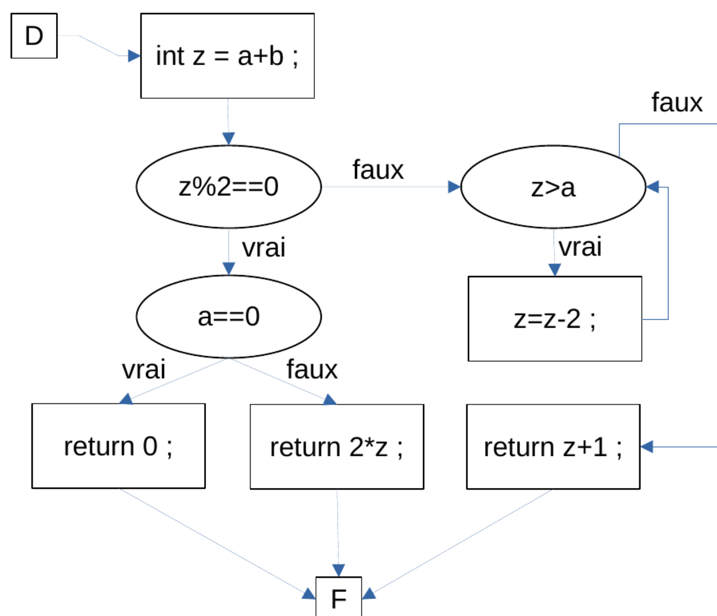
Devoir en temps limité n°4 - 3h

1 Graphe de flot de contrôle

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.*
   Le jeu $\{(a = 0, b = 2), (a = 2, b = 2), (a = 1, b = 2)\}$ convient.

2 Tri fusion

3. *Rappeler le principe du tri fusion en effectuant le tri de la liste* $[3, 0, 5, 8, 1, 2]$.
   Il faut refaire le dessin qui est dans le cours.
4. *Programmer le tri fusion en Ocaml, pour une liste d'entiers.*

```
let rec separe l = match l with
  |[] -> [],[]
  |[t] -> [t],[]
  |t::h::q -> let l1, l2 = separe q in t::l1, h::l2;;

let rec fusion l1 l2 = match l1, l2 with
  |[],_-> l2
  |_,[] -> l1
  |t1::q1, t2::q2 when t1<t2 -> t1::(fusion q1 l2)
  |t1::q1, t2::q2 -> t2::(fusion l1 q2);;

let rec tri_fusion l = match l with
  |[]|[t] -> l
  |_ -> let l1,l2 = separe l in fusion (tri_fusion l1) (tri_fusion l2);;
```

5. *Rappeler la complexité du tri fusion.*
   $O(n \log(n))$ où n est la taille de la liste.

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.*
   $t_2 = c_1(Z, T)$ et $t_3 = c_1(c_2(Z, T), Z)$

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 $\varphi(t) = 6$.*
   $\varphi(t_2) = 2$ et $\varphi(t_3) = 0$.
   On a $\varphi([\![T \oplus [\![T \oplus T]\!]]\!]) = 6$.

8. *Rappeler le principe de la preuve par induction d'une propriété P pour un ensemble $\mathcal{T}$ défini par induction avec $(\mathcal{B}, C)$.*
   cf cours.

9. *Montrer par induction structurelle que $\forall t \in \mathcal{T}$, $\varphi(t)$ est pair.*
   On raisonne par induction =sur la structure des expressions simples.
   Considérons $t \in \mathcal{T}$, alors :

   - Soit $t \in \mathbb{B}$, donc $t = Z$ ou $t = T$. On a alors $\varphi(t)$ qui vaut 0 ou 2 et qui est pair.
   - Soit $t = c_1(t', t'')$ avec t' et t'' qui vérifient la propriété. On a alors $\varphi(t) = \varphi(t') + \varphi(t'')$. Or la somme de deux nombre pairs est paire.
   - Soit $t = c_2(t', t'')$ avec t' et t'' qui vérifient la propriété. On a alors $\varphi(t) = \varphi(t') * \varphi(t'')$. Or le produit de deux nombre pairs est pair.

