Devoir en temps libre n° 3

Ce devoir est à préparer pour le lundi 23 avril 2018. Les questions de la partie I et II pour lesquelles apparaît le symbole ✍ doivent être rédigées au propre et sont à rendre, les autres sont à chercher au brouillon (je vous fais confiance). Le code CAML de la partie III est à rendre en début du TP, comme d'habitude, sous la forme d'un unique fichier `.ml` dont le nom sera obligatoirement et exactement `<login>.ml` où `<login>` est votre nom d'utilisateur du laboratoire d'informatique (tout en minuscules avec les tirets). Votre programme devra respecter scrupuleusement les noms et les types indiqués (ou un type plus général). On prendra bien soin de tester toutes les fonctions sur des cas triviaux, puis un peu plus complexes.

I Quelques exemples de calculs de complexité

Rappel 1

Notons $\mathcal{D}_n$ l'ensemble des données de taille $n \in \mathbb{N}$ d'un problème et coût(d) la complexité (ou le coût) d'un algorithme sur une donnée d (en nombre d'opérations élémentaires). La complexité dans le pire cas *est définie pour $n \in \mathbb{N}$ par $C(n) = \sup_{d\in\mathcal{D}_n}$ coût(d).*

I.1 Retour sur le DM n° 2

EXERCICE 2 (*Minimum et maximum d'une liste*) De nombreuses solutions proposées ont eu la forme suivante :

OCAML
```
let rec min_et_max liste =
  match liste with
  | [] -> failwith "Liste vide"
  | tete :: [] -> (tete, tete)
  | tete :: queue ->
    (min tete (fst (min_et_max queue)),
     max tete (snd (min_et_max queue)))
```

1. ✍ Montrer que si $n \geq 1$, $C(n) = 2C(n-1) + \Theta(1)$.
2. ✍ Montrer que $\sum_{k=1}^{n} \frac{1}{2^k} = \Theta(1)$.
3. ✍ Pour $k \in \mathbb{N}$, on pose $u_k = \frac{C(k)}{2^k}$. Montrer que pour $k \geq 1$, $u_k - u_{k-1} = \Theta(\frac{1}{2^k})$.
4. ✍ En déduire que $C(n) = \Theta(2^n)$.
5. ✍ Décrire précisément le comportement de *tous* les appels récursifs de la fonction sur une liste de taille 4. Expliquer pourquoi la complexité est bien exponentielle.
6. Proposer une modification de la fonction pour obtenir une complexité linéaire (c'est la version donnée par la plupart d'entre vous).
7. Vérifier que *votre* solution était bien de complexité linéaire et la modifier si ce n'était pas le cas.

PROBLÈME 1 *Représentation des polynômes*

Dans la question 7., la fonction `nettoyage` revenait à supprimer les 0 en fin de liste. La plupart des solutions proposées étaient :

OCAML
```
let rec nettoyage polynome =
  match List.rev polynome with
  | [] -> []
  | 0. :: queue -> nettoyage (List.rev queue)
  | _ -> polynome
```

1. Montrer que la complexité de cette fonction est quadratique.
2. Proposer une solution de complexité linéaire.

I.2 Complexité du tri fusion

1. Essayer de réimplémenter le tri fusion sans regarder ni l'énoncé ni le corrigé. Si vous bloquez, reprenez la description du tri donnée dans le TP n° 5. Enfin, vérifiez votre solution à l'aide du corrigé du TP n° 5.
2. Déterminer la complexité (dans le pire cas) des fonctions `division` et `fusion`.

On note $C(n)$ la complexité (dans le pire cas) du tri fusion pour une liste en entrée de longueur $n \in \mathbb{N}$. On pose $c_0 = C(0)$ et $c_1 = C(1)$.

