Devoir en temps limité n°4 – 200min

Calculatrices interdites

On veillera à présenter très clairement sa copie : il faut rédiger les réponses et encadrer les résultats. Pour le code, il doit être indenté, on ne commence pas une fonction en bas de page et on utilise de la couleur pour les commentaires.

Le code doit être commenté dès qu'il dépasse les 5 lignes.

Les fonctions en C et en Ocaml doivent avoir le type précisé. Il est donc recommandé d'utiliser des fonctions auxiliaires en Ocaml.

1 Graphe de flot de contrôle

Voici une fonction écrite en C. Remarque : cette fonction a été écrite au hasard pour servir d'exemple, n'essayez pas de comprendre ce qu'elle renvoie.

```c
int f(int a, int b){
 int z = a+b;
 if (z%2 == 0){
   if (a==0){return 0;}
   else{return 2*z;}
 }
 else{
   while(z>a){
     z = z-2;
   }

   return z+1;
 }
}
```

1. Dessiner un graphe de flot de contrôle pour cette fonction.
2. Donner un jeu de tests qui couvre tous les arcs (toutes les flèches) du graphe de flot de contrôle précedemment donné.
   Remarque : un "jeu de tests" est composé de 1 ou plusieurs valeurs à tester pour les entrées de la fonction.

2 Tri fusion

3. Rappeler le principe du tri fusion en effectuant le tri de la liste $[3, 0, 5, 8, 1, 2]$.
4. Programmer le tri fusion en Ocaml, pour une liste d'entiers.
5. Rappeler la complexité du tri fusion.

3 Expressions simples

On définit par induction structurelle l'ensemble $\mathcal{T}$ des instructions simples par :

- $\mathcal{B} = \{Z, T\}$
- C contient deux constructeurs d'arité 2 : $c_1 : x, y \mapsto [\![x \oplus y]\!]$ et $c_2 : x, y \mapsto [\![x \otimes y]\!]$

On pourra voir les instructions simples comme des "mots" construits avec les symboles Z, T, $[\![$, $]\!]$, $\oplus$ et $\otimes$.
Par exemple, $t_1 = Z$, $t_2 = [\![Z \oplus T]\!]$ ou $t_3 = [\![[\![Z \otimes T]\!] \oplus Z]\!]$ sont des expressions simples.

6. Écrire t_2 et t_3 à l'aide des constructeurs et des éléments de la base.

On définit une fonction $\varphi : \mathcal{T} \to \mathbb{N}$ par induction de la manière suivante :

- $\varphi(Z) = 0$, $\varphi(T) = 2$
- $\varphi([\![x \otimes y]\!]) = \varphi(x) \times \varphi(y)$
- $\varphi([\![x \oplus y]\!]) = \varphi(x) + \varphi(y)$

7. Calculer $\varphi(t_2)$ et $\varphi(t_3)$. Donner une expression simple $t \in \mathcal{T}$ telle que $\varphi(t) = 6$.
8. Rappeler le principe de la preuve par induction d'une propriété P pour un ensemble $\mathcal{T}$ défini par induction par $(\mathcal{B}, C)$.
9. Montrer par induction structurelle que $\forall t \in \mathcal{T}$, $\varphi(t)$ est pair.

4 Parcours de grille

On considère une grille de taille $n \times m$ qui contient des entiers positifs. On veut aller du coin en haut à gauche au coin en bas à droite en suivant un chemin dont la somme des cases est la plus grande possible.

Dans ce chemin, on ne peut se déplacer que à droite et vers le bas, jamais à gauche, jamais en haut.

Voici un exemple de grille dans laquelle la somme maximale atteignable sur un chemin est de 28. Le chemin correspondant est mis en évidence sur la figure (0-b).

3	8	1	8
6	3	2	3
2	8	4	2

Figure (0-a)

3	8	1	8
6	3	2	3
2	8	4	2

Figure (0-b)

10. Sachant que la grille contient n lignes et m colonnes, donner la longueur N (en nombre de déplacements effectués) d'un chemin qui va du coin en haut à gauche au coin en bas à droite sans jamais se déplacer ni vers la gauche, ni vers le haut.
    Remarque : cette longueur est la même pour tous les chemins valides.

