Algorithmique avancée et programmation C

Durée : *1h30*
Documents autorisés : **AUCUN**

Remarques :
— Vos réponses aux exercices 2 et 3 seront sur des copies doubles différentes
— Veuillez lire attentivement les questions avant de répondre ;
— Rendez une copie propre ;
— N'utilisez pas de crayon à papier sur votre copie ;
— Le temps indiqué est le temps approximatif que vous devriez réserver pour répondre aux questions de l'exercice.

1 QCM (10 min)

Répondez au qcm ci joint (à rendre avec votre copie). La note de chaque question peut varier de -1 à $+1$.

Soit B le nombre de bonnes réponses à une question et soit M le nombre de mauvaises réponses à cette même question ($B + M$ est égal au nombre de réponses à la question). Chaque bonne réponse cochée rapporte $1/B$ points et chaque mauvaise réponse $-1/M$ points.

Solution proposée :

Compétences évaluées

— DEV001 : Savoir compiler et linker un programme C (options de base de gcc)
— DEV008 : Savoir traduire des passages de paramètre algorithme en passage de paramètre C
— DEV009 : Maîtriser les pointeurs, tableaux et chaînes de caractères

2 Approximation de la fonction ln (30 min)

On suppose posséder la fonction suivante qui calcule l'exponentiel d'un nombre x :
— **fonction** exp **(**x : **Reel) : ReelPositif**

En s'inspirant de la fonction vue en TD qui calcule l'approximation de la fonction racine carrée à epsilon près, proposez une fonction récursive qui calcule dichotomiquement l'approximation de la fonction logarithme népérien d'une valeur x à epsilon près (conseil : distinguez les cas où $x \geq 1$ et $x < 1$ et pour rappel $ln(1/x) = -ln(x)$).

Quelle est la complexité $T(n)$ dans le pire des cas ? Justifiez.

Solution proposée :

Compétences évaluées

— CP004 : Savoir concevoir une signature (préconditions incluses)
— CD004 : Savoir écrire des algos avec le pseudo code utilisé à l'INSA
— CD005 : Savoir écrire un pseudo code lisible (indentation, identifiant significatif)
— CD009 : Savoir écrire un algorithme qui résout le problème
— CD011 : Savoir utiliser les bons types de données (paramètres formels, variables locales)
— CD201 : Savoir identifier et résoudre le problème des cas non récursifs
— CD202 : Savoir identifier et résoudre le problème des cas récursifs
— CD302 : Savoir définir l'espace de recherche d'un algorithmique dichotomique
— CD303 : Savoir diviser et extraire les bornes de l'espace de recherche d'un algorithme dichotomique (cas discret ou continu)
— CD101 : Savoir estimer la taille d'un problème (n)
— CD102 : Savoir calculer une complexité dans le pire et le meilleur des cas

fonction ln **(**x : **ReelPositif**, epsilon : **ReelPositif) : Reel**

⌊**précondition(s)** x>0

```
debut
   si x≥1 alors
      retourner lnR(x,0,x,epsilon)
   sinon
      retourner -lnR(1/x,0,1/x,epsilon)
   finsi
fin
fonction lnR (x,ln_g,ln_d,epsilon :ReelPositif) : ReelPositif
   ⌊précondition(s)  x≤1
   Déclaration  ln_m : ReelPositif
debut
   si ln_d-ln_g > epsilon alors
      ln_m ← (ln_g+ln_d)/2
      si exp(ln_m) < x alors
         ln_g ← ln_m
      sinon
         ln_d ← ln_m
      finsi
      retourner lnR(x,ln_g,ln_d,epsilon)
   finsi
   retourner ln_g
fin
```

Soit $nb = arrondi(x/epsilon) + 1$ et soit n le nombre de bits pour représenter nb : nb vaut au maximum 2^n.

nb représente la taille de l'espace de recherche de notre algorithmique récursif dichotomique. Le nombre d'appels récursifs est en $log_2(nb)$ car à chaque appel récursif on divisie l'espace de recherche par 2. A chaque appel récursif il y a deux affectations, donc $T(n) = O(2 * log_2(nb)) = O(log_2(nb)) = O(log_2(2^n)) = O(n)$.