3. Montrer que pour $n \geq 2$ cette complexité vérifie une équation de la forme
$$C(n) = aC\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + bC\left(\left\lceil \frac{n}{2} \right\rceil\right) + c_n \quad (1)$$
avec $a, b \in \mathbb{N}$, $a + b \geq 1$ et avec $(c_n)_{n\geq 1}$ une suite positive croissante d'entiers telle que $c_n = \Theta(n)$.
4. On suppose dans un premier temps que $n = 2^k$ est une puissance de 2 et on note, pour $k \in \mathbb{N}$, $u_k = C(2^k) = C(n)$ et $v_k = \frac{u_k}{2^k}$.
5. ✍ Montrer que $v_k = \Theta(k)$.
6. ✍ En déduire que $C(n) = \Theta(n \log n)$.

On revient au cas général, avec $n \in \mathbb{N}$.

7. Montrer que la suite $(C(n))_{n\in\mathbb{N}^*}$ est croissante. *Indication : procéder par récurrence forte.*
8. ✍ En déduire que $C(n) = \Omega(n \log n)$ et $C(n) = O(n \log n)$ et que donc que $C(n) = \Theta(n \log n)$.
9. Que pensez-vous de cette complexité par rapport à celle du tri par insertion ou celle du tri par sélection ?

II Autour du tri rapide

Une séquence s de taille n de valeurs v_i avec $\{1 \leq i \leq n\}$ est notée $(v_n, \ldots, v_1)$. Sa taille n est notée $|s|$. Une même valeur v peut figurer plusieurs fois dans s, nous noterons $|s|_v = \text{card}(\{i \in \mathbb{N} / \ v_i = v\})$ le cardinal de v dans s, c'est-à-dire le nombre de fois que v figure dans s. Le symbole de Kronecker, noté δ, est tel que δ_{v_1,v_2} vaut 1 si $v_1 = v_2$ et 0 sinon.

On considère les deux fonctions suivantes en CAML :

OCAML
```
let rec partition liste pivot =
  match liste with
  | [] -> ([], pivot, [])
  | tete :: queue ->
    let (petits, milieu, grands) = (partition queue pivot) in
    if (tete <= milieu) then
      (tete :: petits, milieu, grands)
    else
      (petits, milieu, tete :: grands)
```

OCAML
```
let rec tri liste =
  match liste with
  | [] -> []
  | tete :: queue ->
    let (petits, milieu, grands) = (partition queue tete) in
    tri petits @ milieu :: tri grands
```

1. Quel est le type de ces deux fonctions ?
2. Soit la constante définie par `let exemple = [3; 1; 4; 2];;`. Détailler soigneusement les étapes du calcul de `tri exemple;;` en précisant, pour chaque appel aux fonctions `partition` et `tri`, la valeur des paramètres et du résultat.
3. Expliquer, de manière informelle, ce que font les deux fonctions, en détaillant bien le rôle de chacune des variables. Des illustrations seront appréciées.
4. ✍ Expliquer en quoi ce tri, dit « tri rapide », illustre le paradigme *diviser pour régner*, en indiquant bien à quoi correspondent les trois étapes.

Soit les entiers e et v et les séquences d'entiers $p = (p_n, \ldots, p_1)$ de taille n, $g = (g_r, \ldots, g_1)$ de taille r et $d = (d_s, \ldots, d_1)$ de taille s, telles que `(g, v, d) = (partition p e)`. On veut montrer que

(a) $e = v$
(b) $\forall i \in [\![1, n]\!],\ |p|_{p_i} = |g|_{p_i} + |d|_{p_i}$
(c) $0 \leq s \leq n$ et $0 \leq r \leq n$
(d) $n = s + r$
(e) $\forall i \in [\![1, r]\!],\ g_i \leq v$
(f) $\forall i \in [\![1, s]\!],\ v < d_i$

On va montrer cette propriété par récurrence sur $n \in \mathbb{N}$, taille de la séquence argument de la fonction `partition`. On note donc H_n la propriété ci-dessus, qui dépend de $n \in \mathbb{N}$.

5. Montrer très soigneusement que H_0 est vraie.

Soit $n \in \mathbb{N}$, on suppose que H_n est vraie au rang n et on veut montrer que c'est encore le cas au rang $n + 1$.