   Par principe d'induction structurelle, la propriété est vérifiée pour toute expressions simple.

4 Parcours de grille

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 droite, ni vers le haut.*
    On a $N = n - 1 + m - 1$. Pour aller du coin en haut gauche au coin en bas à droite, il faut aller $n - 1$ fois vers le bas et $m - 1$ fois vers la droite.

1. Approche exhaustive

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.*

```
bool suivant(int* chemin, int grandN){
  int i = grandN-1;
  int fini = false;

  while (i>=0 && !fini){
    if (chemin[i] == 1){chemin[i] = 0;
                        i-=1;}
    else {chemin[i] = 1;
          fini = true;}
  }

  if (!fini) {return true;}
  else {return false;}
}
```

12. *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 $(n - 1, m - 1)$.*
    Deux méthodes : simuler le déplacement OU vérifier qu'il y a $n - 1$ 0 et $m - 1$ 1.

```c
bool verifie_chemin(int* chemin, int n, int m){
  int iactuel = 0;
  int jactuel = 0;

  int grandN = n=m-2;
  for (int i=0;i<grandN; i+=1){
    if (chemin[i]==0){
      iactuel += 1;
    }
    else{
      jactuel +=1;
    }
  }

  return (iactuel==n-1) && (jactuel==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.*

```c
int somme_chemin(int* chemin, int** grille, int n, int m){
  int somme = grille[0][0];
  int iactuel = 0;
  int jactuel = 0;

  int grandN = n=m-2;
  for (int i=0;i<grandN; i+=1){
    if (chemin[i]==0){
      iactuel += 1;
    }
    else{
      jactuel +=1;
    }
    somme+=grille[iactuel][jactuel];
  }

  return somme;
}
```

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 = n+m-2;
  int* chemin = malloc(grandN*sizeof(int));
  for(int i=0; i<grandN; i+=1){
    chemin[i]=0;
  }//Initialiser le premier chemin avec que des 0

  int maxi = 0;

  while(!suivant(chemin, grandN)){
    if (verifie_chemin(chemin, n, m)){ //Vérifier si chemin valide
      int somme = somme_chemin(chemin, grille, n, m);
      if (somme>maxi){ //Recherche de la somme max
        maxi = somme;
      }
    }
  }

  return maxi;
}
```

15. *Combien de chemins teste-t'on dans la fonction `approche exhaustive` ? En déduire un minorant de la complexité de la fonction.*

Il existe 2^N tableaux de 0 et de 1 de taille N. (On a 2 choix pour chaque cases et les choix se multiplient ensemble)

La boucle while de la fonction effectue donc exactement 2^N tours. Peu importe la complexité d'un tour de boucle, la complexité totale sera supérieure à 2^N. La complexité est nécessairement exponentielle (ou pire).

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` *?*
Un chemin valide contient $n-1$ fois le chiffre 0 et $m-1$ le chiffre 1. N'importe quel tableau qui vérifie cette propriété est un chemin valide.
Il suffit donc de compter le nombre de tableaux qui contiennent $n-1$ fois le chiffre 0 et $m-1$ le chiffre 1.
Pour créer un tel tableau :
   - On choisit $n-1$ positions parmi les N disponibles pour placer les 0 : il y a $\binom{N}{n-1}$ possibilités.
   - On place les 1 dans les positions restantes. Il n'y a qu'une seule façon de faire.

Il y a donc $\binom{N}{n-1} = \frac{(n+m-2)!}{(n-1)!(m-1)!}$ tableaux valides.
Ce nombre est strictement plus petit que 2^N (car la somme des coefficients binomiaux parmi N vaut 2^N).
En revanche on peut montrer $\binom{N}{n-1} \geq (\frac{N}{n-1})^{n-1}$. Donc le nombre de tours de la boucle while reste exponentiel en fonction de n et m.