3 Traitements d'image (50 min)

3.1 Analyse : TAD `ImageNG`

Une image bitmap est un ensemble de pixels organisés en deux dimensions. Le pixel de coordonnée $(0, 0)$ se trouvant en haut à gauche. Les « teintes » des pixels d'une image en niveaux de gris sont représentées à l'aide d'un naturel dont les valeurs vont de 0 à 255.

Proposez le TAD `ImageNG` (axiomes inclus) sachant que l'on peut :
— créer une image en niveaux de gris à partir des dimensions et d'un niveau de gris donnés ;
— obtenir les dimensions d'une image ;
— obtenir ou modifier le niveau de gris d'un pixel d'une image.

Solution proposée :

Compétences évaluées

— AN002 : Savoir être précis quant aux types de données utilisés
— AN201 : Savoir identifier les dépendances d'un TAD
— AN204 : Savoir formaliser des opérations d'un TAD
— AN205 : Savoir formaliser les préconditions d'une opération d'un TAD
— AN206 : Savoir formaliser des axiomes ou savoir définir la sémantique d'une opération d'un TAD

Nom: ImageNG
Utilise: **Naturel**, **NaturelNonNul**, 0..255
Opérations: imageNG: **NaturelNonNul** × **NaturelNonNul** × 0..255 → ImageNG
dimensions: ImageNG → **NaturelNonNul** × **NaturelNonNul**
obtenirNG: ImageNG × **Naturel** × **Naturel** ↛ 0..255
fixerNG: ImageNG × **Naturel** × **Naturel** × 0..255 ↛ ImageNG
Préconditions: obtenirNG(imageNG(l,h,ng),x,y): $x < l$ et $y < h$
fixerNG(imageNG(l,h,ng),x,y,c): $x < l$ et $y < h$

Axiomes: - dimensions(imageNG(l,h,ng)=l,h
- obtenirNG(imageNG(l,h,ng),x,y)=ng
- obtenirNG(fixerNG(im,x,y,ng),x,y)=ng

3.2 Conception préliminaire

Donnez les signatures des fonctions et procédures des opérations précédentes.

Solution proposée :

Compétences évaluées

— CP003 : Savoir choisir entre une fonction et une procédure
— CP004 : Savoir concevoir une signature (préconditions incluses)
— CP005 : Savoir choisir un passage de paramètre (E, S, E/S)

— **fonction** imageNG **(**largeur,hauteur : **NaturelNonNul**, couleurFond : 0..255**) :** ImageNG
— **fonction** dimensions **(**im : ImageNG**) : NaturelNonNul**, **NaturelNonNul**
— **fonction** obtenirCouleur **(**im : ImageNG,x,y : **Naturel) :** 0..255
⌊**précondition(s)** l,h←dimensions(im), x<l, y<h
— **procédure** fixerCouleur **(E/S** im : ImageNG,**E** x,y : **Naturel**, couleur : 0..255**)**
⌊**précondition(s)** l,h←dimensions(im), x<l, y<h

3.3 Utilisation : filtre moyenneur

En traitement d'image, un filtre est une fonction qui calcule une image destination à partir d'une image source. Chaque pixel de l'image destination est fonction des couleurs des pixels de l'image source et optionnellement des couleurs des pixels de l'image destination déjà calculées.

Le principe du filtre moyenneur est que la couleur de chaque pixel x, y de l'image destination est égale à la moyenne des couleurs des pixels appartenant à une fenêtre carrée centrée en x, y de l'image source. La largeur de cette fenêtre carrée est obligatoirement impaire.

La figure 1 présente l'utilisation d'un filtre moyenneur avec une fenêtre de taille 5 sur une image en niveaux de gris (exemple issue de `http://perso.telecom-paristech.fr/~maitre/BETI//filtres_lin_nlin/filtres.html`).

(a) Image source (b) Image destination

FIGURE 1 – Utilisation d'un filtre moyenneur de taille 5

3.3.1 Analyse

Proposez une analyse descendante du problème pour les images en niveaux de gris.