Soit les entiers e et v', et les séquences d'entiers $p' = (p'_{n+1}, p'_n, \ldots, p'_1)$ de taille $n + 1$, $g' = (g'_{r'}, \ldots, g'_1)$ de taille r' et $d' = (d'_{s'}, \ldots, d'_1)$ de taille s', telles que `(g', v', d') = (partition p' e)`. On note alors $p = (p_n, \ldots, p_1) = (p'_n, \ldots, p'_1)$ la séquence d'entier de taille n, $g = (g_r, \ldots, g_1)$ de taille r et $d = (d_s, \ldots, d_1)$ de taille s, telles que `(g, v, d) = (partition p e)`.

6. Écrire ce que donne l'hypothèse de récurrence.

7. On suppose que $p'_{n+1} \leq v$. Montrer très soigneusement qu'alors H_{n+1} est vraie.
8. Conclure. On remarquera simplement que le cas $p'_{n+1} > v$ se traite de manière analogue, sans le détailler.

Soit la séquence $p = (p_m, \ldots, p_1)$ de taille m, soit la séquence $r = (r_n, \ldots, r_1)$ telle que `r = tri p`. On veut montrer de même que

(a) $m = n$

(b) $\forall i \in [\![1, m]\!], |r|_{p_i} = |p|_{p_i}$

(c) $\forall i \in [\![1, n-1]\!],\ r_i \leq r_{i+1}$.

9. Montrer très soigneusement ce résultat.
10. Montrer que le calcul des fonctions `partition` et `tri` se termine quelles que soient les valeurs de leurs paramètres respectant le type des fonctions.

Pour les calculs de complexité, on comptera comme opération élémentaire uniquement les comparaisons d'éléments. Toutes les autres opérations auront un coût nul. On note $P(n)$, respectivement $T(n)$, la complexité de la fonction `partition`, respectivement `tri`, dans le pire des cas. $T(n)$ est donc un majorant du nombre de comparaisons effectuées pour trier une entrée de taille n quelconque. Quand on demande des exemples réalisant un pire cas, on ne demande pas une liste particulière, mais plutôt la forme de telles listes, en fonction du paramètre n (par exemple une liste triée par ordre croissant ou décroissant, une liste constante, etc.).

11. Estimer $P(n)$. Donner des exemples qui correspondent au pire des cas.
12. Que valent $T(0)$ et $T(1)$?

On suppose que le pire des cas pour la fonction `tri` correspond à celui où lors de chaque appel, en dehors du cas de base, l'une des deux partitions est vide.

13. ✍ Donner un exemple de forme de liste aboutissant à ce pire cas.
14. ✍ Trouver, dans ce cas précis, une relation de récurrence vérifiée par $T(n)$ en fonction de $T(n-1)$, pour $n \in \mathbb{N}^*$.
15. ✍ Montrer qu'alors, pour $n \in \mathbb{N}^*$, $T(n) = \frac{n(n-1)}{2}$.

On va maintenant montrer rigoureusement que ce cas est bien le pire des cas, c'est-à-dire que $T(n) = \frac{n(n-1)}{2}$.

16. Justifier que $T(n) = \max_{0 \leq p \leq n-1} \left(T(p) + T(n-p-1)\right) + n - 1$ pour $n \geq 1$.
17. À l'aide d'une récurrence forte sur $n \in \mathbb{N}^*$, montrer le résultat attendu. On pourra étudier la fonction $p \mapsto p(p-1) + (n-p-1)(n-p-2)$ et montrer qu'elle atteint son maximum sur $\{0, \ldots, n-1\}$ à ses extrémités.