2. Approche gloutonne

17. *Rappeler le principe d'un algorithme glouton.*
Un algorithme glouton est une méthode simple pour résoudre un problème d'optimisation. L'algorithme effectue à chaque étape le choix qui semble optimal à cette étape.

18. *Écrire une fonction* `int approche_gloutonne(int** grille, int n, int m)` *qui calcule la solution selon la méthode gloutonne.*

```c
int approche_gloutonne(int** grille, int n, int m){
  int iactuel = 0;
  int jactuel = 0;

  int somme = 0;
  while(iactuel!=n-1 || jactuel!=m-1){
    if(i==n-1){
      jactuel += 1;
    }
    else if (j==n-1){
      iactuel+=1;
    }
    else{
      int vb = grille[iactuel+1][jactuel];
      int vd = grille[iactuel][jactuel+1];

      if (vb>vd){
        iactuel+=1;
      }
      else{
        jactuel+=1;
      }
    }

    somme += grille[iactuel][jactuel];
  }

  return somme;
}
```

19. *Quelle est la complexité de votre fonction ?*
On effectue N tours de boucle car chaque tour correspond à une étape du chemin.
Chaque tour effectue quelques opérations élémentaires.
La complexité est donc linéaire en N.

20. *Montrer sur un exemple que la méthode gloutonne n'est pas optimale.*
Sur la grille suivante :

1	1	1	1000
3	1	1	1
1	1	1	1

Le glouton va commencer par aller au 3 et trouver un somme de 8 alors que la meilleure somme est 1005.

3. Approche par programmation dynamique

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 est soit venu de $(i-1, j)$ en allant vers le bas, soit de $(i, j-1)$ en allant 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.

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

21. *Que vaut $PGS(0,0)$?*
    Le "chemin" allant de la case $(0,0)$ à la case $(0,0)$ coute $grille[0][0]$.
22. **Quelle valeur de PGS veut on calculer pour avoir la solution à notre problème ?**
    On cherche à aller à la case en bas à droite, donc on veut calculer $PGS(n-1, m-1)$.
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.*

```c
int approche_ascendante(int** grille, int n, int m){
  int** pgs = malloc(sizeof(int*)*n);
  for(int i=0; i<n; i+=1){
    pgs[i] = malloc(sizeof(int)*m);
  }

  //cas de base
  pgs[0][0] = grille[0][0];

  //On sépare les différents cas récursifs pour que ce soit plus simple à écrire
  for(int i=0; i<n; i+=1){
    pgs[i][0] = pgs[i-1][0]+grille[i][0];
  }

  for(int j=0; j<m; j+=1){
    pgs[0][j] = pgs[0][j-1]+grille[0][j];
  }

  for(int i=0; i<n; i+=1){
    for(int j=0; j<m; j+=1){
      pgs[i][j] = max(pgs[i-1][j], pgs[i][j-1])+grille[i][j];
    }
  }

  return pgs[n-1][m-1];
}
```

24. *Quelle est la complexité de la fonction `approche_ascendante` ?*
    Créer la matrice coute $O(m \times n)$. Les deux premières boucles ont un coût inférieur à la double boucle, qui a un coût de $O(nm)$.
25. *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.*

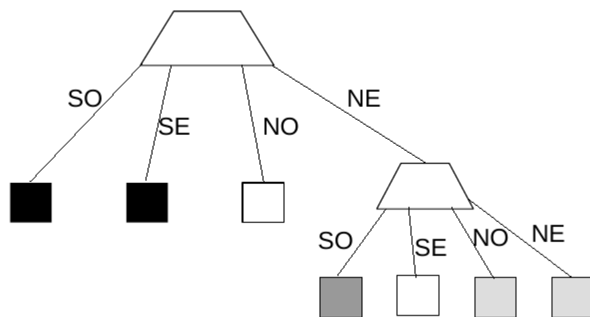
```c
int* reconstruit(int** grille, int n, int m, int** pgs){
  int grandN = n+m-2;
  int* chemin = malloc(grandN*sizeof(int));
```

```
    //Cette fois on parcourt le chemin dans l'autre sens
    int iactuel = n-1;
    int jactuel = n-1;

    for(int k=grandN-1; k>=0; k-=1){
      //Pour savoir d'où on est venu, il suffit de trouver quel terme était le plus grand dans le max
      if(iactuel>0 && pgs[iactuel][jactuel] == pgs[iactuel-1][jactuel]+grille[iactuel][jactuel]){
        chemin[k] = 0;
        iactuel -= 1;
        }
      else{
        chemin[k] = 1;
        jactuel -= 1;
      }
    }
    return chemin;
  }
```