1. Approche exhaustive

Dans cette section, on veut résoudre le problème par une approche exhaustive, c'est à dire tester tous les chemins et déterminer celui qui a la plus grande somme.

Pour énumérer les chemins, on va les représenter par des tableaux de longueur N (déterminée en question 10) qui contiennent des 0 et des 1 : un 0 signifie qu'on va en bas, un 1 signifie qu'on va à droite.

Par exemple, le chemin de la figure (0-b) serait représenté par le tableau $[1, 0, 0, 1, 1]$.

Pour énumérer les chemins possibles, on considère que le premier chemin est celui qui n'a que des zéros, que le deuxième est de la forme $[0, ..., 0, 1]$, le troisième de la forme $[0, ..., 0, 1, 0]$, etc... et le dernier ne contient que des 1.

11. Compléter la fonction `suivant` ci-dessous qui permet de passer d'un chemin au suivant. La fonction modifie le chemin donné en entrée et renvoie un booléen : vrai si le chemin donné est le dernier, faux sinon.

```c
bool suivant(int* chemin, int grandN){
  int i = ...;
  int fini = false;

  while (...){
    if (chemin[i] == ...){chemin[i] = ...;
                          i-=1;}
    else {chemin[i] = ...;
          fini = true;}
  }

  if (...) {return true;}
  else {return false;}
}
```

12. Notre méthode pour représenter les chemins peut donner des chemins qui ne vont pas jusqu'au coin en bas à droite (par exemple le chemin avec que des 0).
    Écrire une fonction `bool verifie_chemin(int* chemin, int n, int m)` qui vérifie qu'en partant de la case $(0, 0)$ et en suivant les instructions du chemin, on arrive bien à la case $(n-1, m-1)$.
13. Écrire une fonction `int somme_chemin(int* chemin, int** grille, int n, int m)` qui renvoie la somme des cases rencontrées sur le chemin.
    Dans cette question, on suppose que le chemin est valide et arrive bien à la case $(n-1, m-1)$.
14. En utilisant les fonctions précédentes, compléter la fonction suivante qui donne la meilleure somme d'un chemin valide.

```c
int approche_exhaustive(int** grille, int n, int m){
  int grandN = ...;
  int* chemin = ...;
  ...; //Initialiser le premier chemin avec que des 0
```

```
  int maxi = ...;

  while(...){
    if (...){ //Vérifier si chemin valide
      int somme = ...;
      if (...){ //Recherche de la somme max
        maxi = ...;
      }
    }
  }

  return maxi;
}
```

15. Combien de chemins teste-t'on dans la fonction `approche exhaustive` ? En déduire un minorant de la complexité de la fonction.
16. Combien de chemins valides existe-t'il réellement (qui vont vraiment de $(0,0)$ à $(n-1,m-1)$) ? Tester uniquement ces chemins permettrait-il de réduire le nombre asymptotique de tours de la boucle while dans la fonction `approche exhaustive` ?

2. Approche gloutonne

Dans cette section, on veut résoudre le problème par une approche gloutonne, dans l'espoir d'obtenir une méthode moins coûteuse que l'exploration exhaustive.

17. Rappeler le principe d'un algorithme glouton.

Dans notre cas, quand on se trouve sur une certaine case qui n'est pas sur le bord droit ou le bord du bas, on a le choix d'aller à droite et d'ajouter une valeur v_d à somme de notre chemin ou d'aller en bas et d'ajouter une valeur v_b à notre somme.

Notre but est d'avoir la plus grande somme possible, donc il semble localement optimal d'aller à droite si $v_d > v_b$ et d'aller en bas sinon.

Si la case actuelle est sur le bord du bas, on est obligés d'aller à droite jusqu'à la fin.

Si la case actuelle est sur le bord de droite, on est obligés d'aller en bas jusqu'à la fin.