La complexité dans le pire des cas est donc de l'ordre de $\Theta(n^2)$ ce qui n'est pas très satisfaisant. Nous allons montrer que la complexité *en moyenne*, que l'on note $T_{\text{moy}}(n)$, est cependant bien meilleure. On fait ici l'hypothèse forte consistant à supposer que lors de chaque appel récursif, la position de la valeur du pivot dans la liste une fois triée est équiprobable. On a alors pour $n \in \mathbb{N}^*$:

$$T_{\text{moy}}(n) = \frac{1}{n} \sum_{k=0}^{n-1} \left(T_{\text{moy}}(k) + T_{\text{moy}}(n-k-1)\right) + n - 1$$

18. Montrer que pour tout $n \geq 1$ on a

$$\frac{T_{\text{moy}}(n)}{n+1} - \frac{T_{\text{moy}}(n-1)}{n} = 2\frac{n-1}{n(n+1)}$$

19. En admettant que $\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$, en déduire que $T_{\text{moy}}(n) = \Theta(n \log n)$.

INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B]  // HERE. SORRY.
```

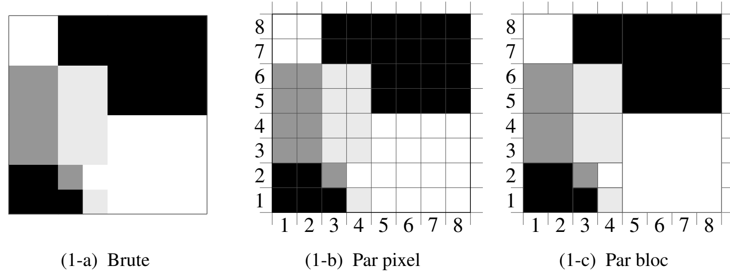
```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOGN)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): //COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

III Problème : représentation d'images par des arbres quaternaires

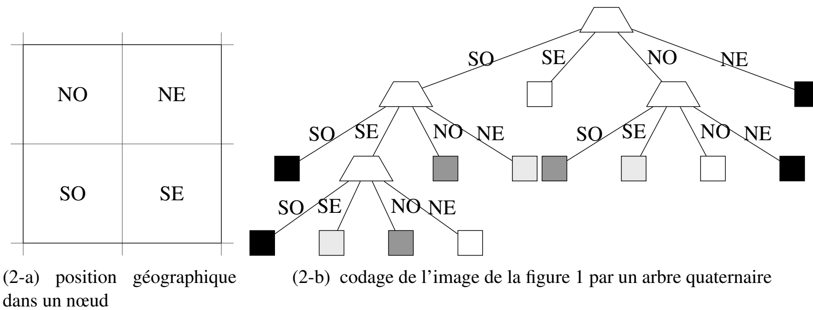
La structure d'arbre quaternaire est une extension de la structure d'arbre binaire qui permet de représenter des informations en deux dimensions. En particulier, la structure d'arbre binaire est utilisée pour représenter de manière plus compacte des images. Il s'agit de décomposer une image par dichotomie sur les deux dimensions jusqu'à obtenir des blocs de la même couleur.



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

FIGURE 1: images

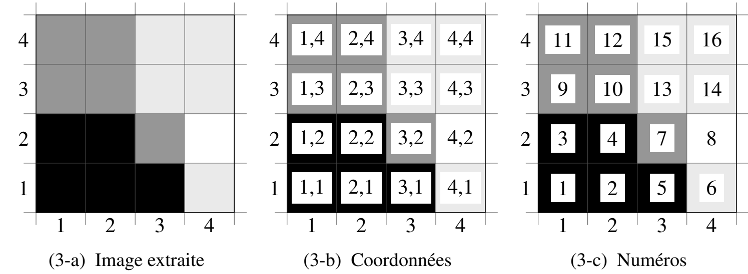
Les images des figures 1 et 2 illustrent ce principe. L'image brute (1-a) contient des carrés de différentes couleurs. Cette image est découpée uniformément en pixels (picture elements) dans l'image (1-b). Les pixels sont des carrés de la plus petite taille nécessaire. Ce découpage fait apparaître de nombreux pixels de la même couleur. Pour réduire la taille du codage, l'image (1-c) illustre un découpage dichotomique de l'image en carrés qui regroupent certains pixels de la même couleur. L'arbre quaternaire de l'image (2-b) représente les carrés contenus dans (1-c). Les étiquettes sur les fils de chaque nœud réprésentent la position géographique des fils dans le nœud : Sud-Ouest, Sud-Est, Nord-Ouest, Nord-Est comme indiqué dans (2-a).



