que (nbMonotonies = 1)
Fin tri_par_fusion

j) Donnez le code de la procédure d'éclatement en langage C.

```
void eclatement(const char * nomFicX, int lg, \
                const char * nomFicA, const char * nomFicB)
{
  FILE * ficX, *ficA, *ficB ;
  int mettreDansFicA = 1;
  int i = 0;
  double elem_lu;

  ficX = fopen(nomFicX, "rb");
  if (ficX == NULL)
    {
      printf("Impossible de lire le fichier %s\n", nomFicX);
      printf("Il s'agit peut-etre d'un probleme de droits d'acces, \n");
      printf("ou peut-etre que le fichier n'existe pas. \n");
      exit(EXIT_FAILURE);
    }

  ficA = fopen(nomFicA, "wb");
  if (ficA == NULL)
    {
      printf("Impossible de creer le fichier %s en ecriture\n", nomFicA);
      printf("Il s'agit peut-etre d'un probleme de droits d'acces.\n");
      exit(EXIT_FAILURE);
    }

  ficB = fopen(nomFicB, "wb");
  if (ficB == NULL)
    {
      printf("Impossible de creer le fichier %s en ecriture\n", nomFicB);
      printf("Il s'agit peut-etre d'un probleme de droits d'acces.\n");
      exit(EXIT_FAILURE);
    }

  /* On boucle en fonction de la valeur de retour de fread et non en fonction
```

```
     de feof, cf cours magistral */
  while(fread(&elem_lu, sizeof(double), 1, ficX) == 1)
    {
      if (mettreDansFicA == 1)  fwrite(&elem_lu, sizeof(double), 1, ficA);
      else fwrite(&elem_lu, sizeof(double), 1, ficB);

      i++;
      if (i == lg)
              {
                if (mettreDansFicA == 1) mettreDansFicA = 0;
                else mettreDansFicA = 1;
                i = 0;
              }
    }

  fclose(ficX);
  fclose(ficA);
  fclose(ficB);
}
```

Exercice 3 *** : Compression RLE (CC mi-parcours avril 2011)

Soit un fichier binaire de *n* octets pouvant contenir des séquences répétitives d'un même octet. Par exemple, en décodant chaque octet comme un « unsigned char » :

{12, 6, 6, 6, 6, 6, 1, 3, 8, 8, 10, 2, 53, 53, 53, 53, 53, 6, 6, 6, 13}

Une première technique de compression consiste à écrire pour chaque octet le nombre de répétitions supplémentaires. On pourrait donc remplacer {53, 53, 53, 53, 53} par {53, 4}, ce qui signifie : l'octet 53, suivi d'encore 4 fois l'octet 53. Le problème de cette première idée est que la séquence {53, 53, 53, 53, 53} va bien être compressée, mais qu'au contraire, toutes les séquences non répétitives vont être dilatées : par exemple, {10} serait réécrit en {10, 0}.

Une meilleure idée consiste à ne transformer une séquence que si un octet est immédiatement répété au moins une fois. Ainsi, {53, 53, 53, 53, 53} serait réécrite en {53, 53, 3}, {10} serait inchangée et {8, 8} serait réécrite en {8, 8, 0}. La séquence entière donnée en exemple serait donc réécrite de la façon suivante : {12, 6, 6, **3**, 1, 3, 8, 8, **0**, 10, 2, 53, 53, **3**, 6, 6, **1**, 13}. Lors de la décompression, c'est le fait d'avoir deux octets identiques à la suite qui indique que l'octet suivant est un nombre d'occurrences supplémentaires et non un octet du fichier de départ. **C'est à cette seconde idée que vous allez vous intéresser dans tout cet exercice**. Il s'agit de la compression dite « RLE » (Run Length Encoding). Notez que bien que ce second algorithme soit la plupart du temps meilleur que le premier, il n'est pas parfait pour autant : il existe tout de même des cas pour lesquels il dilate au lieu de compresser.

k) Supposons que le fichier d'entrée contient 10 octets. Donnez un exemple de contenu de fichier qui donnerait la plus petite taille possible pour le fichier de sortie, puis un exemple de contenu qui donnerait la plus grande taille possible en sortie.

