Algorithmique

Correction de l'examen de première session 2015/2016

Durée : 2 heures. Aucun document autorisé.

Exercice 1. (3 points) **/3**

On dispose d'une image de 1024 × 512 pixels en 256 niveaux de gris.

1. Aucune technique de compression particulière n'est appliquée pour stocker cette image. Quelle est la taille, mesurée en méga-octets (Mo), de l'espace mémoire nécessaire à son stockage ? **/2**

```
L'image contient 2^10 x 2^9= 2^19 pixels.
Chaque pixel est une valeur entre 0 et 255 qui peut être codée sur 8 bits, soit
1 octet.
Le stockage de l'image nécessite donc 2^19 octets, soit 1/2 Mo.
```

2. Préciser une structure de donnée adaptée à la manipulation de cette image. **/1**

```
L'image sera représentée avec un tableau 2D de 1024 lignes et 512 colonnes
d'entiers non signés codés sur 1 octet.
```

Exercice 2. (25 points) **/25**

La procédure `LireImage` initialise un tableau 2D d'entiers à partir d'une variable fichier `img` qui contient une image à niveaux de gris. Son en-tête est :

```
procedure LireImage(img: in fichier, l: in entier, c: : in entier, t: out tableau[l,c] d'entiers)
```

Il existe aussi une procédure `EcrireImage` qui effectue la transformation inverse de `LireImage`. Tout traitement d'image doit être réalisé sur le tableau 2D d'entiers obtenu par `LireImage`, comme `t` dans l'exemple suivant.

```
declare
   carre : fichier = ''la_photo_d_un_carré_gris''
   carreModifie : fichier = ''la_photo_d_un_carré_transformé''
   t : tableau[512, 512] d'entiers
debut
   LireImage(carre, 512, 512, t) // t contient les valeurs des pixels de carre
   // traitement à réaliser sur t
   EcrireImage(carreModifie, 512, 512, t)
fin
```

1. On dispose d'une image de 1024 $\times$ 512 pixels en 256 niveaux de gris. Modifier l'algorithme partiel précédent pour préparer le traitement de cette image. **/1**

```
declare
   img : fichier = ``l_image_de_l_exo''
   t : tableau[1024, 512] d'entiers
debut
   LireImage(img, 1024, 512, t) // t contient les valeurs des pixels de img
   // traitement à venir sur t
   EcrireImage(img, 1024, 512, t)
fin
```

Les questions suivantes utiliseront si besoin cet algorithme partiel sans le ré-écrire en entier à chaque fois.

2. Ecrire le traitement qui calcule et affiche le pourcentage de pixels blancs dans l'image. **/4**

```
declare
   img : fichier = ``l_image_de_l_exo''
   t : tableau[1024, 512] d'entiers
   i, j : entiers
   nbB : entier = 0
   ratioB : entier
debut
   LireImage(img, 1024, 512, t)

   // compter les blancs
   pour i de 0 à 1023 faire
     pour j de 0 à 512 faire
       si t[i,j] == 255 alors
         nbB = nbB + 1
       finsi
     finpour
   finpour

   // calculer pourcentage de blancs
   ratioB = nbB * 100 / (1024*512)     // division entre 2 entiers qui -> flottant

   // affichage
   afficher(ratioB)

fin
```

3. Ecrire l'en-tête d'une fonction `nbOcc` qui calcule le nombre d'occurrences d'une valeur (donnée en entrée) présente dans un tableau 2D d'entiers (donné en entrée). **/2**

```
fonction nbOcc(val : entier, l: entier, c: entier, t: tableau[l,c] d'entiers)
  retourne entier
```

4. Ecrire à nouveau le traitement de la question 2 à l'aide de la fonction `nbOcc`. **/2**

```
declare
   img : fichier = ``l_image_de_l_exo''
   t : tableau[1024, 512] d'entiers
   nbB : entier
   ratioB : entier

   avec fonction nbOcc(val : entier, l: entier, c: entier,
                       t: tableau[l,c] d'entiers) retourne entier

debut
   LireImage(img, 1024, 512, t)

   nbB = nbOcc(255, 1024, 512, t)
   ratioB = nbB * 100 / (1024*512)

   afficher(ratioB)
fin
```

5. Ecrire le corps de la fonction `nbOcc`. **/3**

```
fonction nbOcc(val : entier, l: entier, c: entier, t: tableau[l,c] d'entiers)
  retourne entier

   i, j : entiers
   nb : entier = 0

debut
   pour i de 0 à l-1 faire
     pour j de 0 à c-1 faire
       si t[i,j] == val alors
         nb = nb + 1
       finsi
     finpour
   finpour

  retourne nb
fin fonction
```

6. (⋆) Proposer une mesure et un paramètre de la complexité en temps de la fonction `nbOcc`. **/2**

```
La fonction nbOcc compte le nombre d'occurrences d'une valeur donnée dans un
tableau 2D. Sa complexité peut être mesurée comme une fonction du nombre de
comparaisons à cette valeur.
Ce nombre dépend de la taille l x c du tableau 2D, cad. du
nombre n de pixels total de l'image d'origine.
```

7. (⋆) Quelle est la complexité asymptotique en temps de la fonction `nbOcc`. **/2**

```
Le nombre de comparaisons est égal au nombre l x c = n.
La complexité de nbOcc est donc linéaire en n.
```

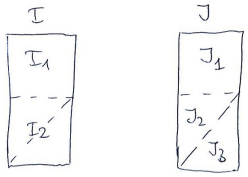
8. Ecrire l'en-tête puis le corps de la fonction `rev256` qui calcule et retourne le négatif en niveaux de gris d'une valeur donnée de pixel.