(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

L'objectif de ce problème est l'étude de cette structure d'arbre quaternaire et de son utilisation pour le codage d'images en niveau de gris. Pour simplifier les programmes, nous nous limitons à des images carrées dont la longueur du côté est une puissance de 2. Les résultats étudiés se généralisent au cas des images rectangles de taille quelconque.



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

FIGURE 3: identification des pixels

2.1 Arbres quaternaires

Déf. III.3 (Arbre quaternaire associé à une image) Un arbre quaternaire a qui représente une image carrée est composé de nœuds, qui peuvent être des blocs ou des divisions. Les ensembles des nœuds, des divisions et des blocs, de l'arbre a sont notés $\mathcal{N}(a)$, $\mathcal{D}(a)$ et $\mathcal{B}(a)$ avec $\mathcal{N}(a) = \mathcal{D}(a) \cup \mathcal{B}(a)$. Chaque nœud $n \in \mathcal{N}(a)$ contient une abscisse, une ordonnée et une taille, notées $\mathcal{X}(n)$, $\mathcal{Y}(n)$ et $\mathcal{T}(n)$, qui correspondent aux coordonnées et à la longueur du côté de la partie carrée de l'image représentée par n. Chaque bloc $b \in \mathcal{B}(a)$ contient une couleur notée $\mathcal{C}(n)$ qui correspond à la couleur de la partie carrée de l'image représentée par b. Chaque division $d \in \mathcal{D}(a)$ contient quatre fils $f_{SO}, f_{SE}, f_{NO}, f_{NE} \in \mathcal{N}(a)$. Ces fils sont indexés par la position relative de la partie carrée de l'image qu'ils représentent dans l'image représentée par la division d : Sud-Ouest, Sud-Est, Nord-Ouest et Nord-Est.

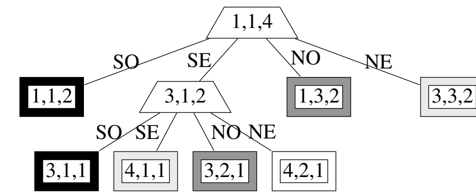


FIGURE 4: structure de données correspondant à l'image (3-a)

Exemple III.1 La figure 3 contient l'image (3-a) représentée par le sous-arbre situé au Sud-Ouest de l'arbre (2-b) page 9 ainsi que l'indication (3-b) des coordonnées de chaque point de cette image. Elle contient également la technique de numérotation des points exploitée dans la section 2.3 (page 13). La figure 4 (ci-dessus) contient l'arbre qui représente cette image dont les blocs et les divisions ont été annotés avec les coordonnées, tailles et couleurs.

Déf. III.4 (Profondeur d'un arbre quaternaire) La profondeur d'un bloc $b \in \mathcal{B}(a)$ d'un arbre quaternaire a est égale au nombre de divisions qui figurent dans la branche conduisant de la racine de a au bloc b. La profondeur d'un arbre a est le maximum des profondeurs de ses blocs $b \in \mathcal{B}(a)$.

Exemple III.2 La profondeur de l'arbre de la figure 2-b (page 9) est 3. La profondeur de l'arbre de la figure 4 (ci-dessus) est 2.

Déf. III.5 (Arbre quaternaire valide) Un arbre quaternaire a est valide, si et seulement si :
- les tailles, abscisses et ordonnées de chaque nœud $n \in \mathcal{N}(a)$ sont strictement positives ;
- les tailles des quatre fils $f_{SO}, f_{SE}, f_{NO}, f_{NE}$ de chaque division $d \in \mathcal{D}(a)$ sont identiques et égales à la moitié de la taille de d ;
- les abscisses et ordonnées des fils de chaque division $d \in \mathcal{D}(a)$ sont cohérentes avec la position géographique de chaque fils $f_{SO}, f_{SE}, f_{NO}, f_{NE}$ et avec l'abscisse, l'ordonnée et la taille de la division d ;
- au moins deux des quatre fils $f_{SO}, f_{SE}, f_{NO}, f_{NE}$ de chaque division $d \in \mathcal{D}(a)$ contiennent des blocs de couleurs différentes.