Exemple de contenu donnant la plus petite taille possible en sortie :
1, 1, 1, 1, 1, 1, 1, 1, 1, 1 → 1, 1, 8

Exemple de contenu donnant la plus grande taille possible en sortie :
1, 1, 2, 2, 3, 3, 4, 4, 5, 5 → 1, 1, 0, 2, 2, 0, 3, 3, 0, 4, 4, 0, 5, 5, 0

Considérons la procédure C suivante, qui compresse un fichier binaire selon l'algorithme RLE.

```
void compresser(const char * nomFichierEntree, const char * nomFichierSortie)
{
  FILE *entree, *sortie;
  unsigned char octet_courant, octet_prec, nb_occ = 1, nb_occ_suppl = 0;

  entree = fopen(nomFichierEntree, "rb"); if (entree == NULL) {exit(EXIT_FAILURE);}
  sortie = fopen(nomFichierSortie, "wb"); if (sortie == NULL) {exit(EXIT_FAILURE);}

  if( fread(&octet_prec, 1, 1, entree) != 1 )
    {
```

```
      /* Fichier d'entree vide, on laisse le fichier de sortie vide */
      fclose(entree);
      fclose(sortie);
      return;
    }
  fwrite(&octet_prec, 1, 1, sortie); /* on recopie le 1er octet */

  while (fread(&octet_courant, 1, 1, entree) == 1)
    {
      if (octet_courant != octet_prec)
        {
          if (nb_occ > 1)
            {
              nb_occ_suppl = nb_occ - 2;
              fwrite(&octet_prec, 1, 1, sortie);
              fwrite(&nb_occ_suppl, 1, 1, sortie);
              nb_occ = 1;
            }
           fwrite(&octet_courant, 1, 1, sortie);
           octet_prec = octet_courant;
        }
      else
        {
          nb_occ ++;
          nb_occ_suppl = nb_occ - 2;
          if (nb_occ == 255)
            {
              fwrite(&octet_prec, 1, 1, sortie);
              fwrite(&nb_occ_suppl, 1, 1, sortie);
              nb_occ = 1;
              octet_prec += 5; /* modification arbitraire de octet_prec,
                                  pour etre dans le bon cas a l'iteration suivante */
            }
        }
    }

  if (nb_occ > 1) /* si le fichier se termine par une séquence répétitive */
    {
      nb_occ_suppl = nb_occ - 2;
      fwrite(&octet_prec, 1, 1, sortie);
      fwrite(&nb_occ_suppl, 1, 1, sortie);
    }
  fclose(entree);
  fclose(sortie);
}
```

l) Considérons un fichier d'entrée contenant *n* octets avec *n* pair et non nul, et dont le contenu correspondrait au cas le pire en termes de taux de compression (voir la question a). Combien fait-on d'opérations de chaque type lors de l'exécution de la procédure compresser ? Pour simplifier, vous supposerez que la valeur de retour de fread est de type unsigned char (elle est en réalité de type size_t, un type dont la taille dépend de la machine). Vous supposerez aussi qu'un appel à fread ne compte pas en plus pour une affectation d'unsigned char.

- Nombre d'affectations d'unsigned char :
- Nombre d'additions d'unsigned char :
- Nombre de soustractions d'unsigned char
- Nombre de comparaisons d'unsigned char :
- Nombre d'affectations de pointeurs :
- Nombre de comparaisons de pointeurs :
- Nombre d'appels à fopen :
- Nombre d'appels à fclose :
- Nombre d'appels à fread :
- Nombre d'appels à fwrite :

Remarques :
- en ligne 18, il y aura n appels à fread : n-1 appels conduiront à un passage dans le while, puis le dernier provoquera la sortie du while (tentative de lecture d'un octet inexistant)
- le « if » de la ligne 20 sera vrai dans n/2 - 1 cas, faux dans n/2 cas
- le « if » de la ligne 22 sera toujours vrai : à chaque fois que l'on tombe sur un octet différent, nbocc vaut 2
- le « if » de la ligne 36 est toujours faux, car nbocc ne dépasse pas 2

Lignes	Aff. char	Add char	Soustr char	Comp char	Aff. Pointeur	Comp pointeurs	fread	fwrite
4	2							
6-7					2	2		
9				1			1	
16								1
18				n			n	
20				n - 1				
22				n/2 - 1				
24-27	2(n/2 – 1)		n/2 - 1					2(n/2 – 1)
29-30	n/2 - 1							n/2 – 1
34	n/2	n/2						
35	n/2		n/2					
36				n/2				
38-42								
47				1				
49-51	1		1					2
TOTAL	n +	n/2	n	3n	2	2	n + 1	3n/2

	3n/2							

m) A quoi servent les lignes 36 à 43 ?