Solution proposée :

Compétences évaluées

- AN001 : Savoir désigner les choses (identifiant significatif)
- AN101 : Savoir identifier les entrées et sorties d'un problème
- AN102 : Savoir décomposer logiquement un problème
- AN104 : Savoir si un problème doit être décomposé

— filtreMoyenneur : ImageNG × **NaturelNonNul** ⇸ ImageNB
  — moyenneFenetree : ImageNG × **Naturel** × **Naturel** × **NaturelNonNul** ⇸ 0..255
    — sommeFenetree : ImageNG × **Naturel** × **Naturel** × **NaturelNonNul** ⇸ **Naturel** × **NaturelNonNul**
      — min : **Entier** × **Entier** → **Entier**
      — max : **Entier** × **Entier** → **Entier**

3.3.2 Conception préliminaire

Donnez les signatures des fonctions et procédures issues de votre analyse.

Solution proposée :

Compétences évaluées

- CP004 : Savoir concevoir une signature (préconditions incluses)

fonction filtreMoyenneur **(**im : ImageNG, tailleFenetre : **NaturelNonNul) :** ImageNG

⌊**précondition(s)** impair(tailleFenetre)

fonction moyenneFenetree **(**im : imageNB, x,y : **Naturel**, tailleFenetre : **NaturelNonNul) :** 0..255

⌊**précondition(s)** l,h←dimensions(im), x<l, y<h et impair(tailleFenetre)

fonction sommeFenetree **(**im : imageNB, x,y : **Naturel**, tailleFenetre : **NaturelNonNul) : Naturel**, **NaturelNonNul**

⌊**précondition(s)** l,h←dimensions(im), x<l, y<h et impair(tailleFenetre)

3.3.3 Conception détaillée

Donnez les algorithmes des fonctions et/ou procédures permettant de résoudre ce problème.

Solution proposée :

Compétences évaluées

- CD002 : En tant qu'utilisateur, savoir respecter une signature
- CD004 : Savoir écrire des algos avec le pseudo code utilisé à l'INSA
- CD005 : Savoir écrire un pseudo code lisible (indentation, identifiant significatif)
- CD006 : Savoir choisir la bonne itération
- CD009 : Savoir écrire un algorithme qui résout le problème

fonction filtreMoyenneur **(**im : ImageNG, tailleFenetre : **NaturelNonNul) :** ImageNG

⌊**précondition(s)** impair(tailleFenetre)

Déclaration i,j,largeur,hauteur : **Naturel**
res : ImageNG

debut
im ← imageNB(largeur(im),hauteur(im),0)
largeur,hauteur ← dimensions(im)
pour i ←0 **à** largeur **faire**
  pour j ←0 **à** hauteur **faire**
    fixerCouleur(res,i,j,moyenneFenetree(im,x,y,tailleFenetre))
  finpour
finpour

```
    retourner res
fin
fonction moyenneFenetree (im : imageNB, x,y : Naturel, tailleFenetre : NaturelNonNul) : 0..255
    ⌊précondition(s)  l,h←dimensions(im), x<l, y<h et impair(tailleFenetre)
    Déclaration  somme :NaturelNonNul
                 nbPixels :Naturel
debut
    somme, nbPixels ← sommeFenetree(im,x,y,tailleFenetre)
    retourner somme div nbPixels
fin
fonction sommeFenetree (im : imageNB, x,y : Naturel, tailleFenetre : NaturelNonNul) : Naturel, Naturel-
RelNonNul
    ⌊précondition(s)  l,h←dimensions(im), x<l, y<h et impair(tailleFenetre)
    Déclaration  i,j,somme,largeur,hauteur,nbPixels : Naturel
debut
    somme ← 0
    nbPixels ← 0
    largeur,hauteur ← dimensions(im)
    pour i ←entierEnNaturel(max(0,x-taillefrenetre div 2)) à entierEnNaturel(min(largeur-1,x+taillefrenetre
    div 2)) faire
        pour j ←entierEnNaturel(max(0,y-taillefrenetre div 2)) à entierEnNaturel(min(hauteur-1,y+taillefrenetre
        div 2)) faire
            somme ← somme+obtenirCouleur(im,i,j)
            nbPixels ← nbPixels+1
        finpour
    finpour
    retourner somme, nbPixels
fin
```