   On a par exemple les transformations suivantes : $0 \longleftrightarrow 255, 63 \longleftrightarrow 192, 128 \longleftrightarrow 127$. **/2**

```
fonction rev256(v: entier) retourne entier
debut
  retourne 255 - v
fin fonction
```

9. (⋆) Ecrire l'algorithme qui réalise le traitement suivant en veillant à minimiser l'espace mémoire supplémentaire nécessaire. A partir de l'image I (1024 x 512) d'origine découpée en 2 sous-images carrées I_1, I_2, on créé une image J composée des 3 sous-image J_1, J_2, J_3 telles que :

   — $J_1 = I_1$
   — J_2 est la partie triangulaire supérieure par rapport à la seconde diagonale de I_2
   — J_3 est le symétrique du négatif en niveaux de gris de J_2



/4

```
declare
   I : fichier = ``l_image_I''
   J : fichier = ``l_image_J''
   t : tableau[1024, 512] d'entiers
   i, j, k, ii : entiers
debut
   LireImage(I, 1024, 512, t)

   // J1, J2: rien à faire
   // J3
   pour ii de 0 à 511 faire    // numéro de lignes de la partie basse
     i = 512+ii                // numéro des lignes originales de t
     pour j de 0 à ii faire // partie sup de la partie basse
       t[1024-j, 512-i] = rev256(t[i,j])
     finpour
   finpour

   EcrireImage(J, 1024, 512, t)

fin
```

10. (⋆) Quel est l'espace mémoire supplémentaire nécessaire ? Justifier votre réponse. **/3**

```
Aucun espace mémoire supplémentaire est nécessaire : le traitement est effectué
en place.
. J1 est I1 sans modification
. J2 est la partie triangulaire supérieure de I2, diagonale comprise, sans la modifier
. J3 est obtenu colonne après colonne (en partant de la droite vers la gauche,
et pour la partie inférieure correspondante) à partir d'un traitement des lignes
(de haut en bas, et la partie gauche correspondante)
de J2. J3 peut donc écraser la partie inférieure de I2.
```

Exercice 3. (22 points) **/22**

1. Donner un exemple (différent de celui de la question suivante) de problème résolu par un algorithme itératif ou un algorithme récursif. Expliciter ces 2 algorithmes. **/3**

```
Voir cours : factorielle, affichage boucle, exponentiation ...
```

2. Quel peut-être l'avantage d'une solution récursive et comment en profiter ? Justifier votre réponse avec un exemple **/3**

```
Voir cours : réduire la complexité grâce à la stratégie diviser pour régner
Exemple : exponentiation rapide, tris récursifs
```

3. Pourquoi la question précédente indique "peut-être" ? Justifier votre réponse avec un exemple. **/3**

```
Voir cours : Fibonacci explose en récursif
```

4. Soit $s2(n)$ la somme des n premiers entiers **pairs** strictement positifs.
   Ecrire le corps d'une fonction `s2_It` qui calcule $s2(n)$ avec un algorithme itératif. **/2**

```
fonction s2_It(n: entier) retourne entier
  s : entier = 0
  i : entier
début
  pour i de 1 à n faire
    s = s + 2*i
  finpour
  retourne s
fin fonction
```

5. Ecrire le corps d'une fonction `s2_Rec` qui calcule $s2(n)$ avec un algorithme récursif. **/2**

```
fonction s2_Rec(n: entier) retourne entier
  s : entier = 0
  i : entier
  avec fonction s2_Rec(n: entier) retourne entier

début
  si n==0 alors
    retourne 0
  sinon
    retourne 2*n + s2_Rec(n-1)
  finsi
fin fonction
```

6. (⋆) Expliciter l'arbre des appels et les environnements successifs de l'évaluation `s2_Rec(3)`. **/3**

```
s(3):   s(2):   s(1):   s(0):
6+s(2)
        4+s(1)
                2+s(0)
                        <-0
                <-2+0
        <-4+2
<-6+6
```

7. Combien d'appels récursifs ont été nécessaires ? **/1**

```
3 + 1 appels
```

8. (⋆) Comment mesurer la complexité en temps de `s2_Rec` ? **/2**

```
On compte le nombre d'additions qui est égal au nombre d'appels (à 1 près).
```

9. (⋆) Quelle est la complexité (en temps) asymptotique de `s2_Rec` ? Qu'en penser ? **/3**

```
La complexité asymptotique (en temps) est linéaire, comme celle la version itérative.
Aucune stratégie diviser pour régner n'a pu être appliquée ici pour que la
version récursive améliore la complexité de la version itérative.
(Non demandé: la complexité en espace de la version récursive est moins
intéressante car il faut conserver la pile des appels, soit n variables de
retour et n valeurs à sommer).
```