Comme nb_occ et nb_occ_suppl sont de type unsigned char, ils ne peuvent pas prendre des valeurs supérieures à 255. Sans ce bloc de lignes, si jamais un octet était répété plus de 255 fois de suite, on tenterait d'incrémenter la variable alors qu'elle vaudrait 255. Au lieu de valoir 256 comme souhaité, elle vaudrait alors 0. On aurait donc une perte d'information dans le fichier compressé.

n) Considérons le « main » suivant. Il comporte une faille de sécurité. Indiquez où, expliquez de quelle faille il s'agit (quel est le risque) et comment la corriger.

```
#include <stdio.h>
#include <stdlib.h>

void compresser(const char * nomFichierEntree, const char * nomFichierSortie);

int main()
{
  char nomGrosFichier[50];

  printf("Tapez le nom du fichier à compresser (45 caracteres max, pas d'espace) :\n");
  scanf("%s", nomGrosFichier);
  compresser(nomGrosFichier, "compression.bin");

  return 0;
}
```

Instruction incorrecte :
scanf("%s", nomGrosFichier);

Type de faille de sécurité, nature du risque :
Il s'agit d'une faille de sécurité de type « buffer overflow ». Si l'utilisateur saisit un nom de fichier plus long que 49 caractères, les caractères surnuméraires écraseront les données de la pile situées au-dessus du tableau. Avec une chaîne bien choisie, un utilisateur malveillant pourrait modifier le pointeur d'instruction et faire exécuter un code de son choix.

Comment corriger le problème :
scanf("%49s", nomGrosFichier);

o) Donnez le code C de la procédure de décompression, dont l'entête est la suivante :

void decompresser(const char * nomFichierEntree, const char * nomFichierSortie);
/* Préconditions : nomFichierEntree est le nom d'un fichier binaire compressé selon l'algorithme RLE.
Postconditions : un nouveau fichier nommé comme spécifié dans la chaîne nomFichierSortie est créé, son contenu correspond à la décompression du fichier d'entrée. */

```
void decompresser(const char * nomFichierEntree, const char * nomFichierSortie)
{
  FILE *entree, *sortie;
  unsigned char octet_courant, octet_prec;
  unsigned char nb_occ_suppl = 0;
  char onVientDeFinaliserUneRepet = 0 ;

  entree = fopen(nomFichierEntree, "rb");
  if (entree == NULL)
    {
```

```
      fprintf(stderr, "Erreur, impossible d'ouvrir le fichier %s.\n",
              nomFichierEntree);
      exit(EXIT_FAILURE);
    }
  sortie = fopen(nomFichierSortie, "wb");
  if (sortie == NULL)
    {
      fprintf(stderr, "Erreur, impossible d'ouvrir le fichier %s.\n",
              nomFichierSortie);
      exit(EXIT_FAILURE);
    }

  if( fread(&octet_prec, 1, 1, entree) != 1 )
    {
      /* Fichier d'entree vide, on laisse le fichier de sortie vide */
      fclose(entree);
      fclose(sortie);
      return;
    }

  fwrite(&octet_prec, 1, 1, sortie);
  while (fread(&octet_courant, 1, 1, entree) == 1)
    {
      if ((octet_courant == octet_prec) && (onVientDeFinaliserUneRepet != 1))
        {
          /* on écrit une seconde fois l'octet répété */
          fwrite(&octet_courant, 1, 1, sortie);

          if (fread(&nb_occ_suppl, 1, 1, entree) == 1)
            {
              /* attention a la sortie de la boucle :
                 nb_occ_suppl ne peut pas valoir -1, c'est un unsigned char */
              while (nb_occ_suppl > 0)
                {
                  fwrite(&octet_courant, 1, 1, sortie);
                  nb_occ_suppl --;
                }
              onVientDeFinaliserUneRepet = 1 ;
              /* penser aux cas comme : a, a, 253, a, a, 7, ... */
            }
          else
            {
              fprintf(stderr, "Erreur, format de fichier compresse incorrect.\n");
              exit(EXIT_FAILURE);
            }
        }
      else
        {
          onVientDeFinaliserUneRepet  =0 ;
          fwrite(&octet_courant, 1, 1, sortie);
          octet_prec = octet_courant;
        }
    }

  fclose(entree);
  fclose(sortie);
}
```