18. Écrire une fonction `int approche_gloutonne(int** grille, int n, int m)` qui calcule la solution selon la méthode gloutonne.
19. Quelle est la complexité de votre fonction ?
20. Montrer sur un exemple que la méthode gloutonne n'est pas optimale.

3. Approche par programmation dynamique

Dans cette section, on va appliquer une approche par programmation dynamique pour combiner une complexité polynomiale et une certitude d'obtenir la bonne réponse.

On définit PGS une fonction telle que $PGS(i,j)$ vaut la valeur de la plus grande somme d'un chemin qui va de la case $(0,0)$ à la case (i,j).

En reprenant la grille de la figure (0-a) on aurait :

$PGS(0,1) = 3+8 = 11$, $PGS(0,3) = 3+8+1+8 = 20$, $PGS(1,1) = 3+8+3 = 14$ ou encore $PGS(2,2) = 3+8+3+8+4 = 26$.

PGS peut être calculée par récurrence. En effet pour arriver en (i,j) (une case qui n'est ni sur la première ligne, ni sur la première colonne), on vient soit de $(i-1,j)$ par un déplacement vers le bas, soit de $(i,j-1)$ par un déplacement vers la droite. On choisit la possibilité qui a la plus grande somme et on ajoute la valeur de la case (i,j).

Voici la formule de récurrence pour PGS. Il y a un cas de base et trois cas récursifs.

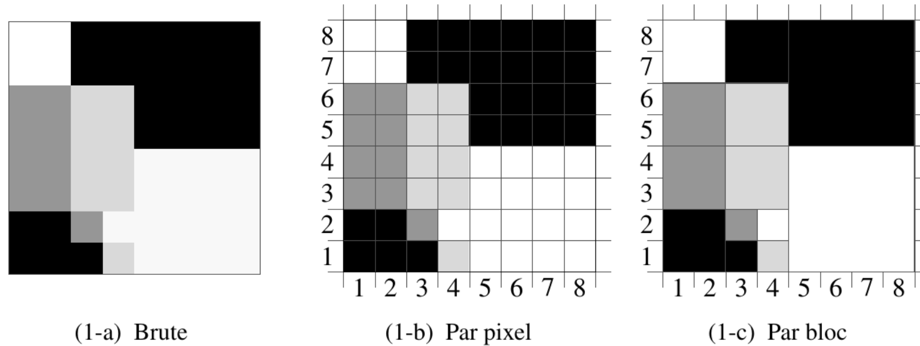
$$
\begin{aligned}
PGS(0,0) &= ??? \text{ cas de base}\\
PGS(i,j) &= PGS(i-1,j) + grille[i][j] \text{ si } j = 0\\
PGS(i,j) &= PGS(i,j-1) + grille[i][j] \text{ si } i = 0\\
PGS(i,j) &= max(PGS(i-1,j), PGS(i,j-1)) + grille[i][j] \text{ cas général}
\end{aligned}
$$

21. Que vaut $PGS(0,0)$?
22. Quelle valeur de PGS veut on calculer pour avoir la solution à notre problème ?
23. Programmer l'approche ascendante en C : on écrira une fonction
    `int approche_ascendante(int** grille, int n, int m)` qui renvoie la plus grande somme possible. On supposera écrite une fonction `int max(int a, int b)` qui calcule le maximum de deux entiers.
24. Quelle est la complexité de la fonction `approche_ascendante` ?
25. On a obtenu la valeur de la somme maximale, mais pas le chemin associé.
    Écrire une fonction `int* reconstruit(int** grille, int n, int m, int** pgs)` qui reconstruit le chemin associé à la meilleur somme à partir de la matrice contenant toutes les valeurs de PGS.
    On utilisera la même représentation de chemin que dans la partie "Approche exhaustive".

5 Arbres quaternaires

Ce sujet est librement inspiré du sujet CCINP Informatique MP de 2013

Un arbre quaternaire est un arbre dans lequel tout noeud à quatre enfants. Dans ce problème, on va utiliser des arbres quaternaires pour représenter de manière plus compacte des images en niveau de gris. Dans toute la suite, pour simplifier les programmes, nous nous limitons à des images carrées dont la longueur du côté est une puissance de 2.