Question III.6 *Exprimer le fait qu'un arbre quaternaire a est valide sous la forme d'une propriété* $\mathrm{VAQ}(a)$.

2.2 Représentation en CaML

Un arbre quaternaire est représenté par le type `quater`. La position d'un sous-arbre dans un nœud est représentée par le type énuméré `position` contenant les valeurs `SO`, `SE`, `NO` et `NE` (représentant respectivement les sous-arbres situés au Sud-Ouest, Sud-Est, Nord-Ouest et Nord-Est d'une division). Les définitions en CaML de ces types sont :

```
type quater =
  | Division of int * int * int * quater * quater * quater * quater
  | Bloc of int * int * int * int;;
type position = SO | SE | NO | NE;;
```

Dans l'appel `Bloc(x,y,t,c)` qui construit un arbre quaternaire dont la racine est un bloc, les paramètres `x` et `y` sont les coordonnées du point situé en bas à gauche de l'image représentée par ce bloc, `t` est la longueur du côté de l'image carrée représentée par ce bloc et `c` est la couleur des points de l'image représentée par ce bloc.
Dans l'appel `Division(x,y,t,so,se,no,ne)` qui construit un arbre quaternaire dont la racine est une division, les paramètres `x` et `y` sont les coordonnées du point situé en bas à gauche de l'image représentée par cette division, `t` est la longueur du côté de l'image carrée représentée par cette division, `so`, `se`, `no` et `ne` sont quatre arbres quaternaires qui sont les quatre parties de la sub-division de l'image représentée par cette division. Celles-ci sont respectivement, `so` la partie en bas à gauche (Sud-Ouest), `se` la partie en bas à droite (Sud-Est), `no` la partie en haut à gauche (Nord-Ouest), `ne` la partie en haut à droite (Nord-Est).

Exemple III.3 En associant les valeurs `0`, `1`, `2`, `3` 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, 2) in (* SE du SE racine *)
let b32_1 = Bloc( 3, 2, 1, 1) in (* NO du SE racine *)
let b42_1 = Bloc( 4, 2, 1, 3) in (* NE du SE racine *)
let d31_2 = (* SE racine *)
  Division( 3, 1, 2, b31_1, b41_1, b32_1, b42_1) in
let b13_2 = Bloc( 1, 3, 2, 1) in (* NO racine *)
let b33_2 = Bloc( 3, 3, 2, 2) 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 (page 10).

2.2.1 Scission d'un arbre quaternaire

Question III.7 *Ecrire en CaML une fonction* `scinder` *de type* `quater -> quater` *telle que l'appel* `(scinder a)` *sur un arbre quaternaire valide* `a` *renvoie, soit un arbre quaternaire identique à l'arbre* `a` *si la racine de celui-ci est une division, soit un arbre quaternaire dont la racine est une division contenant quatre blocs de même couleur identique à celle du bloc à la racine de l'arbre* `a`*. Les coordonnées et les tailles des blocs et de la division doivent être cohérentes. Toutes les contraintes de validité de l'arbre renvoyé doivent être satisfaites sauf celle concernant les couleurs.*

2.2.2 Fusion d'arbres quaternaires

Question III.8 *Ecrire en CaML une fonction* `fusionner` *de type* `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`*. Les abscisses, ordonnées et tailles de* `so`, `se`, `no` *et* `ne` *sont cohérentes avec leur position dans l'image représentée par le résultat renvoyé.*

2.2.3 Calcul de la profondeur d'un arbre quaternaire

Question III.9 *Ecrire en CaML une fonction* `profondeur` *de type* `quater -> int` *telle que l'appel* `(profondeur a)` *sur un arbre quaternaire valide* `a` *renvoie la profondeur de l'arbre* `a`*. 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.*