5 Arbres quaternaires

26. *Dessiner l'arbre quaternaire associé à l'image suivante :*



1. Arbre quaternaire

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

```
let b11_2 = Bloc( 1, 1, 2, 0) in (* SO racine *)
let b31_1 = Bloc( 3, 1, 2, 0) in (* SE racine *)
let b13_1 = Bloc( 1, 3, 2, 255) in (* NO racine *)
let b33_1 = Bloc( 3, 3, 1, 150) in (* SO du NE racine *)
let b43_1 = Bloc( 4, 3, 1, 255) in (* SE du NE racine *)
let b34_1 = Bloc( 3, 4, 1, 234) in (* NO du NE racine *)
let b44_1 = Bloc( 4, 4, 1, 234) in (* NE du NE racine *)
let d33_2 = Division( 3, 3, 2, b33_1, b43_1, b34_1, b44_1) in (* NE racine *)
Division( 1, 1, 4, b11_2, d31_1, b13_1, b33_2) (* racine *)
```

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).

```
let scinder a = match a with
  |Division(x,y,t,so,se,no,ne) -> a
  |Bloc(x,y,t,c) -> Division(x,y,t,Bloc(x,y,t/2,c),Bloc(x+t/2,y,t/2,c),
                          Bloc(x,y+t/2,t/2,c),Bloc(x+t/2,y+t/2,t/2,c));;
```

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`.

Pour construire la division, on a juste besoin de l'abcisse l'ordonnée et la taille. L'abcisse et l'ordonnée sont les mêmes que pour *so*. La taille est deux fois celle des sous-arbres.

```
let scinder so se no ne = match so with
 |Bloc(x,y,t,c) -> Division(x,y,2*t,so,se,no,ne)
 |Division(x,y,t,_,_,_,_) -> Division(x,y,2*t,so,se,no,ne);;
```

30. *Ecrire en Ocaml une fonction* `hauteur : quater -> int` *qui calcule la hauteur de l'arbre donné en entrée.*

```
let rec hauteur a = match a with
 |Bloc(_,_,_,_) -> 0 (*les blocs sont de hauteur 0*)
 (*on récupère la plus grande hauteur parmi les 4 sous-arbres*)
 |Division(_,_,_,so,se,no,ne)-> 1 + max (hauteur so) (max (hauteur se) (max (hauteur no) (hauteur ne))
```

31. *On considère un carré de côté t et dont le point inférieur gauche a comme coordonnées (x_0, y_0). Exprimer les valeurs que peuvent prendre x et y si le point (x, y) est dans le carré.*

Exprimer les valeurs que peuvent prendre x et y si le point (x, y) est dans le sous-carré SE puis le sous-carré NO.

Dans le carré entier, on a $x_0 \leq x < x_0 + t$ et $y_0 \leq y < y_0 + t$

Dans le sous-carré SE, on a $x_0 + t/2 \leq x < x_0 + t$ et $y_0 \leq y < y_0 + t/2$.

Dans le sous-carré NO, on a $x_0 \leq x < x_0 + t/2$ et $y_0 + t/2 \leq y < y_0 + t$.

Dans cette question et la suivante, les points ont été accordés même si les inégalités strictes n'ont pas correctement été écrites.