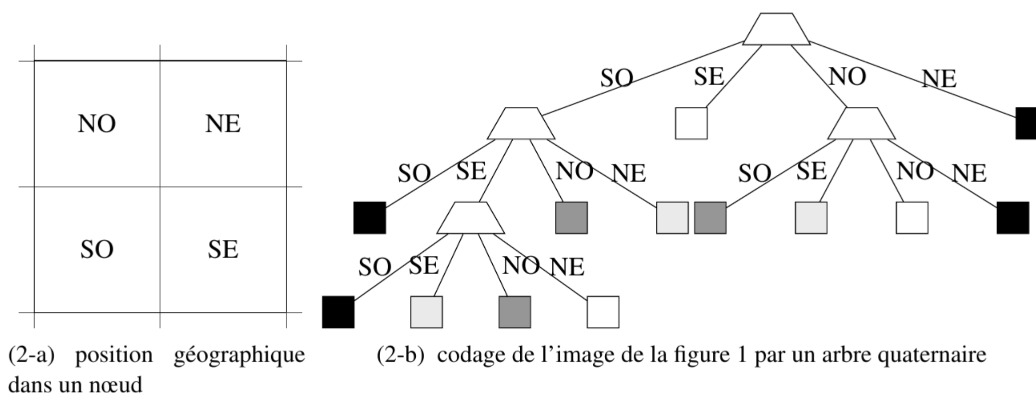
(1-a) Brute (1-b) Par pixel (1-c) Par bloc

FIGURE 1: images

Les images des figures 1 et 2 illustrent ce principe. L'image brute (1-a) contient des rectangles de différentes couleurs. Cette image est composée de pixels, visible dans la figure (1-b). Les pixels sont tous des carrés et on ne peut pas décomposer l'image en élément plus petits. En temps normal, pour stocker l'image, il faut stocker la couleur de chaque pixel, par exemple dans une matrice (ici de taille 8×8).

Pour réduire la taille de l'espace de stockage, on effectue un découpage dichotomique (figure (1-c)) : on coupe l'image de quatre carrés nommés NO, SE, SO, SE, tels qu'illustrés sur la figure (2-a). Si un carré est de couleur uniforme, comme le carré SE dans la figure (1-c), alors on dit que le carré SE forme un bloc et on le stockera comme tel. Si un carré n'est pas de couleur uniforme, comme SO dans la figure (1-c), alors on le découpe lui-même en 4 et on regarde si ses carrés NO, SE, SO, SE forment des blocs.

On reproduit cette méthode jusqu'à avoir trouvé que des blocs ou ne plus avoir que des pixels seuls, qui forment chacun un bloc.

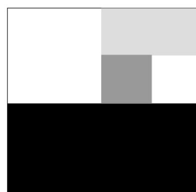


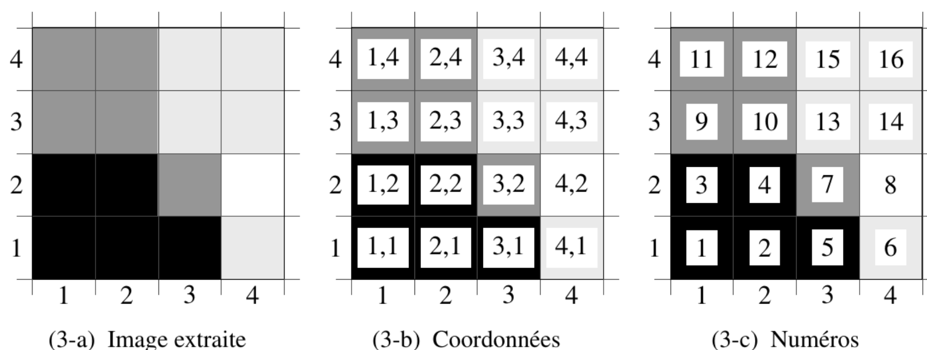
(2-a) position géographique dans un nœud (2-b) codage de l'image de la figure 1 par un arbre quaternaire