2.2.4 Consulter la couleur d'un point dans un arbre quaternaire

Question III.10 *Ecrire en CaML une fonction* `consulter` *de type* `int -> int -> quater -> int` *telle que l'appel* `(consulter x y a)` *sur un point d'abscisse* `x` *et d'ordonnée* `y` *et sur un arbre quaternaire valide* `a` *tel que le point d'abscisse* `x` *et d'ordonnée* `y` *soit contenu dans l'image représentée par l'arbre* `a`*, renvoie la couleur du point d'abscisse* `x` *et d'ordonnée* `y` *dans l'image représentée par l'arbre* `a`*. 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.*

2.2.5 Peindre un point dans un arbre quaternaire

Question III.11 *Ecrire en CaML une fonction* `peindre` *de type* `int -> int -> int -> quater -> quater` *telle que l'appel* `(peindre x y c a)` *sur un point d'abscisse* `x` *et d'ordonnée* `y`*, une couleur* `c` *et un arbre quaternaire valide* `a` *renvoie un arbre quaternaire valide. Le point de coordonnées* `x` *et* `y` *doit être contenu dans l'image représentée par l'arbre* `a`*. L'arbre renvoyé représente une image contenant les mêmes couleurs que l'image représentée par l'arbre* `a` *sauf pour le point de coordonnées* `x` *et* `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.*

2.2.6 Validation d'un arbre quaternaire

Question III.12 *Ecrire en CaML une fonction* `valider` *de type* `quater -> bool` *telle que l'appel* `(valider a)` *sur un arbre quaternaire* `a` *renvoie la valeur* `true` *si l'arbre* `a` *est valide, c'est-à-dire si* VAQ(`a`) *et la valeur* `false` *sinon. 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.*

2.3 Sauvegarde et restauration

Pour sauvegarder dans un fichier les couleurs contenues dans un arbre quaternaire valide, celles-ci sont rangées dans une séquence triée selon la position des points dans l'arbre. L'ordre choisi permet de restaurer l'arbre efficacement. La figure (3-c) (page 10) indique l'ordre dans lequel les points doivent être sauvegardés. La séquence manipulée contiendra les couleurs et les numéros associés à la position de chaque couleur dans l'arbre.

2.3.1 Codage en CaML

Une séquence de positions et de couleurs est représentée par le type *sequence* dont la définition est :

```
type sequence == (int * int) list;;
```

La position figure avant la couleur associée dans la séquence.

Exemple III.4 En associant les valeurs `0`, `1`, `2`, `3` aux couleurs noir, gris foncé, gris clair et blanc, la sauvegarde de l'image de la figure (3-c) (page 10) produit la séquence :

```
[(1,0); (2,0); (3,0); (4,0); (5,0); (6,2); (7,1); (8,3);
 (9,1); (10,1); (11,1); (12,1); (13,2); (14,2); (15,2); (16,2)]
```

2.3.2 Sauvegarde d'un arbre quaternaire

Question III.13 *Ecrire en CaML une fonction* `sauvegarder` *de type* `quater -> sequence` *telle que l'appel* `(sauvegarder a)` *sur un arbre quaternaire valide* `a` *renvoie une séquence triée contenant les mêmes couleurs à la même position que l'arbre* `a`*. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre* `a` *et ne devra pas reparcourir la (ou les) séquence(s) créée(s) en dehors de leur création. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

2.3.3 Restauration d'un arbre quaternaire

Question III.14 *Ecrire en CaML une fonction* `restaurer` *de type* `sequence -> quater` *telle que l'appel* `(restaurer s)` *sur une séquence valide* `s` *renvoie un arbre quaternaire valide contenant exactement les points contenus dans la séquence* `s`*. L'algorithme utilisé ne devra parcourir qu'une seule fois la séquence* `s` *et ne devra pas reparcourir les arbres quaternaires créés en dehors de leur création. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*