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 point d'abscisse x et d'ordonnée y et sur un arbre quaternaire valide a tel que le point (x, y) soit contenu dans l'image représentée par l'arbre a, renvoie la couleur du point (x, y).*

```
let rec consulter x y a = match a with
 |Division(x0,y0,t,so,se,no,ne) -> if x < x0+t/2 then
                                     if y < y0+t/2 then consulter x y so
                                     else consulter x y no
                                   else
                                     if y < y0+t/2 then consulter x y se
                                     else consulter x y ne
 |Bloc(x0,y0,t,c) -> c (*Si on a localisé le point dans un bloc, on connait sa couleur*);;
```

33. *Écrire en Ocaml une fonction* `peindre : 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 a'. Le point (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 point de coordonnées (x, y) dont la couleur sera c.*

Il y a une phase pendant laquelle on cherche le bloc associé au point (x, y), qui est la même que dans la question précédente. une fois qu'on a trouvé le bon bloc, il faut le découper pour isoler LE pixel (x, y).

```
let rec peindre x y c a =
 |Division(x0,y0,t,so,se,no,ne) -> if x <= x0+t/2 then
                                     if y <= y0+t/2 then Division(x0,y0,t,peindre x y c so,se,no,ne)
                                     else Division(x0,y0,t,so,se,peindre x y c no,ne)
                                   else
                                     if y <= y0+t/2 then Division(x0,y0,t,so,peindre x y c se,no,ne)
                                     else Division(x0,y0,t,so,se,no,peindre x y c ne)
 |Bloc(x0,y0,t,cp) when t=1 -> Bloc(x0,y0,t,c)
 (*Quand on est dans un bloc qui est le pixel (x,y), on change la couleur*)
 |Bloc(x0,y0,t,cp) -> let ap = scinder a in (*on découpe en 4*)
                      peindre x y c ap (*on recommence avec la nouvelle division qu'on vient d'ajouter*)
```

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.*

```
let valider a =
  (*Fonction auxiliaire pour répertorier les couleurs dans un quadtree
  tab est un tableau de 256 cases qui correspondent chacune à une couleur*)
  let rec repetorie_couleurs a tab = match a with
    |Bloc(_,_,_,c) -> tab.(c)<-1
```

```
    |Division(_,_,_,so,se,no,ne) -> repertorie_couleurs so tab;
                                    repertorie_couleurs se tab;
                                    repertorie_couleurs no tab;
                                    repertorie_couleurs ne tab;
  in

  (*Fonction auxiliaire qui compte le nombre de 1 dans le tableau*)
  let rec compte_couleurs tab i =
    if i = Array.length tab then 0
    else if tab.(i) = 1 then 1+compte_couleurs tab (i+1)
    else compte_couleurs tab (i+1)
  in

  (*Fonction qui vérifie toutes les conditions*)
  let rec aux a = match a with
   |Bloc(x,y,t,_) -> x>0 && y>0 && t>0
   |Division(x0,y0,t0,so,se,no,ne) -> let coords a = match a with
                                        |Bloc(x,y,t,_) -> x,y,t
                                        |Division(x,y,t,_,_,_,_) -> x,y,t
                                      in

                                      (*On vérifie les coordonnées du noeuds puis les coordonnées et
                                      tailles des enfants*)
                                      let x1,y1,t1 = coords so in
                                      let x2,y2,t2 = coords se in
                                      let x3,y3,t3 = coords no in
                                      let x4,y4,t4 = coords so in

                                      if not (x0>0 && y0>0 && t0>0 &&
                                              x1=x0 && y1=y0 && t0/2=t1 &&
                                              x2=x0+t0/2 && y2=y0 && t0/2=t2 &&
                                              x3=x0 && y3=y0+t0/2 && t0/2=t3 &&
                                              x4=x0+t/2 && y4=y0+t/2 && t0/2=t4) then false

                                      (*Si tout est bon, on vérifie les couleurs*)
                                      else begin let tab = Array.make 256 0 in
                                           repertorie_couleurs a tab;
                                           aux so && aux se && aux no && aux ne && compte_couleurs tab >=2
                                           end
  in aux a;;
```