FIGURE 2: arbre quaternaire

Les blocs sont stockés sous forme d'arbre quaternaire : la racine représente l'image entière et elle a 4 fils représentant les carrés NO, SE, SO, SE. Les blocs sont les feuilles, et tout noeud représentant un carré qui n'est pas de couleur uniforme a 4 enfants. La figure (2-b) montre l'arbre permettant de représenter le découpage de la figure (1-c)

26. Dessiner l'arbre quaternaire associé à l'image suivante (on pourra écrire le nom des couleurs : blanc, noir, gris clair, gris foncé plutot que colorier les petits carrés) :





(3-a) Image extraite (3-b) Coordonnées (3-c) Numéros

FIGURE 3: identification des pixels

Pour identifier les pixels, on leur affecte des coordonnées, comme montré dans la figure (3-b) et un numéro, dans l'ordre illustré dans la figure (3-c)

1. Représentation de l'arbre quaternaire en Ocaml

L'arbre quaternaire en lui-même est composé de 2 types de noeuds : les noeuds `Division` qui sont des noeuds internes et des noeuds `Bloc` qui sont des feuilles. Il n'y a pas d'arbre vide.

Chaque noeud n, peu importe sa nature, contient 3 informations : une abcisse, une ordonnée et une taille (dans cet ordre). L'abcisse et l'ordonnée sont les coordonnées du coin en bas à gauche du carré représenté par le noeud et la taille est la longueur du côté du carré représenté par le noeud.

Chaque noeud `Division` possède en plus quatre fils et chaque noeud `Bloc` possède en plus une couleur, représentée par un nombre entre 0 et 255 (qui représente un niveau de gris, une nuance de couleur entre noir et blanc)

Les ordonnées, abcisses et taille des noeuds de l'arbre doivent toujours être logiques au vu de la position des blocs dans la grille.

Les notions de hauteur d'un arbre quaternaire et de profondeur dans un arbre quaternaire sont les mêmes que dans les arbres d'arité quelconque étudiés en cours.

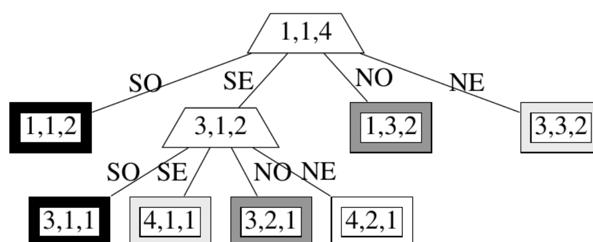


FIGURE 4: structure de données correspondant à l'image (3-a)

On utilise le type `quater` suivant pour représenter l'arbre :

```
type quater =
 | Division of int * int * int * quater * quater * quater * quater
 | Bloc of int * int * int * int;;
```

En associant les valeurs 0, 150, 234, 255 aux couleurs noir, gris foncé, gris clair et blanc, l'expression suivante :

```
let b11_2 = Bloc( 1, 1, 2, 0) in (* SO racine *)
let b31_1 = Bloc( 3, 1, 1, 0) in (* SO du SE racine *)
let b41_1 = Bloc( 4, 1, 1, 234) in (* SE du SE racine *)
let b32_1 = Bloc( 3, 2, 1, 150) in (* NO du SE racine *)
let b42_1 = Bloc( 4, 2, 1, 255) in (* NE du SE racine *)
let d31_2 = Division( 3, 1, 2, b31_1, b41_1, b32_1, b42_1) in (* SE racine *)
let b13_2 = Bloc( 1, 3, 2, 150) in (* NO racine *)
let b33_2 = Bloc( 3, 3, 2, 234) in (* NE racine *)
Division( 1, 1, 4, b11_2, d31_2, b13_2, b33_2) (* racine *)
```

est alors associée à l'arbre quaternaire représenté graphiquement sur la figure 4

27. Définir en Ocaml l'arbre quaternaire que vous avez dessiné à la question 26. Les couleurs sont les mêmes que pour l'exemple ci-dessus.

2. Opérations

28. Ecrire en Ocaml une fonction `scinder : quater -> quater` qui :
    - si la racine de a est une division, renvoie a
    - si la racine de a est un bloc `Bloc(x,y,t,c)`, renvoie un arbre quaternaire dont la racine est une division contenant quatre blocs de couleur `c`.
      Les coordonnées et les tailles des blocs et de la division doivent être cohérentes à un découpage en 4 du carré du bloc initial (qui a comme coin inférieur gauche (x, y) et est de taille t).
29. Ecrire en Ocaml une fonction `fusionner : quater -> quater -> quater -> quater -> quater` telle que l'appel `fusionner so se no ne` sur quatre arbres quaternaires valides `so`, `se`, `no` et `ne` renvoie un arbre quaternaire valide codant l'image dont les parties Sud-Ouest, Sud-Est, Nord-Ouest et Nord-Est sont codées par `so`, `se`, `no` et `ne`.
    On supposera sans le vérifier que les abscisses, ordonnées et tailles de `so`, `se`, `no` et `ne` sont cohérentes avec leur position dans l'image représentée par l'arbre renvoyé.
30. Ecrire en Ocaml une fonction `hauteur : quater -> int` qui calcule la hauteur de l'arbre donné en entrée.
    L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
31. On considère un carré de côté t et dont le pixel inférieur gauche a comme coordonnées (x_0, y_0). Exprimer les valeurs que peuvent prendre x et y si le pixel (x, y) est dans le carré.
    Exprimer les valeurs que peuvent prendre x et y si le pixel (x, y) est dans le sous-carré SE puis le sous-carré NO.
32. En s'aidant de la question précédente, écrire en Ocaml une fonction `consulter : int -> int -> quater -> int` telle que l'appel `consulter x y a` sur un pixel d'abscisse x et d'ordonnée y et sur un arbre quaternaire valide a tel que le pixel (x, y) soit contenu dans l'image représentée par l'arbre a, renvoie la couleur du pixel (x, y).
    L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre a. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
33. Écrire en Ocaml une fonction `peindre : int -> int -> int -> quater -> quater` telle que l'appel `peindre x y c a` sur un pixel d'abscisse x et d'ordonnée y, une couleur c et un arbre quaternaire valide a renvoie un arbre quaternaire valide a'. Le pixel (x, y) doit être contenu dans l'image représentée par l'arbre a. L'arbre renvoyé a' représente une image contenant les mêmes couleurs que l'image représentée par l'arbre a sauf pour le pixel de coordonnées (x, y) dont la couleur sera c.
    L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre a. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
    Attention, on veut bien ici colorier le pixel de coordonnées (x, y) et pas le bloc entier qui contient le pixel (x, y). On suppose que la couleur initiale du pixel n'est pas c.

3. Validité

On dit que l'arbre quaternaire a est valide si et seulement si :

- les abcisses, ordonnées et tailles de chaque noeud sont strictement positives,
- les tailles des quatres fils f_{SO}, f_{SE}, f_{NO} et f_{NE} de chaque division d sont identiques et égales à la moitié de la taille de d,

- les abcisses et ordonnées des quatres fils f_{SO}, f_{SE}, f_{NO} et f_{NE} de chaque division d sont cohérentes avec l'abcisse, l'ordonnée et la taille de d et avec les positions géographiques des fils (SO pour f_{SO}, SE pour f_{SE}, ...)
- au moins deux des quatres sous-arbres enracinés en les fils f_{SO}, f_{SE}, f_{NO} et f_{NE} de chaque division d contiennent des blocs de couleurs différentes. (Autrement dit, on a jamais divisé un carré de couleur uniforme)

34. Écrire en Ocaml une fonction `valider : quater -> bool` telle que l'appel `valider a` sur un arbre quaternaire valide a renvoie **`true`** si l'arbre est valide, c'est à dire s'il vérifie les propriétés énoncées ci-dessus et **`false`** sinon.
    Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.